DUBLIN CITY UNIVERSITY
SCHOOL OF ELECTRONIC ENGINEERING

Rust-Powered Multi-Channel Energy Monitor for Smart Home Applications

Tiarnan Ryan

August 2024-2025

BACHELOR OF ENGINEERING

IN

ELECTRONIC AND COMPUTER ENGINEERING

MAJORING IN

THE INTERNET OF THINGS

Supervised by

Dr Derek Molloy

Acknowledgements

A special thank you to Dr Derek Molloy for the guidance and inspiration for this project, and to William Roarty for the support in securing the necessary hardware.

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited in the assignment references. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

I have read and understood the DCU Academic Integrity and Plagiarism policy at https://www.dcu.ie/policies/academic-integrity-plagiarism-policy

Name: Tiarnan Ryan Date: 18/03/2025

Abstract

In this report, I detail the design and implementation of a fully encapsulated energy monitoring system for smart home applications. The goal was to design an open-source rust-powered multi-channel energy monitor for smart home applications. To achieve this as efficiently as possible, the client-server model was built in bare metal rust leveraging rust's async/await functionality. The result was a self-reliant energy monitor that successfully displayed the collected data on the GUI in real time. While it was possible to achieve this, the resource constraints of the chosen microcontrollers negatively impacted the overall speed and scalability of the system even with the efficiency measures in place.

Table of Contents

Chapter 1 - Introduction

In the current day and age, there is an enormous reliance on energy in order to keep things running smoothly in both industrial and household settings. Unfortunately, energy is also getting increasingly expensive. It then of course follows that monitoring and tracking energy expenditure would be of interest. However, monitoring the energy being consumed by a device is not only of financial interest but can also give clues as to how the machine is performing. An overuse of energy can hint at wear and tear of a machine or a large spike in the energy could point to a short circuit.

1.1 Equipment and Software

1.1.1 The ESP32C3

The ESP32C3 is a single-core RISC-V microcontroller produced by Esspresif for the purpose of embedded systems. The chip provides both WiFi and Bluetooth capabilities along with 400kB of SRAM as usable memory. The ESP32C3 also provides plenty of configurable GPIOs for peripheral functionality[1]. Along with the ESP32C3, there is the esp-hal library which provides hardware abstraction layers for bare-metal (no_std) implementations. There is also the esp-idf-hal which is available for std applications[2],[3].

1.1.2 The ADS1015

The ADS1015 is an Analog to Digital Converter (ADC) which interfaces with the I2C protocol. It is capable of sampling up to 3.3kHz at a resolution of 12 bits. This is used to interpret the analog signal and also provide a buffer to protect the ESP32C3[4].

1.1.3 The ADALM2000 & Scopy

In order to safely generate the signals for testing, a scaled down version can be generated using an ADALM2000. This is a multifunctional device capable of delivering a voltage supply and generating an appropriate signal for testing. Scopy provides an interface in order to use the ADALM2000. There are many other functionalities available but they are beyond the scope of this report.

1.1.4 Embassy

Embassy is a highly adaptable ecosystem designed for programming in Rust in a no_std environment. It provides both blocking and async functionality through the embassy-executor, along with alternative time and networking Hardware Abstraction Layers (HALs) compatible with esp-hal[5].

1.1.5 Qt Creator, PySide & QML

Qt Creator is an IDE designed for easy frontend design. PySide provides a way to use the Qt framework with python as opposed to C++. QML is a markup language for designing the physical layout of the GUI, whose backend is powered by Python[6].

1.1.6 Rust-analyzer, Cargo & Git

The standard toolchain for Rust development includes both the rust-analyzer LSP (Language Server Protocol) and cargo, which is the package manager for Rust. These were integrated using VSCode which provides the LSP through extensions. For version control, GitHub was used. GitHub guarantees that no code is lost and allows for the tracking of changes over time as the project was developed.

1.2 System Overview

The ESP32C3 clients interface with the ADS1015 ADCs in order to capture the readings from the source. The client ESPs then format the data received and pass it to the server. The server then asynchronously processes and timestamps the received data before loading the converted readings to the HTTP socket. The GUI will then poll the HTTP socket and graph/further process the data. A further breakdown of each element is covered later in the report.

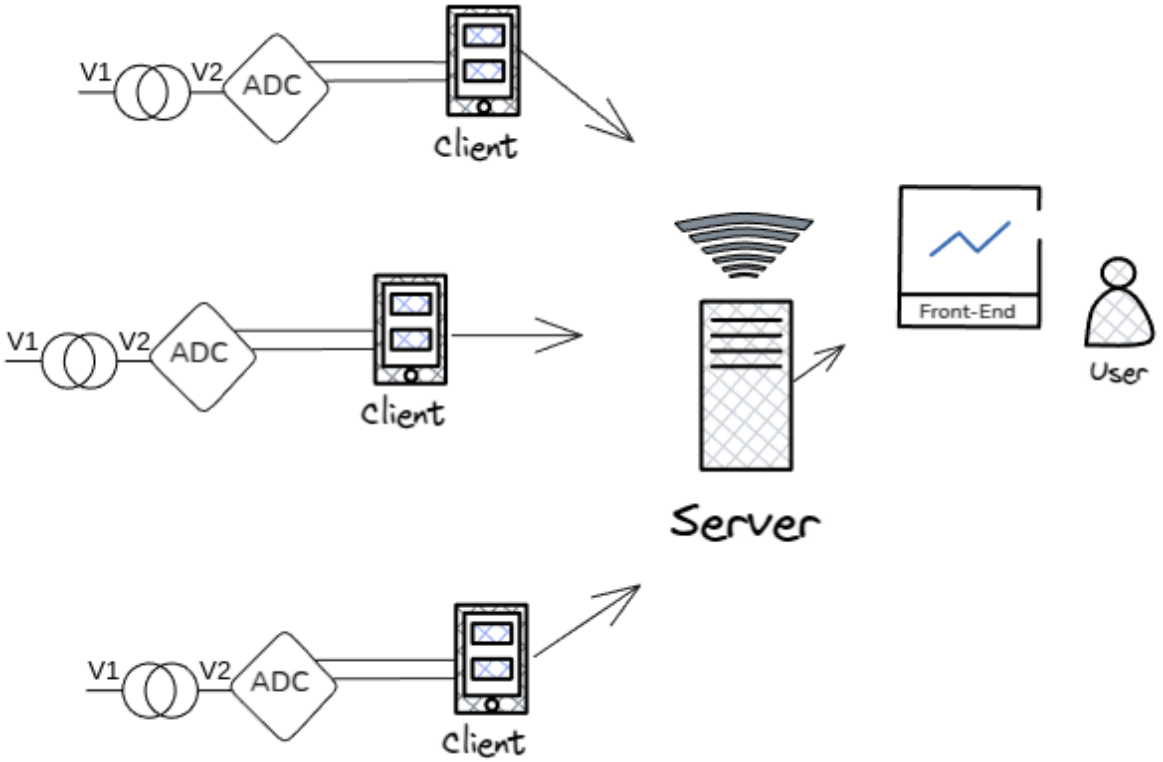


Fig 1.1. High-level overview of the system architecture.

Chapter 2 - Technical Background

In this chapter, there is an explanation behind the key technologies used to make the project. This section aims to inform about the chosen technologies, but the rationale for choosing them is explained in later sections.

2.1 Rust

Rust is a modern programming language whose applications are fairly broad, but it really excels in embedded applications. This is because of the mix between speed which tends to be faster than Java and the compile-time guarantees that C++ handles manually. A lot of these speed gains over Java can be attributed to Rust's borrow checker as opposed to Java's runtime garbage collector[7],[8]. One of the most notable benefits of Rust is that it guarantees *memory safety.* This is significant, as "around 70% of high security bugs are

memory unsafety problems (C/C++ pointer mistakes)"[9]. These comparisons are further discussed in the design section.

```
fn generate_data(rng: &mut Rng) -> f32 {
        let mut x = rng.random() as f32; //gets random number
        x = PRECALCED_RECIP*x;
        return x;
    }
```

Listing 2.1. A simple Rust function declaration which generates random test data.

The structure of a Rust project is vital for both maintainability, collaboration and cargo integration. The most important places from the programmer's point of view are the src folder, the Cargo.toml file and the .cargo/config.toml file. The src folder contains all of the actual Rust code files while the Cargo.toml file contains the packages which you use in your code. Finally, the .cargo/config.toml file contains the compilation target for your implementation which depends on the architecture (conditional compilation is available for more flexible code). These .toml files are written in Tom's Obvious, Minimal Language (TOML) which is both human and machine readable language ideal for data structures[10].

2.1.1 Cargo

Cargo is the package manager for Rust. It focuses on managing dependencies and the overall project structure. The packages that are to be used can be manually declared within the Cargo.toml file, or added via the command line which automatically generates the TOML. Cargo also acts as a build tool, providing dependency checking (cargo check), building and even flashing utilities[11].

When programming using bare-metal rust, it is important to ensure that the packages are correctly configured for this environment, as it is often the case that the default package added via the command line is incompatible. This is often why it is more practical to manually configure and alter the Cargo.toml file directly rather than through the command line.

```
[package]
name = "server"
version = "0.1.0"
authors = ["Tiarban <tiarnanryan13@gmail.com>"]

[dependencies]
log = "0.4"
esp-idf-svc = { version = "0.51", features = ["critical-section",
"embassy-time-driver", "embassy-sync"] }

[build-dependencies]
embuild = "0.33"
```

Listing 2.2. A basic Cargo.toml file

2.1.2 Asynchronous Programming with Embassy

Asynchronous programming is a lightweight approach to handling concurrency[12]. An asynchronous task can be defined as a task which is non-blocking. This allows multiple tasks to work in conjunction with each other without having to wait for each other to finish. This avoids the time slicing that occurs when dealing with multithreading on a single core.

Asynchronous programming is implemented using the async/await framework, where a task is declared asynchronous with the async keyword. When an async task is reached, it immediately returns an object called a future to the executor. The executor then polls the future until the final value is ready. When, during a task, an await is encountered, control is yielded back to the executor which polls the awaited function until the value is ready (eg. a timer is awaited until the time has elapsed). Practically speaking, the main function acts as the entry point, accepting a spawner as an argument. Then, the asynchronous tasks are spawned from main using the spawner[13].

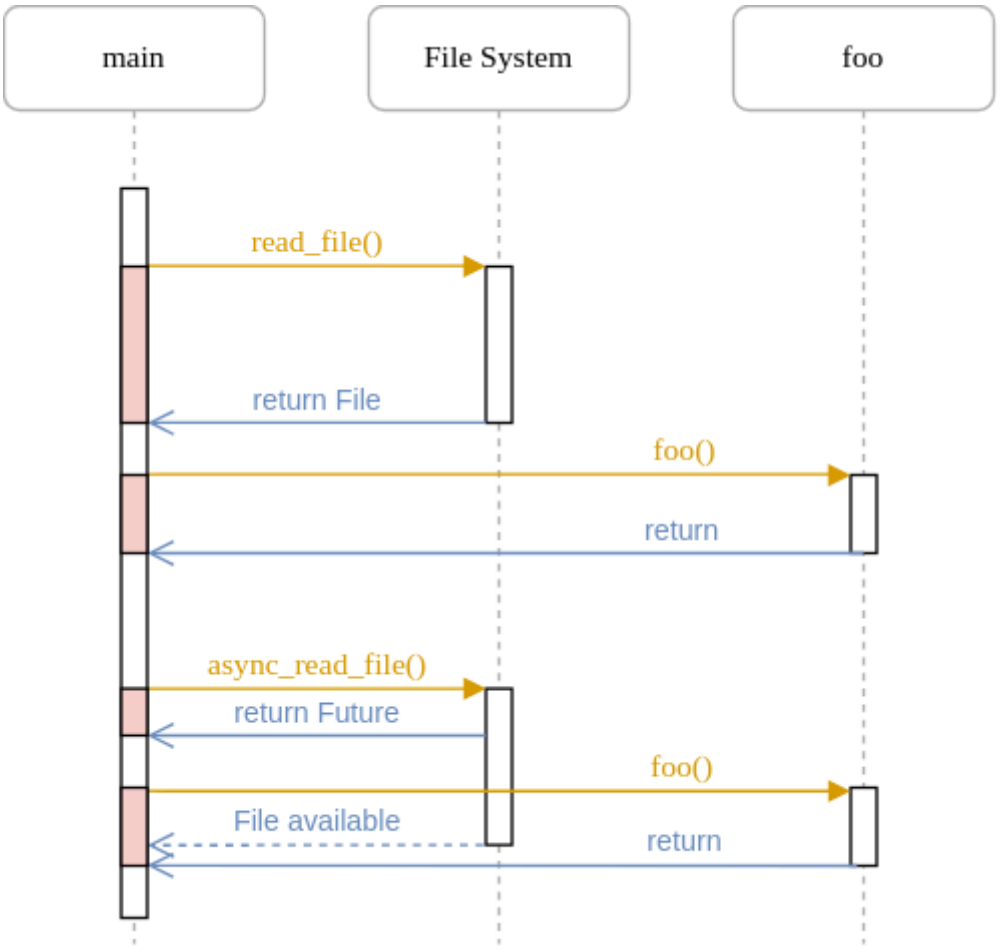


Fig 2.1. Basic asynchronous programming example vs a synchronous equivalent[14].

In order for asynchronous programming to work, it needs an ecosystem which supports it. This is provided by Embassy through the embassy-executor, embassy-sync and other associated packages. The embassy-executor provides the executor, whereas the embassy-sync package provides a mutex which manages shared memory between tasks.

```
#[embassy_executor::task]
async fn wait_1second () {
   Timer::after(Duration::from_millis(1000)).await;
}
#[esp_hal_embassy::main]
async fn main(spawner: Spawner) {
   spawner.spawn(wait_1second()).ok();
}
```

Listing 2.3. A basic asynchronous task declaration and usage in Embassy.

2.2 Networking and Communication

There are a few different methods of communication relevant to the scope of this project as the transfer of data is vital to the success of an IoT project.

2.2.1 Inter Integrated Communication (I2C)

I2C is a half-duplex simple and effective way to transmit data at high speeds (between 100kbit/s in Standard and up to 3.4Mbit/s in High-speed mode) and at a short distance. I2C works via a two-wire bus; the SDA and SCL wires. The SDA is the data delivery wire which transmits data from the slave device to the master device. The SCL is the clock wire, through which the master device delivers the clock signal to the slave device. In practice, pull-up resistors are also required for proper I2C communication. I2C can support multiple channels of communication, allowing a signal slave device to be controlled by multiple masters or vice-versa[15].

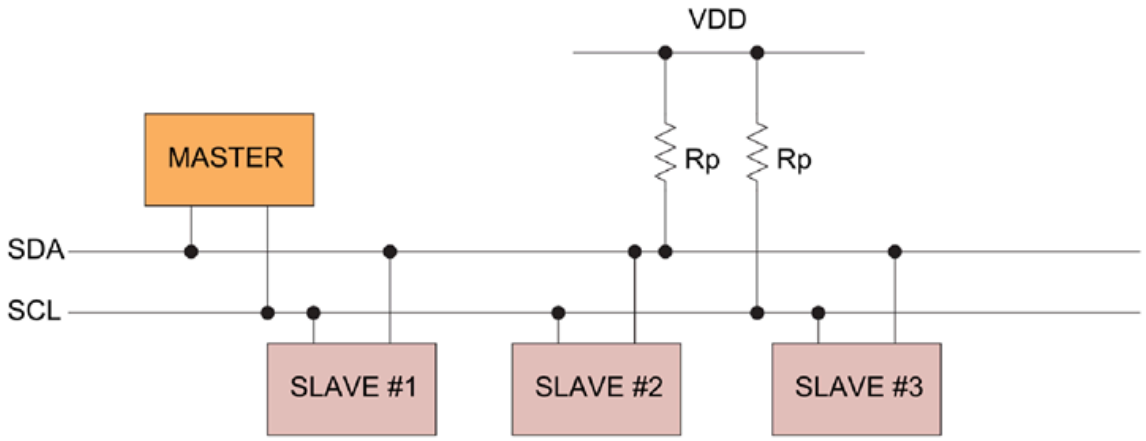


Fig 2.2. I2C diagram, where Rp are the pullup resistors[15].

When it comes to embedded Rust, I2C instances can be initialised using the esp-hal library. This instance can then be passed to client drivers such as the ones found in the ADS1x1x package[17]. Although it should be noted that in order to use I2C in an asynchronous manner, using the non-blocking io crate “nb” is required[18].

2.2.2 Transmission Control Protocol/Internet Protocol (TCP/IP)

The TCP/IP protocol is one of the most widespread protocols in use today. It ensures reliable end-to-end communication, including error checking and retransmissions. The IP

part of the TCP/IP protocol handles the addressing and routing. IPs can be either declared statically or assigned dynamically using a DHCP client.

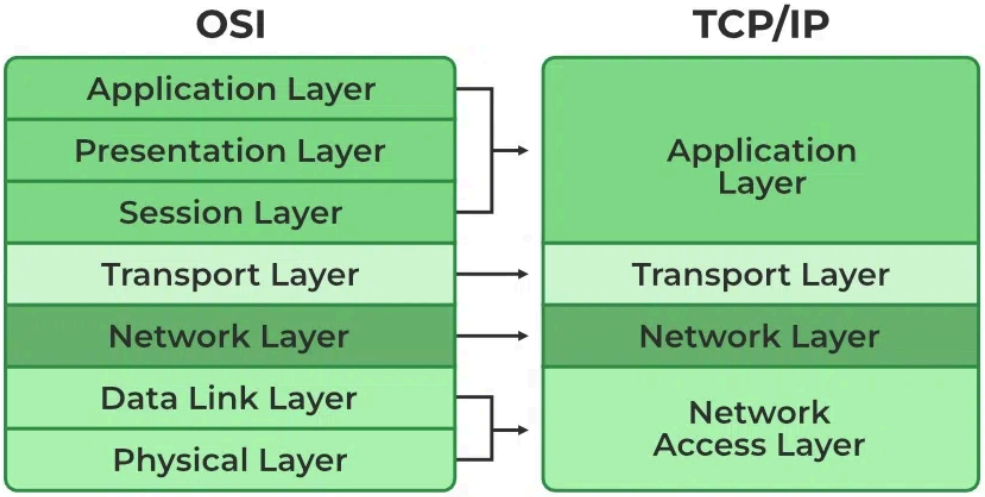


Fig 2.3. OSI vs TCP/IP protocol layers[19].

In order to achieve this reliability, TCP uses a three-way handshake. First, the client sends a SYN which aims to request a connection, then the server responds with a SYN+ACK which acknowledges and accepts the request, and finally the client responds with an ACK, acknowledging the server's SYN+ACK. At this stage, the connection is established and the data flow begins. The data flow is checked similarly with ACK if the response is received, and NACK if not[20].

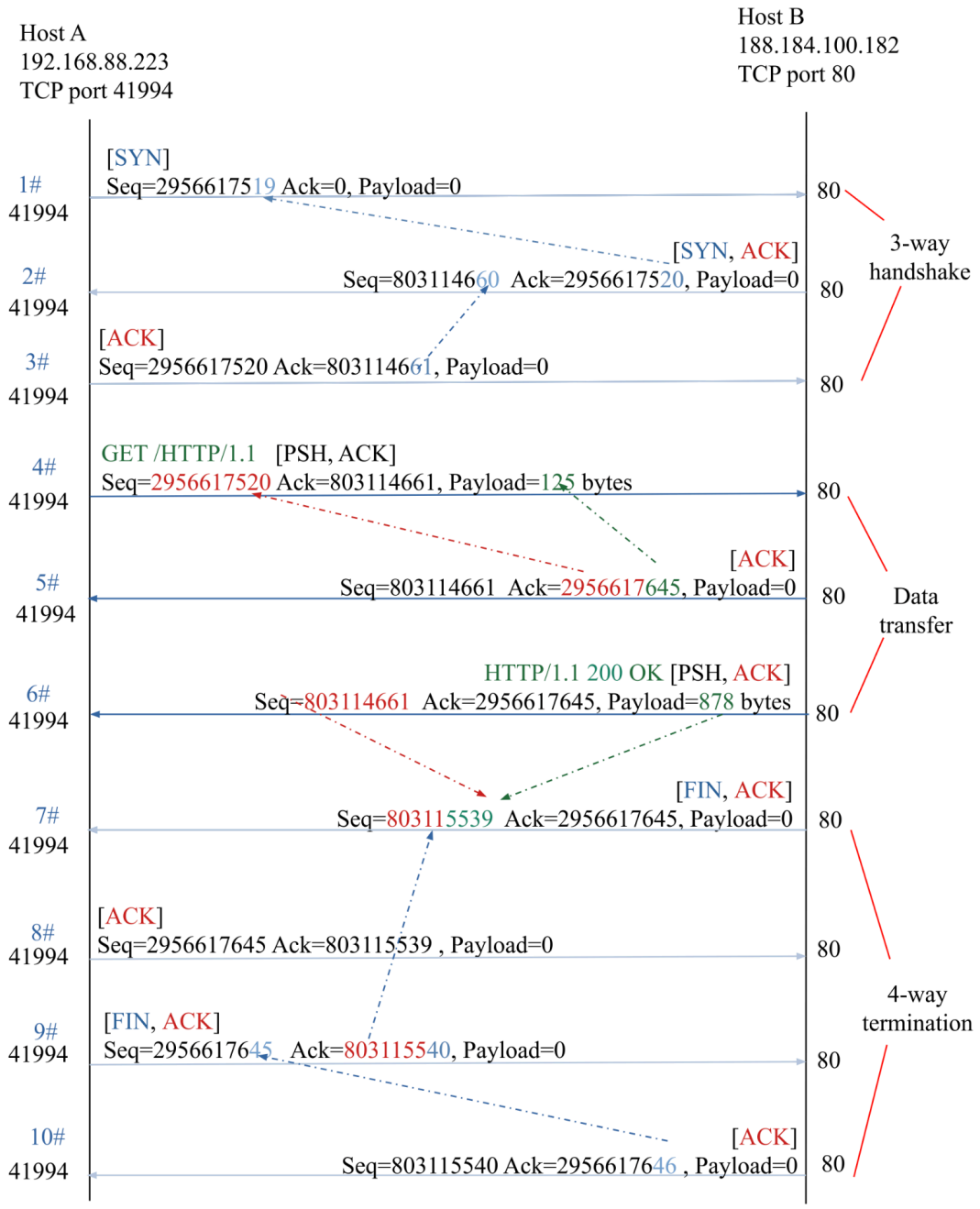


Fig 2.4. Flow control diagram demonstrating a typical TCP connection[20].

To handle the network stack for TCP/IP in embedded Rust, there exists the embassy-net package[21]. This package provides the network stack and a way to initialise and configure TCP sockets. Care is needed however to initialise buffers appropriately and with a sufficient size to ensure the TCP socket can be correctly accessed wherever it is used.

2.2.3 WiFi

WiFi is a common method of wireless communication in either the 2.4GHz or the 5GHz bands. WiFi usually includes a device in Access Point (AP) mode which provides access to the network and one or more devices in Station (STA) mode. WiFi allows for high flexibility along with a decent tolerance for physical distance between STA devices and the AP device. At increasing distances the latency and packet loss can increase, but this can often be alleviated by leveraging TCP's error checking and retransmission.

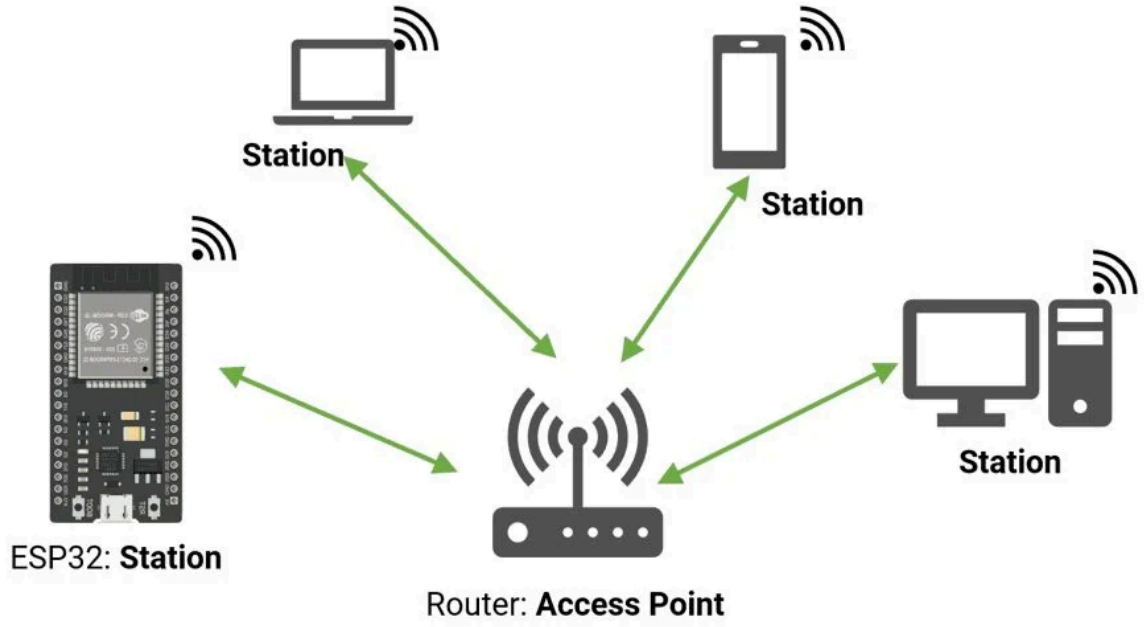


Fig 2.5. Station mode and Access point mode interaction[22].

The WiFi drivers can be initialised and configured in embedded Rust by first getting access to the peripherals via the esp-hal crate and then configuring the wifi controller as either an AP or STA mode mode. Then, depending on the mode, one can set the login credentials or the target credentials[23].

2.2.4 Hypertext Transfer Protocol (HTTP)

HTTP is the standard for communicating across the web. It is a stateless protocol, meaning that all the information for a given request is available in the request itself and no data is stored. A HTTP request consists of a method, a path, headers for metadata and a body. A HTTP response is similar, but instead of a method and path, there is a status code. For

example, an application might send a GET request to a server, and the server would respond with 200 OK if the command was successful[24].

```
GET /data HTTP/1.0
...
HTTP/1.0 200 OK\r\nContent-Type: application/json\r\n\r\n
```

Listing 2.4. A basic request and successful reply.

2.3 Data Acquisition and Processing

In order to accurately gauge the energy usage, the raw data must be gathered, processed and then printed in a readable and logical manner.

2.3.1 JavaScript Object Notation (JSON)

The goto method in many IoT projects for storing and transmitting data is JSON. JSON is easily readable by both humans and machines making it a good choice for debugging purposes. JSON is actually not dissimilar to TOML, which was discussed earlier.

```
{
  "sensor_name": "sensor1",
  "sensor_value": 100.5
}
```

Listing 2.5. Short JSON example.

In Rust, there exists the serde_json library. This allows the easy serialisation and deserialisation of Rust structs into JSON objects for transmission or reading provided the struct is defined[25].

2.3.2 Deriving the Instantaneous Power and Energy

In order to get the instantaneous power, one can use a voltage reading and an estimated load resistance:

$$p(t) = v(t)\, i(t)$$

Equation 2.1. Instantaneous power formula

Then, from:

$$v(t) = i(t)\,R$$

Equation 2.2. Ohm's Law

We can substitute to get:

$$P(t) = \frac{v(t)^2}{R}$$

Equation 2.3. Instantaneous power in terms of voltage and resistance.

We can then get the total energy consumption by integrating over the interval:

$$E = \int_{t_1}^{t_2} \frac{v(t)^2}{R}\,dt$$

Equation 2.4. Energy formula.

Or, discretely,

$$E = \sum_{n=1}^{N} \frac{v[n]^2}{R}\,\Delta t$$

Equation 2.5. Discrete energy version

The merit of this approach for monitoring the power and energy is discussed later, as it is flawed.

2.4 Graphical User Interface (GUI)

In order to display the information gathered by the clients, a GUI can be used.

2.4.1 Python and Qt Meta-object Language (QML)

Python is a high-level programming language with near limitless applications. There are many Python packages available for programming the backend and displaying graphs for the GUI.

PySide is a library that allows for use of the Qt ecosystem for building GUIs which is typically done in C++. It provides many modules that can be used for developing both the front-end and the back-end. The QtNetwork modules for instance are used to code network functionality into the GUI. PyQtGraph is a graphing package similar to matplotlib that can be used to graph data parsed in the backend seamlessly with the Qt ecosystem[26].

```
def sendRequest(self):
        url = QUrl("http://192.168.2.1:8080")
        request = QNetworkRequest(url)
        self.reply = self.nam.get(request)
```

Listing 2.6. A simple Qt Python function which sends a network request to the url.

QML is a declarative object language that allows for the description of physical elements on the GUI in relation to each other. The QML front end is initialised and bound to the Python back-end in the Python file. Interactive elements on the QML front-end can be coded in JavaScript which is a scripting language commonly used in web development for interactive elements.

```
function setText () {
    var i = Math.round(Math.random()*3)
    text.text = texts[i]
}

ColumnLayout {
anchors.fill: parent

Text {
    id: text
    text: "Hello world!"
    Layout.alignment: Qt.AlignHCenter
}

Button {
    text: "Click me"
    Layout.alignment: Qt.AlignHCenter
    onClicked: setText()
}
```

Listing 2.7. QML with an embedded JavaScript function[27].

Chapter 3 - Design

This chapter details the design choices and trade-offs involved in the development of the Rust-powered multi-channel energy monitor. The goal is to produce a full energy monitoring system which leverages Rust's unique features tailored for a smart home environment. It is to be able to monitor the energy consumption from multiple sources in real time, and then display the information in a human readable form.

In order to achieve this, there are many choices to be made. Such as:

- A suitable wireless architecture
- A system for capturing readings
- A microcontroller capable of:
    - Wireless communication
    - Using Rust with existing drivers
    - Handling the load of multiple concurrent tasks
- A lightweight runtime compatible with the microcontroller
- A suitable display method for information

The following section discusses each of these choices and more, showing some alternatives that were considered and the rationale behind the final choices.

3.1 System Architecture

This section focuses on the higher-level design of both the hardware and software architecture.

3.1.1 Hardware Architecture

The hardware for converting the analog signal into usable data for use on the network is shown below. The ADS1015 is the first hardware element to encounter the signal. This ADC is responsible for not only isolating the ESP32C3 from any potential power surges, but also acts as a slave device, digitising the analog signal.

The ADS1015 then communicates with the ESP32C3 via the I2C protocol, with the address set to ground as there is only a single master device. The I2C protocol requires pullup resistors across the SDA and SCL lines; two 10kΩ resistors connected to a 3.3V rail were chosen for this[4].

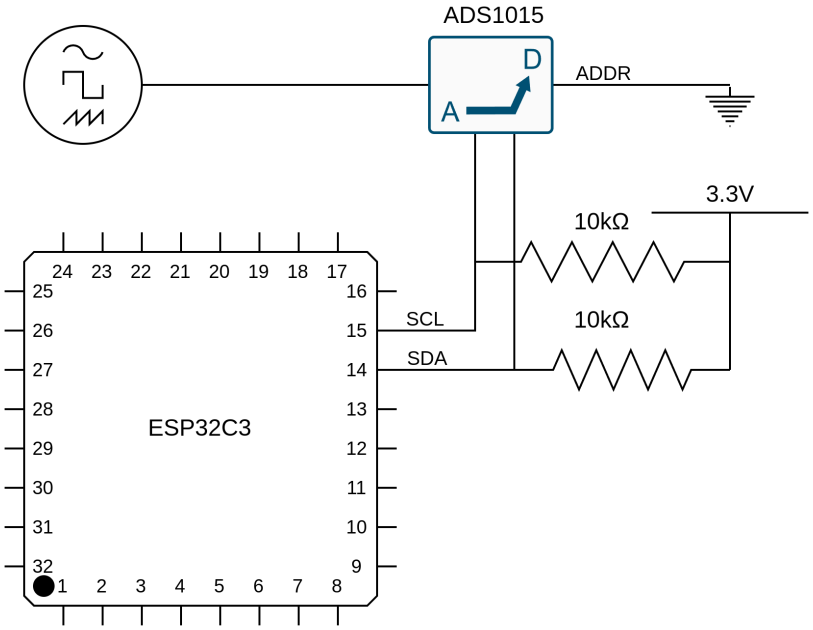


Fig 3.1. Layout of hardware configuration (ADC & ESP source and ground not shown)

On the ESP32C3, the General Purpose Input Output (GPIO) pins can be configured to communicate via I2C, with pins 13 and 14 corresponding to GPIO pins 8 and 9[1]. The ESP32C3 then handles the conversion into voltage. The ESP32C3, in this case, is the client device in the client-server model.

3.1.2 Software Architecture

As mentioned before, the chosen architecture is based on the client-server model. Multiple client ESP32C3s concurrently connect and communicate with the server ESP32C3. The server then writes the received data as JSON to the TCP socket which is available for access by the GUI.

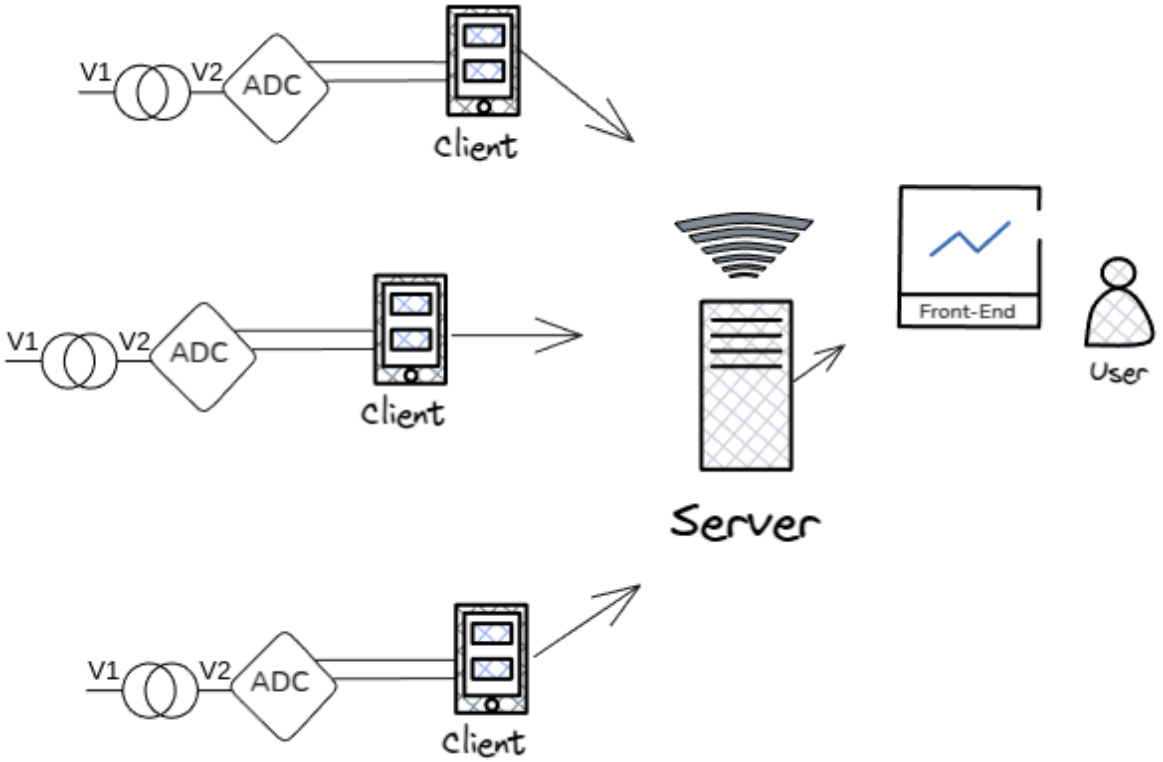


Fig 3.2. High level overview of the client-server model as shown in Chapter 1.

For the client software, Embassy's asynchronous framework was leveraged to support the asynchronous connection with the AP mode while running the stack and both reading and serialising the data to the socket. In order to circumvent socket ownership and lifetime issues, the data was read from the I2C pins and sent in the same asynchronous task.

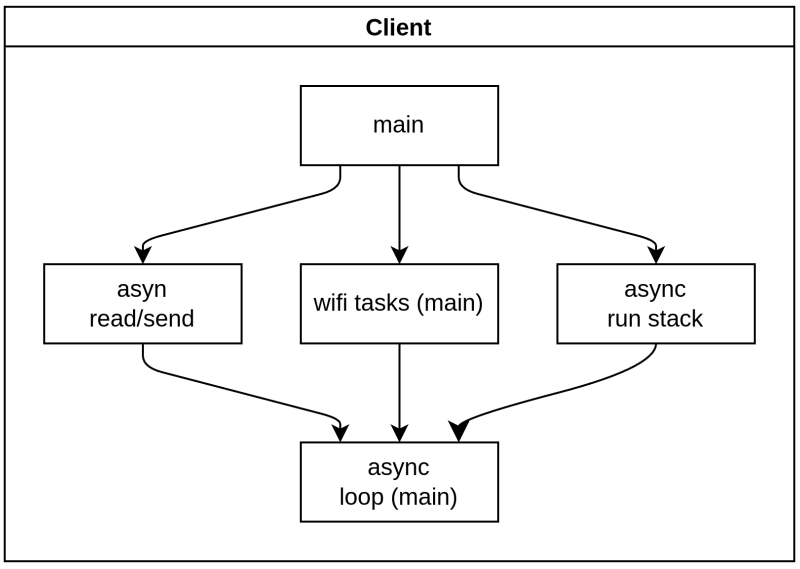


Fig 3.3. High level architecture of the client code

In main, the client first initialises the peripherals and sets configurations for the STA mode WiFi controller. Then, the main function handles the WiFi connection tasks synchronously to avoid race conditions with the stack and socket handler tasks, which are both asynchronous. After the WiFi is connected, the task responsible for running the network stack is spawned which runs asynchronously with main. Finally, the read/send task is spawned and given ownership over the key peripherals. This task builds the socket, reads the ADC and then serialises the data from the ADC onto the TCP socket as a JSON object.

The server is responsible for receiving the data from multiple clients simultaneously. For this, it also uses the Embassy asynchronous framework. The server uses multiple tasks to achieve its many responsibilities.

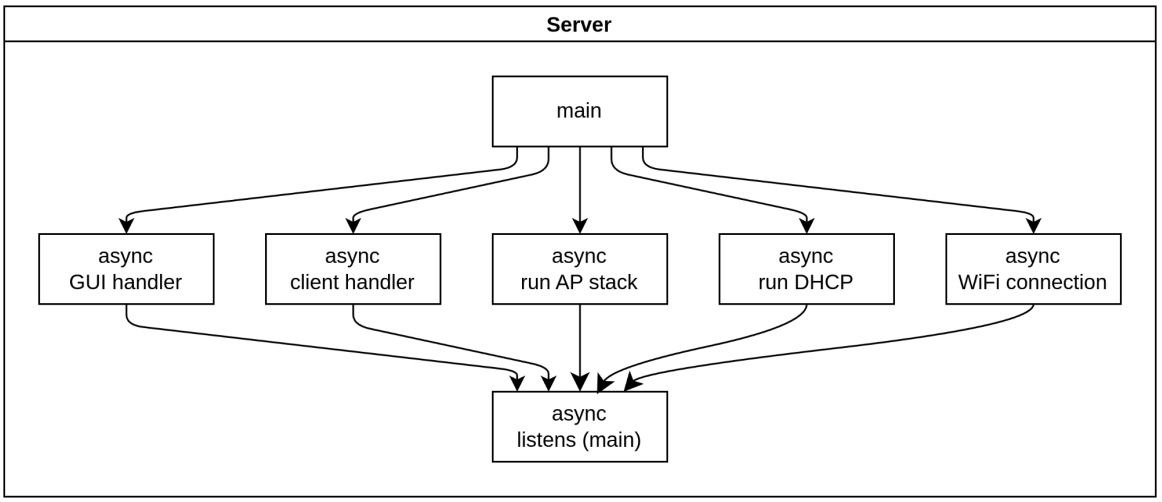


Fig 3.4. High level architecture of the server code

Similar to the client code, main handles the initialisation of the peripherals and the handles. From main, the above tasks are spawned. The connection task is spawned asynchronously (unlike the client) as it always needs to be available for new clients[23]. The stack task also runs asynchronously similarly to the client. The DHCP task is also initialised, but it is not used in practice as the IPs are statically declared for the clients for testing purposes. In main, a client socket is built as opposed to the socket being built from the stack in the client. This is because many new sockets need to be made - one for each client. The client handler receives ownership of the socket (whose handle is overwritten on the next loop iteration) for

receiving and serialising the data. The GUI handler is spawned using a socket created in main similarly to the client handler. The GUI writes the data received by the client handlers to a HTTP response, which can be requested by GUI.

The GUI is built using PySide, a way to use the Qt libraries with Python. The backend requests the data on the AP network at the designated port. The received JSON is then deserialised converted into a dictionary. These dictionaries are iterated through in a for loop in order to display the data using PyQtGraph for each client.

3.2 Design Choices

In this section, the rationale for choosing each element of the project is explained and compared with alternatives.

3.2.1 Rust Programming Language

The leading choice for programming in embedded systems tends to be C or C++[28]. Rust offers a similar level of low-level control as these languages but includes memory safety guarantees through compile time checking. This incurs an additional overhead and, in some cases, slower performance. Java is also considered a memory safe language but with significant overhead which may not make it appropriate for highly resource constrained embedded systems[29].

In terms of speed, C/C++ have overall comparable speed in algorithmic execution as Rust, with Rust being slightly slower whereas microPython can be seen as being significantly slower when tested by the same methods[30]. When Rust is tested against Java, there is also evidence that Rust is significantly faster in the methods tested[7]. For this reason, Rust is certainly worth considering in areas where speed is important.

When it comes to file size, it can be seen that Rust files tend to be quite a bit larger than those of C/C++ (~79% larger) in no_std context. The paper covering these findings quotes:

- Deeply ingrained monomorphization,
- Suboptimal compiler generated support code,

- Hidden data structures and data,
- Fewer compiler optimizations

as causes of this bloating. However, by following best practices, one can mitigate the binary growth during development[31].

However, Rust really shines in its memory safety. While languages like Java are memory safe, they incur an enormous overhead in terms of execution time from runtime garbage collection and the Java Virtual Machine (JVM) in the case of Java. Rust largely avoids this by using a borrow checker at compile time to guarantee lifetimes. This avoids the mistakes with pointers one might make in C/C++ which, as mentioned in chapter two, are responsible for the vast majority of severe security bugs[9]. In IoT, robustness and safety are very important as people's privacy could be at risk. For these reasons, Rust is a preferable choice for a smart home energy monitor.

3.2.2 Wireless Protocol

Since this is a wireless energy monitor, the choice of protocol is very important and serves as the basis for many more decisions down the road - such as the architecture of the system and microcontroller.

WiFi and bluetooth can be directly compared due to their similar architecture. In WiFi the AP mode and STA mode devices are mirrored by the Bluetooths central node (such as a mobile phone) and its peripheral devices (such as wireless earbuds). Bluetooth specialises in short range communication, with a slower speed and lower power consumption. The lower power is attractive from a power expenditure standpoint, but the range is too short to facilitate reliable communication across a smart home. WiFi on the other hand, is well suited to these ranges making it the better choice[32].

Zigbee uses a different architecture to WiFi and Bluetooth: a mesh architecture. This allows Zigbee devices to reach a longer range than WiFi at the cost of hop latency. Zigbee also tends to use less power making it ideal for sensor devices. On the other hand, Zigbee doesn’t have the same level of documentation and support as those for WiFi and Bluetooth. The management of the mesh’s network topology could also involve more complicated management of the network[33]. These factors, combined with the developers lack of

experience with the protocol, could lead to longer development times. For this reason, WiFi was the more practical choice.

3.2.3 Data Protocol

The two main choices in transmission protocol are TCP or User Datagram Protocol (UDP). Both have their benefits, with UDP boasting a lower overhead and faster communication, while TCP is more focussed on reliable data delivery with robust error correction as discussed in Chapter 2. Considering that in a smart home, there isn't a clear line of sight for communication due to walls and obstacles and the range can be significant, the signal could potentially get quite weak. There is also the possibility of interference from other wireless devices on the same frequency such as the home's WiFi connection (if using 2.4GHz rather than 5GHz) which could lead to packet loss or unstable connections. These conditions could make it so there is an obscuring of unusual behaviour, which an energy monitor could be able to detect.

Another benefit of TCP, is the reusability of socket code. HTTP is built on TCP, meaning one wouldn't need to manage two separate socket types when communicating with the GUI. With these factors in mind, TCP sockets were chosen as the go-to protocol for data transmission.

3.2.4 std or no_std? That is the question.

The choice between std and no_std can have a significant impact on both the development of the project, the tools available and the overall binary size. While using std can make for simpler development through the use of common Rust libraries and added stack overflow protection, when in a resource constrained environment the added overhead of importing std cannot be ignored. Not importing std can also lead to less latency. Another edge of no_std is the increased amount of control over hardware and concurrency, allowing for less unexpected behaviour[34]. Considering that Rust already has a generally larger binary size than equivalent C/C++ code, using no_std is likely the better choice in this regard[31].

3.2.5 Operating System/Environment

There exists a few solid options for programming in Rust in a no_std environment such as Embassy and RTIC. There are other options available, but these are not necessarily Rust native meaning that there is less support for compiling in Rust (although it can be done). RTIC offers only a Real Time Operating System (RTOS) while Embassy offers both an

asynchronous runtime and HALs[35]. While both are open source, Embassy offers packages such as embassy-net which integrate well with the environment whereas RTIC requires the user to provide HAL implementations and networking. Considering networking with WiFi is a core part of the project, Embassy was chosen.

3.2.6 Microcontroller

The choice of microcontroller is an important one, as it dictates the limitations of the software. Given the runtime environment is Embassy, one must pick from the options which are known to be compatible. The compatible devices fall under four categories[36]:

- nRF kits,
- STM32 kits,
- RP2040 kits,
- ESP32 kits (specifically, the ESP32C3).

Any of these options are fine, provided they have WiFi capabilities. The ESP32C3 was chosen because it is both cheap and uses the open source RISC-V architecture and therefore toolchain[1]. The developer had experience with this toolchain which could improve debugging speed (for example, interpreting the MCAUSE register values to determine fault source).

3.2.7 Bus Protocol

When it comes to bus protocols, there is no shortage of choices available. For this project, the ADC to client only requires a half-duplex protocol as there is no reason for the client to communicate back to the ADC. For this reason, Serial Peripheral Interface (SPI) is likely overkill as it requires more pins to deliver full-duplex communication. There is also the option of having multiple sensors on the same client via the same I2C interface. This could allow for an easy collection of both voltage and current simultaneously without having to use separate SDA/SCL buses[15].

3.2.8 Analog to Digital Converter

The use of an external ADC can prove very beneficial to the long-term operation of the client device. The extra layer of the ADC isolates the client from any unexpected power surges at the cost of an increase in energy expenditure.

There are many cheap options for ADCs which are suitable for the use-case while also using I2C. An important element to look out for is whether or not Rust packages exist for the

ADC of choice. The ADS1x1x series of ADCs by Adafruit satisfy the requirements listed above, offering various levels of accuracy and sampling rates. There is a tradeoff with resolution and sampling rates, with higher resolution comes lower sampling rates[17].

Considering the frequency of the signals from main are generally only 50Hz, the sampling rate isn't too important provided it's sufficient for capturing the waveform - for instance if one wanted to use cubic spline interpolation to recreate the signal one would only need 4 data points which, at 50Hz, only requires >200 samples per second. For resolution, a 12-bit ADC would net 4,096 discrete levels whereas a 16-bit ADC would net 65,536. Energy monitoring is not necessarily an extremely precise application, so having a 16-bit result would likely just waste space without providing much insight. With this in mind, the ADS1015 was chosen. The 12-bit resolution provides more than enough accuracy without wasting space and the maximum sampling rate of 3.3kHz ensures that there is no bottleneck. The sampling rate could even be decreased to reduce power consumption by limiting the duty cycle.

3.2.9 Graphical User Interface

Displaying the data in a human readable fashion was an important final step in the pipeline of the data. For this part, there were three main options: a web application, a mobile application or a desktop application. The mobile application idea was discarded as the deployability would be limited - having to rely on an app store for deployment or having to bypass OS protections. A web app was another potential option with the benefit of not having to install the app but at the cost of having to run JavaScript on the server ESP which could end up having a significant impact on the performance of the server device. A desktop app was therefore the most practical choice. This allows any graphical processing or data analysis to be offloaded to the user's machine.

In terms of building the desktop app, there is a range of potential frameworks that exist that satisfy the minimum requirements of having graphing capabilities. A common choice is using Qt framework through PySide, which provides an easy way to build a desktop app with graphing capabilities. It has its own IDE called Qt Creator which allows for easy management of the Python environment and the QML front-end. This framework generally has more options than that of something like PySimpleGUI and can make an aesthetically pleasing but functional interface[37].

Chapter 4- Implementation and Testing

This chapter details the implementation of the rust-powered multi-channel energy monitor. The full source code is available in the GitHub repository.

4.1 Development Environment

In this section, the development environment and configuration used for building the Rust-powered energy monitor is discussed.

4.1.1 Linux and VSCode

The operating system of choice for developing this project was the Arch distribution of Linux. Linux is famous for its configurability and open source ecosystem, providing an easy and quick way to identify and download the necessary compilers and utilities via “pacman” or via the Arch User Repository (AUR). An example of an easy way using Linux improved workflow is with the following addition to the .bashrc file:

```
alias fyp='cd Projects/FinalYearProject/rustmonitor'
```

Listing 4.1. An alias used for quick access to the project’s directory.

The chosen IDE for development was VSCode. VSCode offers thousands of extensions and is highly configurable. Access to the Rust LSP is given via the rust-analyzer extension which is a must for Rust development. It provides a simple way to view key packages and understand how to use the code therein. Another handy extension is the Even Better TOML which provides links to packages present in the TOML file easily.

4.1.2 Cargo and GitHub

The function of Cargo is discussed at length in Chapter 2. In order to set up cargo, one needs to simply install Cargo and Rust via the command line. It should be noted that the nightly version is recommended for development as it is the most up-to-date version

```
sudo pacman -S <Official-pkg-here> && yay -S <AUR-pkg-here>
```

Listing 4.2. Commands for installation

Cargo can then be used for various features related to the development. Some key commands that were used often are[38]:

`cargo install <cargo-sub-command>`

- This installs cargo subcommands such as cargo-generate and espflash utilities.

`cargo generate esp-rs/esp-template`

- This useful command generates the Rust project structure, with the template option available for either an std or no_std environment.

`cargo check`

- This command simply checks the dependencies for errors.

`cargo espflash flash -monitor -release`

- This flashes the optimised version to the target listed in the .cargo/config.toml file

`cargo run`

- This runs the program in the current directory to avoid flashing again.

`cargo add <package-name>`

- This adds a target package to the Cargo.toml file. Use of this command was avoided, as without careful manual management of features, errors can arise in a no_std environment.

Another core component to the development environment was Git. The version control software can keep track of changes and avoid loss of data in case the laptop was lost or destroyed. After following the instructions on GitHub for a new project, the following commands were used on every working iteration of the project:

```
git add .
git commit -m "<changes made>"
git push
```

Listing 4.3. Commands that add, commit and then push current code to main.

4.2 Client-Side Software Implementation

In this section, the details of the client side software implementation are covered. This includes both the real and fake data.

4.2.1 Initialisation

The initialisation of all of the hardware peripherals was done in main. This allows the code to interface with the hardware of the ESP32C3. The config is set to the default configuration at max clock speed which is adequate for the use-case.

```
let config= esp_hal::Config::default().with_cpu_clock(CpuClock::max());
let peripherals = esp_hal::init(config);
```

Listing 4.4. Getting handles

With this “peripherals” handle, one can now access the individual hardware peripherals by declaring a new handle addressing the correct field in the DEVICE_PERIPHERALS struct for the ESP32C3.With this “peripherals” handle, one can now access the individual hardware peripherals by declaring a new handle addressing the correct field in the DEVICE_PERIPHERALS struct for the ESP32C3.

```
let timg0 = TimerGroup::new(peripherals.TIMG0); //first timer for
//peripheral stuff
let timg1 = TimerGroup::new(peripherals.TIMG1); //embassy timer for
//async stuff
esp_hal_embassy::init(timg1.timer0);
let mut rng = Rng::new(peripherals.RNG); // random number generator
let per_pins = peripherals.I2C0; //handle for peripheral pins
let wifi = peripherals.WIFI; //handle for wifi
```

Listing 4.5. Peripheral initialisation[23]

Once the peripherals are initialised, a heap needs to be allocated manually for any dynamic tasks that may need allocation in the background such as network buffer allocations. Usually this is provided in an std environment.

```
esp_alloc::heap_allocator!(72 * 1024); //heap to 72kbytes
```

Listing 4.6. Heap allocation[23]

There is also a macro declared which is commonly used in WiFi implementations. In the std crate, an equivalent macro exists called make_static!, but since this is a no_std environment, one is declared for use in making things that need to be alive for the duration of the program in all contexts.

```
macro_rules! mk_static {
   ($t:ty,$val:expr) => {{
       static STATIC_CELL: static_cell::StaticCell<$t> =
          static_cell::StaticCell::new();
       #[deny(unused_attributes)]
       let x = STATIC_CELL.uninit().write(($val));
       x
   }};
}
```

Listing 4.7. The mk_static! macro boilerplate.[23]

4.2.2 I2C Implementation and Random Data Generation

For the testing of the full system, only one device will receive real I2C readings from the ADC while the others will generate random data using the Rng peripheral. Firstly, for the real I2C data, the per_pin handle which was taken from the peripherals is passed to the read_send_current task alongside the sta_stack and a copy of the rng hande (unused if data is real):

```
spawner.spawn(read_send_current(sta_stack, per_pins,
generate_data_rng)).ok();
```

```
async fn read_send_current (sta_stack: Stack<'static>, peripherals:
I2C0, mut rng: Rng)
```

Listing 4.8. Task spawner for reading and sending current and the tasks signature.

Inside the read_send_current task, a helper function is declared which uses the received rng peripheral to generate a random unsigned 32-bit integer and converts it to a voltage using the correct scale (4096mV). The reciprocal was precalculated rather than recalculated on each iteration in order to save resources.

```
const PRECALCED_RECIP: f32 = 9.53674317e-7; //1 / (2^32 -1)*(4096)


fn generate_data(rng: &mut Rng) -> f32 {
        let mut x = rng.random() as f32; //gets random number
        x = PRECALCED_RECIP*x;
        return x;
    }
```

Listing 4.9. Data generated and converted to voltage.

Then, for the real data, the SDA and SCL pins are defined. This is done by bypassing the ownership model of Rust using an unsafe scope. The pins are "stolen" by simply assigning their handle to the handle contained in the GpioPin<pin_num> struct, with the generic argument as the pin which is stolen.

```
let sda: gpio::GpioPin<8> = unsafe {gpio::GpioPin::steal()};
let scl: gpio::GpioPin<9> = unsafe {gpio::GpioPin::steal()};
```

Listing 4.10. Declaration of the SDA and SCL pins.

These pins are used to define the I2C driver. This driver can then be passed as an argument to the specific driver of the ADC. The crate ads1x1x provides drivers for the Adafruit ADCs in the family of the ones expanded on in Chapter 3. The ADS1015 can be selected using the new_ads1015 constructor with both the correct target address and native I2C driver[17]. The driver can then be configured to have the required data rate and the voltage scale.

```
let my_i2c = I2c::new(peripherals, Config::default())
    .unwrap().with_sda(sda).with_scl(scl);
let mut adc_i2c = Ads1x1x::new_ads1015(my_i2c, TargetAddr::Gnd);
adc_i2c.set_data_rate(ads1x1x::DataRate12Bit::Sps250).unwrap();
adc_i2c.set_full_scale_range(FullScaleRange::Within4_096V).unwrap();
```

Listing 4.11. Driver declaration and configuration[39].

Finally, the I2C values can be read directly from the adc_i2c handle. It should be noted that this is a non-blocking operation. Given the clock speed is much greater than the sampling rate, this could cause the operation to return an error: WouldBlock. Thankfully, there is the nb (non-blocking) crate which provides a macro that will continuously poll the operation until it no longer returns the WouldBlock error, allowing the data to be read[17]. The raw data is then normalised.

```
let raw_data = nb::block!(adc_i2c.read(SingleA0)).unwrap();
let voltage = (raw_data as f32/32767.0)*4096.0;
```

Listing 4.12. The reading and conversion of I2C data.

4.2.3 Networking

Since the WiFi peripheral is declared, it can now be used in constructing the AP/STA mode interface. While the wifi_ap_interface is a required argument, it is never used or initialised in the stack. The initialisation is declared using the mk_static macro, which makes the init persistent using a static cell. It requires the previously declared timer, a rng for any random numbers used in the network stack for security and the dedicated clock for the WiFi circuitry.

```
let init = &*mk_static!(
       EspWifiController<'static>,
       init(timg0.timer0, rng.clone(), peripherals.RADIO_CLK).unwrap()
   );
```

```
let (_wifi_ap_interface, wifi_sta_interface, mut wifi) =
       esp_wifi::wifi::new_ap_sta(&init, wifi).unwrap();
```

Listing 4.13. Formation of the initialisation argument and construction of the interface[23]

Now that the interface has been declared, one can now set the configuration of the WiFi. The password and SSID are stored in heapless (preallocated) strings, with the DNS stored as a vector. The STA configuration is created as a static IPv4 type with the IP address declared manually. This is because the server is set to parse the data based on the client IP address.

```
let mut ssid = String::<32>::new();
ssid.push_str("esp-wifi").unwrap();
let mut password = String::<64>::new();
password.push_str("").unwrap();
let mut dns_servers = Vec::<Ipv4Address, 3>::new();
dns_servers.push(Ipv4Address::new(8, 8, 8, 8)).unwrap();

let sta_config = embassy_net::Config::ipv4_static(StaticConfigV4 {
   address: Ipv4Cidr::new(Ipv4Address::new(192, 168, 2, 20), 24),
   gateway: Some(Ipv4Address::new(192, 168, 2, 1)),
   dns_servers,
   });
```

Listing 4.14. Setting the password, SSID and IP configuration[23]

These variables can then be passed into the wifi peripheral configuration. Once set, the wifi client can be started with this configuration. This is not done asynchronously as it would conflict with the socket formation and network tasks which rely on the link being established (a better approach may have been to utilise the stack.is_link_up approach seen on the server side).

```
wifi.start().unwrap();


while !wifi.is_connected().unwrap() {
       wifi.connect().unwrap();
       let config = wifi.configuration().unwrap();
       esp_println::println!("Waiting for station {:?}", config);
       Timer::after(Duration::from_millis(5000)).await;
}
```

Listing 4.15. The starting and connecting of the WiFi client[23].

This allows the client device to connect to the SSID provided, but there is no communication taking place. This is what the sta_stack and sta_runner are for. The stack is allowed 3 “stack resources” as space for the network tasks. This is fine since there is only a single socket being used by the client, but if more were required, the stack resource allowance would need to be increased. The sta_task simply runs the network stack asynchronously as shown below:

```
let (sta_stack, sta_runner) = embassy_net::new(
        wifi_sta_interface,
        sta_config,
        mk_static!(StackResources<3>, StackResources::<3>::new()),
        seed,
    );

spawner.spawn(sta_task(sta_runner)).ok();

async fn sta_task(mut runner: Runner<'static, WifiDevice<'static,
WifiStaDevice>>) {
   runner.run().await //runs network stack
}
```

Listing 4.16. Definition of the sta_stack and runner and spawning of the task[23].

Now that the underlying functionality for the WiFi is configured, the sockets can be formed and used. This is done in the read_send_current task as if the socket is created in main, the ownership issues would force the use of static mutables, which can be avoided in this case. The socket requires two buffers, one for receiving and one for transmission. The standard for use is 1536 bytes as it allows for the ethernet maximum transmission unit of 1500 bytes[40] plus a small overhead to hit a multiple of 64 bytes as is standard.

```
let mut sta_rx_buffer: [u8; _] = [0; 1536];
let mut sta_tx_buffer: [u8; _] = [0; 1536];
```

Listing 4.17. Buffer definitions[23].

These buffers are then passed, along with the sta_stack, into the constructor for a TcpSocket with a timeout configured.

```
let mut sta_socket = TcpSocket::new(sta_stack, &mut sta_rx_buffer,
&mut sta_tx_buffer);
sta_socket.set_timeout(Some(embassy_time::Duration::from_secs(10)));
```

Listing 4.18. TCP socket construction[23].

The goal is now to get the socket connected to the correct server port. This can be done by describing the server target as an IpEndpoint. This endpoint contains all relevant information about the server and can be used as an argument to the socket connect function to explicitly define the port.

```
let server_ip = Ipv4Address::new(192, 168, 2, 1);
let server_port = 5050;
let server_endpoint = IpEndpoint::new(server_ip.into(), server_port);
```

Listing 4.19. Server endpoint declaration[41].

The client socket can now repeatedly attempt to connect to the target port on the server until a Result::Ok() is returned by the connect function - meaning the client is successfully connected to the server.

```
loop {
        match sta_socket.connect(server_endpoint).await {
            Ok(_) => {
                println!("Connected to server at port {}",
    server_endpoint.port);
                break;
            }
            Err(e) => {
                println!("Connection failed: {:?}", e);
                Timer::after(Duration::from_secs(1)).await;           }
        }
    }
```

Listing 4.20. Retry loop[23]

4.2.4 Serialisation of Data

To allow for the data (real or fake) to be transmitted by client over the TCP socket, a serialisation scheme must be used. As discussed in Chapter 2 and Chapter 3, JSON is the format with which the data is transported through the system. This starts in the client, with the most simple version of the sensor data struct. It contains a field which identifies the client device and one which contains the voltage data after conversion. In order to serialise this data, the serde_json_core package was used. This provides utilities to not only convert the struct into JSON, but also serialise said JSON for transmission (as a string)[25].

```
#[derive(serde::Serialize)]
struct SensorData {
   sensor_id: u8,
   sensor_value: f32,
}
```

Listing 4.21. Serialisable SensorData struct.

A new SensorData struct is declared on each iteration of the loop, overriding the previous version, in which the current value of the voltage and the sensor ID is stored. This is then converted to a JSON string using serde_json_core, appended with a termination sequence, and then written to the socket. The "\r\n\r\n" is commonly used in HTTP separations and is easy to parse for.

```
let data = SensorData { sensor_id: 20, sensor_value: voltage};
let mut sendable:String<128>  = serde_json_core::to_string(&data).unwrap();
sendable.push_str("\r\n\r\n").unwrap();
sta_socket.write_all(sendable.as_bytes()).await.ok();
```

Listing 4.22. Sending of the data[23]

4.2.5 Dependencies

The dependencies required to make this work are stored in the Cargo.toml file and then defined for use at the top of the file. For example, the embassy-net package was imported and contains a lot of the underlying framework behind the TCP socket connections and stack:

```
use embassy_net::{
   tcp::TcpSocket, IpEndpoint, Ipv4Cidr, Runner, Stack, StackResources,
   StaticConfigV4
};
```

and the corresponding Cargo.toml entry:

```
embassy-net = { version = "0.6.0", features = [ "tcp", "udp", "dhcpv4",
"medium-ethernet"] }
```

Listing 4.23. Example of tying in the Cargo.toml entries with source code.

One of the key changes made in the Cargo.toml file was the manual allocation of the task arena size. This was done because the default allocation was insufficient to handle the amount of work done in the tasks. The correct flag was determined through trial and error. The tradeoff was between allowing enough for the tasks to operate, without overloading the available SRAM of the ESP32C3.

```
embassy-executor = { version = "0.7.0", features =
["task-arena-size-163840"] }
```

Listing 4.24. Task arena size final value.

4.3 Client-Side Hardware Implementation

This section shows the exact hardware configuration and how the final setup was achieved. The setup is quite simple, only requiring a breadboard, two 10kΩ resistors, some wires, the ADS1015 and, of course, the ESP32C3. The GPIO pins 8 and 9 correspond to pins 14 and 15 on the devkit respectively[4], [1].

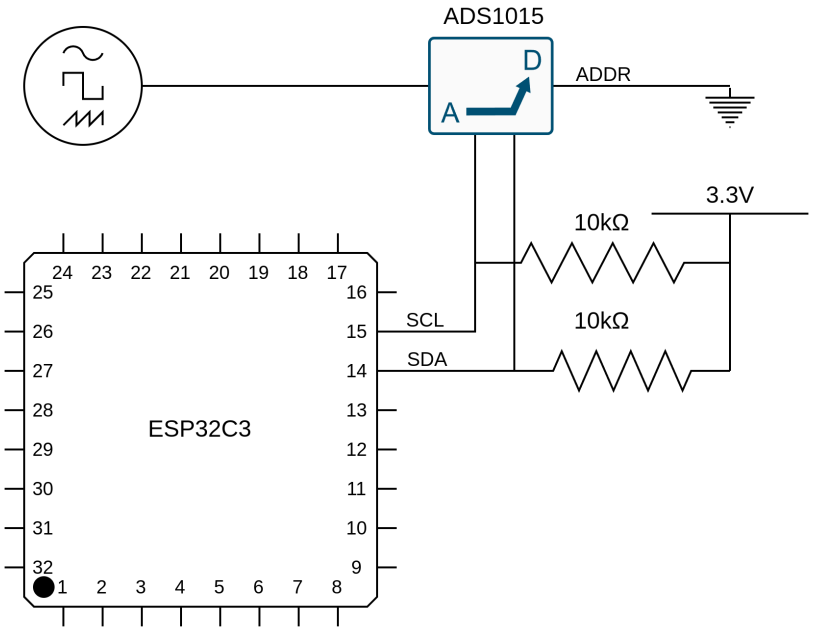


Figure 4.1. The circuit diagram for the hardware as detailed in Chapter 3.

The breakout pins were soldered to the ADS1015, which was then wired to the voltage and ground rails. The SDA and SCL buses were then wired to the ESP32C3. For testing purposes, the ADALM2000 delivered the 3.3V source, and the 5V power supply for the ESP32C3 was delivered via a USB cable.

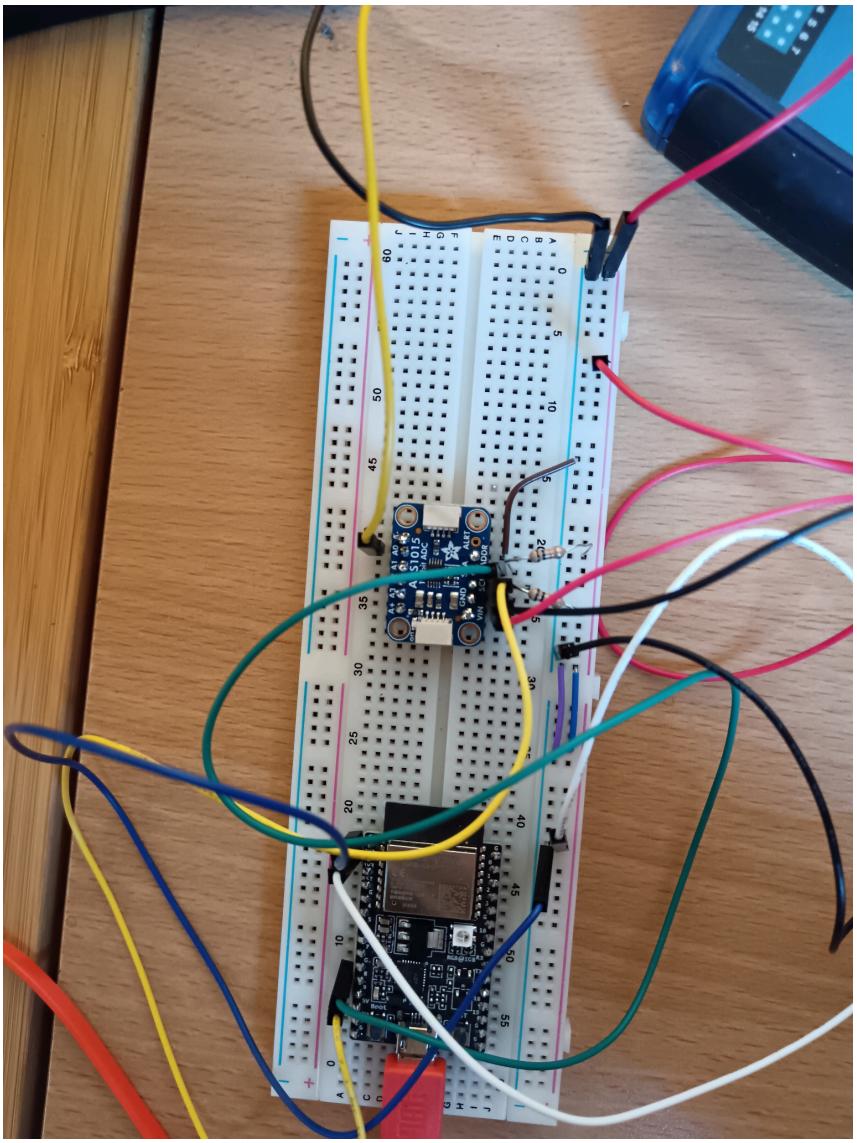


Figure 4.2. Physical layout of the hardware

The exact testing setup, including theADALM200 setup and configuration, will be shown in the testing section later in the chapter.

4.4 Server-Side Software Implementation

This section shows the exact server implementation. The peripheral initialisation is largely the same as on the client-side.

4.4.1 Networking

Considering that the same packages are used in both cases for networking, a lot of the code is similar to the client-side. The key difference between the both, is that the configuration is for an AP mode device.

In the server side, the connection task is asynchronous spawned, as it must be maintained even after a connection is found. This is not needed on the client-side as there only needs to be a single connection. The following is the only line needed in main before the connection task is spawned. Note that WifiApDevice is a struct from the esp-wifi crate, not declared in main.

```
let (wifi_interface, controller) =
       esp_wifi::wifi::new_with_mode(&init, wifi, WifiApDevice).unwrap();
```

Listing 4.25. The declaration of the WiFi controller needed for the following task[42].

In the connection task, the first section of the loop checks for whether or not the AP is up and running. It simply waits for the stop event - indicating the controller is stopping, so that it can delay a further five seconds to avoid any conflicts and allow for a graceful shutdown.

```
match esp_wifi::wifi::wifi_state() {
           WifiState::ApStarted => {
               controller.wait_for_event(WifiEvent::ApStop).await;
               Timer::after(Duration::from_millis(5000)).await
           }
           _ => {}
       }
```

Listing 4.26. Allow for graceful shutdown in case of a stop event[42].

The next section of the loop sets up the access point configuration. The majority of the access point configuration options can be set to default, as the only relevant input to the configuration is the SSID - which is set to “esp-wifi”. The controller is then started up using the async version of the start command. This is because, unlike the client-side code, there is a loop in main waiting on the stack before continuing to spawn the clients.

```
if !matches!(controller.is_started(), Ok(true)) {
           let client_config =
```

```
            Configuration::AccessPoint(AccessPointConfiguration {
                ssid: "esp-wifi".try_into().unwrap(),
                ..Default::default()
            });
            controller.set_configuration(&client_config).unwrap();
            println!("Starting wifi");
            controller.start_async().await.unwrap();
            println!("Wifi started!");
        }
```

Listing 4.27. WiFi connect code[42].

```
loop {
        if stack.is_link_up() {
            break;
        }
        Timer::after(Duration::from_millis(500)).await;
    }
```

Listing 4.28. Loop that waits until the stack is ready before spawning sockets[42].

For the network stack, the configuration is not dissimilar to the one used in the client side. One of the more impactful configuration options which is available, is the stack resources allocated to the network stack. Here, the StackResources provided to the stack and runner is set to 7. This is a clear limiting factor for the number of sockets available - 3 for a single client, and an extra one for each socket added.

```
let (stack, runner) = embassy_net::new(
        wifi_interface,
        config,
        mk_static!(StackResources<7>, StackResources::<7>::new()),
        seed,
    );
```

Listing 4.29. Different resource allocation than with client-side[42].

The scheme for using sockets is different for both the client_handler and the gui_handler. The gui_handler has a single socket. Unlike on the client side, the socket is spawned in main rather than the gui_handler task. This is done because the “stack” (network stack) must be available for both the client and the server. If it was done like the client and the gui_handler received ownership of the stack, it would not be available for creating the new sockets (in

hindsight, the Stack<'_> type implements the copy trait, so this may not have been necessary). The gui_handler was then spawned with this socket as an argument.

```
static mut RX_GUI_BUFFER: [u8; 1536] = [0u8;1536];
static mut TX_GUI_BUFFER: [u8; 1536] = [0u8;1536];


let mut socket = TcpSocket::new(stack, &mut RX_GUI_BUFFER, &mut
TX_GUI_BUFFER);
socket.set_timeout(Some(embassy_time::Duration::from_secs(10)));

...

spawner.spawn(gui_handler(socket)).ok();
```

Listing 4.30. GUI socket initialisation using static mutable buffers (requires unsafe{})

Note: The use of static muts would normally be a larger issue, but if only one piece of code is responsible for the writing, there is a much lower chance of any race conditions taking place.

In the gui_handler, the socket was then asynchronously waited on. The result of socket.accept is awaited until either an error is thrown, or the GUI successfully connects. The local endpoint is defined at port 8080, which will be the target for the GUI. The handling of the GUI will be discussed in a later dedicated subsection.

```
let r = socket
            .accept(IpListenEndpoint {
                addr: None,
                port: 8080,
            })
            .await;
```

Listing 4.31. The socket accept code, with the local endpoint defined.

For the client_handler, in order to make it a “multi-channel” sensor, there is functionality to spawn a new client_handler instance on each socket connection. Like in the case of the gui_handler, the sockets are spawned in main. Instead of it being a simple one and done situation, there is a loop which constantly builds new sockets. When a socket is connected to, its ownership is transferred to the handler instance, and it is overwritten by a new socket, with its own dedicated buffer. Firstly, to allow for multiple instances to spawn, it is necessary to define the following in the task flag:

```
#[embassy_executor::task(pool_size = MAX_CLIENTS)]
```

In order for each socket to have its own buffer, a pool of static mutable buffers had to be defined. This would allow the simple selection of a buffer based on the current number of active clients. To index this pool, an atomic was chosen[43]. This would allow the current count to be global and safe yet avoids using another static mut.

```
static CLIENT_COUNT: AtomicU32 = AtomicU32::new(0);
static mut RX_CLIENT_BUFFER_POOL: [[u8; 1536]; MAX_CLIENTS] = [[0u8; 1536];
MAX_CLIENTS];
static mut TX_CLIENT_BUFFER_POOL: [[u8; 1536]; MAX_CLIENTS] = [[0u8; 1536];
MAX_CLIENTS];
```

Listing 4.32. Declaration of rx/tx buffer pools and the atomic counter.

Then once a socket is created and the CLIENT_COUNT is incremented, the buffer pool should address a fresh buffer - avoiding conflicts (it should be noted that this is not a disconnect-friendly approach, and repeated disconnects could cause this to break). Atomics work through a load and store mechanism, where the atomic can only be loaded by one area at a time.

```
println!("Building new tcp socket for client...");

let new_client_id: usize  =
CLIENT_COUNT.load(Ordering::Relaxed).try_into().unwrap();

println!("Current client count: {}", new_client_id);

let mut client_socket = TcpSocket::new(stack, &mut
RX_CLIENT_BUFFER_POOL[new_client_id], &mut
TX_CLIENT_BUFFER_POOL[new_client_id]);

client_socket.set_timeout(Some(embassy_time::Duration::from_secs(60)));
let current_count = CLIENT_COUNT.load(Ordering::Relaxed);
CLIENT_COUNT.store(current_count.wrapping_add(1), Ordering::Relaxed);
```

Listing 4.33. The building of a new client using the buffer pools and atomic count

The code for the spawning of a new task is quite similar to what was described in the gui_handler, except the spawning of the handler is contingent on a successful accept on the current socket being await-ed. Note that port 5050 is used for the client_handler.

```
let r = client_socket
            .accept(IpListenEndpoint {
                addr: None, //across all ip addresses
                port: 5050,
            })
            .await;
        match r {
            Ok(()) =>{
                println!("Client handler spawned: {:?}",
                client_socket.remote_endpoint());
                spawner.spawn(client_handler(client_socket)).ok();
        },
            Err(e) => println!("Client connection error: {:?}", e),
        }
```

Listing 4.34. Spawning a new client_handler

To close the sockets gracefully, the same approach was used for both the gui_handler and the client_handler - namely flushing the socket, closing the socket and aborting the socket to free up the resources. It was tested as this was critical for the gui_handler, and any less than approximately one second resulted in errors. On shutdown in the client_handler, the client_count is decremented (although this may cause more harm than good by causing overwrites to occur).

```
let r = client_socket.flush().await;
if let Err(e) = r {
        println!("flush error: {:?}", e);
}
Timer::after(Duration::from_millis(1000)).await;

let current_count_post = CLIENT_COUNT.load(Ordering::Relaxed);
CLIENT_COUNT.store(current_count_post.wrapping_sub(1), Ordering::Relaxed);

client_socket.close();
Timer::after(Duration::from_millis(1000)).await;
client_socket.abort();
```

Listing 4.35. Socket shutdown code (green is only in the client_handler)[42]

4.4.2 client_handler

Each client handler is responsible for parsing the incoming JSON for its respective client device. This is done through three main steps: first, the raw data is received and converted to a string. This string is then parsed for the termination sequence "\r\n\r\n" and divided accordingly. The individual JSON objects are then converted back into the struct form that it was sent in so that the fields can be extracted and timestamped. After the data is extracted, it is routed to the correct buffer such that the order is preserved. This buffer is then written to the serialisable data struct for handling before retransmission. Here is a diagram of what happens:

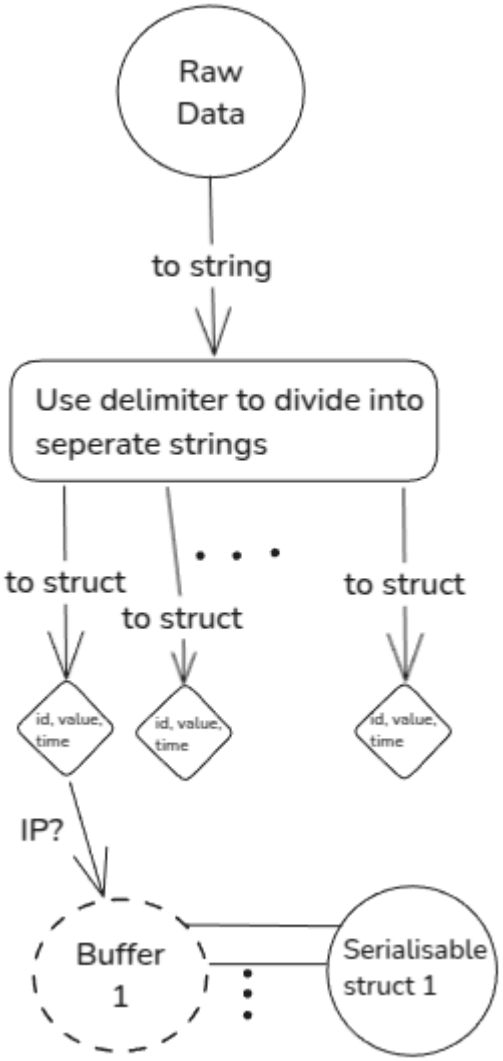


Fig 4.3. Diagram of client_handler logic

The first step is to get the raw data from the socket. Before this can be done, a buffer must be initialised and passed to the socket.read() function. A position variable must also be

declared so the loop can keep track of which slice of the buffer is being read. This variable is incremented by the length of the slice with every iteration - being reset to zero if a delimiter is detected.

```
let mut buffer = [0u8; 1024];
let mut pos = 0;
```

Listing 4.36. Declaration of loop variable and buffer to be sliced[42].

Then, the socket.read function is pattern matched for the Result value which either returns Ok(len) with the slice length, Ok(0) - indicating end of file (data no longer being received on the socket) or Err(e) with the error type. If an Err(e) or EOF is returned, this breaks out of the loop and shuts down the socket. So, if the data is there, it is converted to a string.

```
match client_socket.read(&mut buffer).await {
      Ok(0) => {
         println!("read EOF");
         break;
      }
      Ok(len) => {
         let to_print =
         unsafe { core::str::from_utf8_unchecked(&buffer[..(pos + len)]) };

... more code here ...

         pos += len;
      Err(e) => {
          println!("read error: {:?}", e);
          break; //break only on error
      }
}
```

Listing 4.37. Pattern matching the client_socket.read().[42]

Now that the string has been extracted, the delimiter can be checked for and the data can be parsed accordingly. Once the delimiter is found, the string can be split into parts based on the delimiter using the string.split("delimiter") function. This returns an iterator and substrings of the split string. This iterator can be used in a for loop to iterate over the substrings, trimming the whitespace (\r\n\r\n counts as whitespace) and then using serde_core_json to turn the substrings into individual structs. Deserialize must be used to turn strings into structs:

```rust
#[derive(serde::Deserialize)]
struct SensorData {
   sensor_id: u8,
   sensor_value: f32,
}
```

Listing 4.38. Deserialisable version of the serialisable struct from the client-side.

Here is also where the data received is timestamped. For this to work, a new struct is declared which can store an extra field alongside the ID and sensor value. This was done on the server side to ensure each sensor value arrived at the GUI with consistent timing. Embassy-time provides functionality for getting the current time from startup through Instant::now().

```rust
#[derive(Copy, Clone, serde::Serialize)]
struct TimedSensorData {
   sensor_id: u8,
   sensor_value: f32,
   sensor_time: u64,
}
```

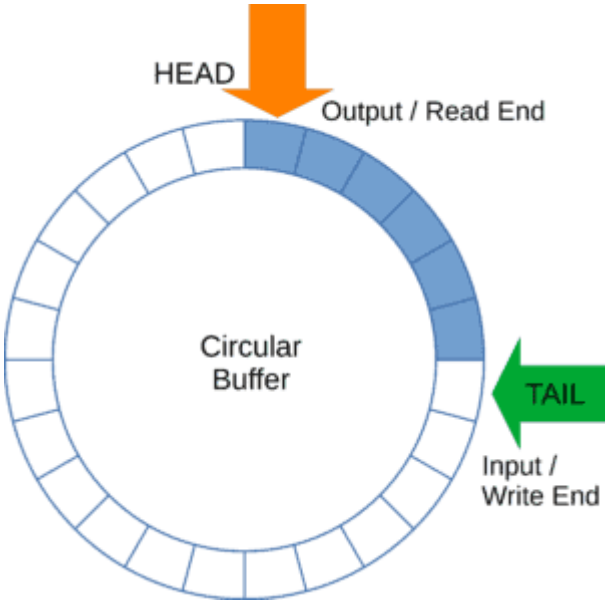
Listing 4.39. New struct for storing timestamp.

```rust
if to_print.contains("\r\n\r\n") {
    let parts = to_print.split("\r\n\r\n");
    for part in parts {
        let temp = part.trim();
        match serde_json_core::from_str::<SensorData>(temp){
              Ok((recieved,_)) => {
                    println!("Data succesfuly parsed: {}", recieved);
                    handle.sensor_id = recieved.sensor_id;
                    handle.sensor_value = recieved.sensor_value;
                    handle.sensor_time = Instant::now().as_millis();
                    print!("{}", handle.sensor_time)
              }
              Err(e) => {println!("Parsing Error: {:?}", e)}
     }
}
```

Listing 4.40. Whole process from string to struct.

Once the struct is received, it must be stored in an appropriate data structure before retransmission. To achieve this, a circular buffer was used - one for each client. A circular

buffer is a cyclical FIFO data structure which allows for the preservation of the order of the members of the struct in the order in which they were parsed. There is a crate compatible with no_std with an efficient implementation: circular_buffer[44]. Through testing, eight elements was determined to be the maximum before overwhelming the ESP32C3.




```
static mut BUF_10: CircularBuffer<8, TimedSensorData> = CircularBuffer::<8,
TimedSensorData>::new();
```

Figure 4.4./Listing 4.41. Circular buffer diagram with initialisation[45].

Finally, the circular buffer is turned into a "ClientReadings" struct. This has to be done because the circular buffer type is not serialisable and therefore cannot be used directly. To facilitate this conversion, a helper function was declared in the client_handler: circ_to_readings. This function leverages the iterator inherent to the circular buffer to iterate over each element and transfer them to their equivalent element in the "ClientReadings" struct.

```
#[derive(serde::Serialize, Clone, Copy)]
struct ClientReadings {
   readings: [TimedSensorData; 8],
}
```

```
fn circ_to_readings(buf: &CircularBuffer<8, TimedSensorData>, output: &mut
ClientReadings) {//&mut since it has to alter the real output
```

```
        let mut i = 0;
        for elem in buf.iter() {
            output.readings[i] = *elem;
            i+=1;
        }
    }
```

Listing 4.42. ClientReadings struct and the circ_to_reading function.

In order to determine the correct buffer, the sockets remote endpoint was checked for the IP address. Each IP address corresponds to a separate pair of buffer and readings struct (this ideally would've been changed to sensor_id rather than IP, allowing for dynamic IP allocation later in development).

```
unsafe{ //messing with static muts requires unsafe code
match client_socket.remote_endpoint() {
   Some(IpEndpoint {port: _, addr}) if addr ==
embassy_net::IpAddress::Ipv4(Ipv4Addr::new(192, 168, 2, 10)) => {
               BUF_10.push_back(handle);
               circ_to_readings(&BUF_10, &mut DATA_10);
               }
   Some(IpEndpoint {port: _, addr}) if addr ==
embassy_net::IpAddress::Ipv4(Ipv4Addr::new(192, 168, 2, 15)) => {
               BUF_15.push_back(handle);
               circ_to_readings(&BUF_15, &mut DATA_15);
               }
    Some(IpEndpoint {port: _, addr}) if addr ==
embassy_net::IpAddress::Ipv4(Ipv4Addr::new(192, 168, 2, 20)) => {
               BUF_20.push_back(handle);
               circ_to_readings(&BUF_20, &mut DATA_20);
               }
    Some(IpEndpoint {port: _, addr}) if addr ==
embassy_net::IpAddress::Ipv4(Ipv4Addr::new(192, 168, 2, 25)) => {
               BUF_25.push_back(handle);
               circ_to_readings(&BUF_25, &mut DATA_25);
               }
     _=> {}
}
}
pos = 0;
continue;
```

Listing 4.42. IP based matching and conversion from buffer to serialisable struct.

4.4.3 gui_handler

The gui_handler is responsible for writing the time stamped readings to the HTTP socket. Since the ClientReadings structs are static, they are visible to the GUI handler. This allows the compilation of the ClientReadings into one “TotalClientReadings” for transmission:

```
#[derive(serde::Serialize)]
struct TotalClientReadings {
   client1: ClientReadings,
   client2: ClientReadings,
   client3: ClientReadings,
   client4: ClientReadings,
}


let totalreadings: TotalClientReadings = TotalClientReadings {
               client1: DATA_10,
               client2: DATA_15,
               client3: DATA_20,
               client4: DATA_25,
           };
```

Listing 4.43. The declaration and usage of the TotalClientReadings struct.

For this to work, default values have to be declared so the values here are defined even if they are not yet full of readings.

```
const TIMED_SENSOR_DATA_DEFAULT: TimedSensorData = TimedSensorData{
   sensor_id: 0,
   sensor_time: 0,
   sensor_value: 0.0,
};

const CLIENT_READINGS_DEFAULT: ClientReadings = ClientReadings{
   readings: [TIMED_SENSOR_DATA_DEFAULT; 8]
};
```

Listing 4.44. Default values.

To make this data sendable, the serde_json_core was used very similarly to how the client originally sent the data struct, but scaled up. The TotalReadings struct is first converted to a JSON string using serde_json_core. Plugging the default version of the TotalReadings JSON into an online JSON size calculator shows 1.69KB as a bare minimum.

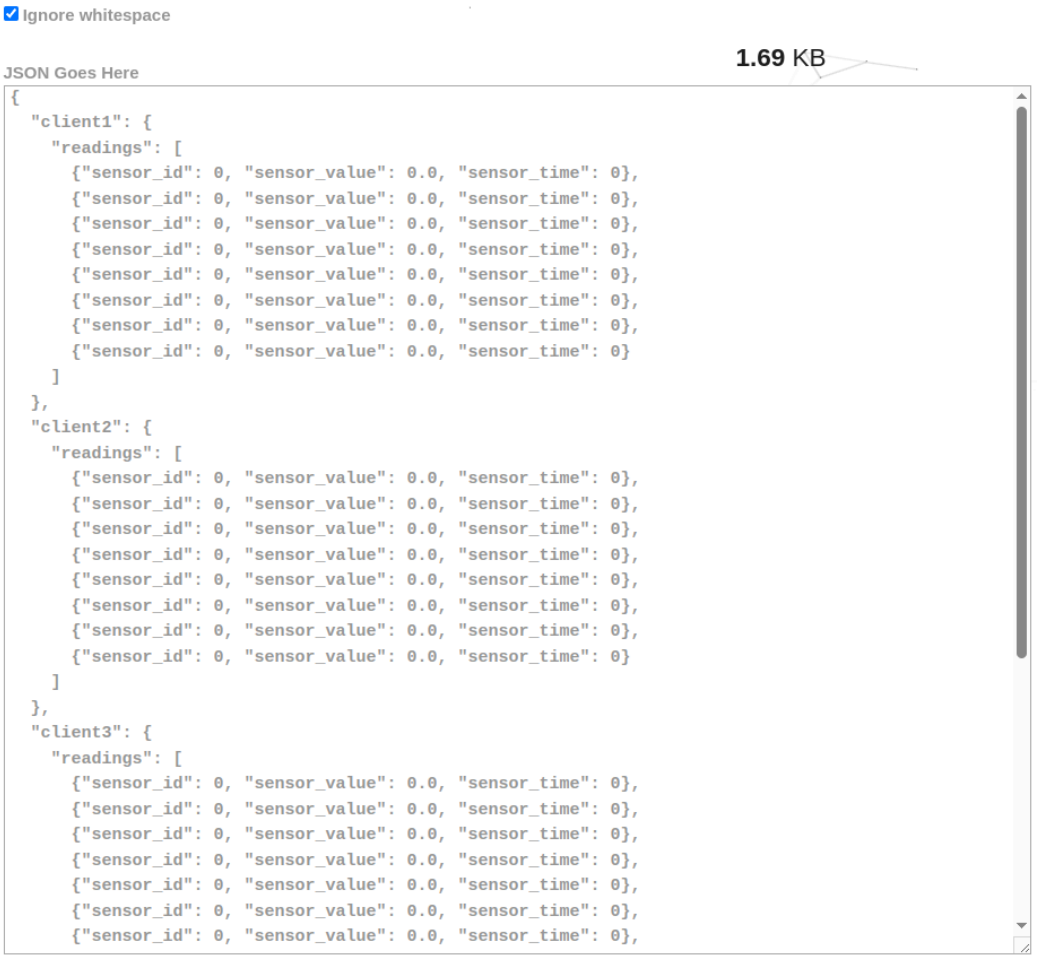


Fig 4.5. Minimum value for JSON string (w/o whitespace)[46]

To be on the safe side since longer values could result in higher memory usage, 2kB of space was used for the JSON payload. The “webpage” is then declared with 3kB of space, to account for any increase in space as a result of the HTTP response (although this could be further optimised). To actually format the response, the write! macro was used. This allows for the dynamic replacement of a placeholder in a string - which in this case attaches the payload to the HTTP response.

A simple HTTP response template was then formulated so that the receiver knows that the GET request was received, and the content-type to be expected:

```
HTTP/1.0 200 OK\r\nContent-Type: application/json\r\n\r\n{}
```

All together:

```
let jsonpayload:String<2000>  =
serde_json_core::to_string(&totalreadings).unwrap();
let mut webpage: String<3000> = String::new();
write!( webpage,
    "HTTP/1.0 200 OK\r\nContent-Type: application/json\r\n\r\n{}",
     jsonpayload,
).unwrap();
```

Listing 4.45. Code for dynamically writing the Readings to the webpage[42].

Finally, similar to when the data was sent from the client-side, the webpage is written to the socket:

```
let r = socket.write_all(webpage.as_bytes()).await;
if let Err(e) = r {
    println!("write error: {:?}", e);
}
```

Listing 4.46. Writing the webpage to the socket[42].

4.5 GUI Implementation

This section shows the implementation of the GUI. As shown in Chapter 3, the stack is PySide and QML.

4.5.1 Networking

The entirety of the backend functionality was implemented in the Netstuff class. This includes both the network handling and the graphing of the data.

In order to interface with the gui_handler on the server side, the library QtNetwork was used. The first step in this process is to send a GET request to the target: 192.169.2.1:8000. This was done by simply implementing the sendRequest() function with reference to the constructor. The “finished” signal is “connected” to the processReply function. This means the reply is processed once the access manager returns “finished”.

```
self.nam = QNetworkAccessManager(self)
self.nam.finished.connect(self.processReply)
```

```
def sendRequest(self):
```

```
        url = QUrl("http://192.168.2.1:8000")
        request = QNetworkRequest(url)
        self.reply = self.nam.get(request)
```

Listing 4.47. Sending request code and constructor for the access manager[47].

When this is finished, the processReply function is triggered. This function not only checks the reply for errors, but handles the received data for display on the GUI. If reply.error() is clean, the serialised data is deserialised to text.

```
def processReply(self, reply: QNetworkReply):
        er = reply.error()
        if er == QNetworkReply.NoError:
            answerAsText = bytes(reply.readAll()).decode("utf-8")
...
        else:
            print("Error occured: ", er)
            print(reply.errorString())
        reply.deleteLater() #cleans up resources
```

Listing 4.48. Deserialisation of received reply and error handling[47].

The frequency at which the server is polled is set in main. Considering that it was found through testing that the server takes two seconds to clean up the socket, the server should be polled more frequently than that, and preferably with a clean divisor of 2000ms. If this is followed, the requests should synchronise with the cleanup. To achieve this, a QTimer is started with the timeout signal connected to the sendRequest function.

```
netstuff = Netstuff(Data)
timer = QTimer()
timer.timeout.connect(netstuff.sendRequest)
timer.start(2000)
```

Listing 4.49. Server polling code.[48]

4.5.2 JSON Parsing

The JSON was parsed using the dedicated json package. Once received, the text is loaded into the loads() function. This turns the JSON text into a Python dictionary. This dictionary can be divided into lists - one for each client. These lists can then be parsed with for loops to extract the data contained for each client. The client1 readings are stored in the data1 list.

This list is iterated through for each target which in essence splits the list into four smaller lists corresponding to the JSON fields.

```
readings = json.loads(answerAsText)
data1 = readings["client1"].get("readings", [])
times1 = [item["sensor_time"] for item in data1]
values1 = [item["sensor_value"] for item in data1]
id1 = [item["sensor_id"] for item in data1]
```

Listing 4.50. Turning text into lists - one for each field[49].

4.5.3 Graphing

The functionality provided in PyQtGraph was used in order to graph the incoming sensor_values. However, this library does not contain an interpolation scheme aside from linear. To achieve a somewhat smoothly reconstructed signal, the SciPy interpolate library was used. The following snippet first constructs a linespace of 100 values using NumPy. This provides a denser and smoother set of time values for the x-axis. The make_interp_spline(_, _, 3) function generates the cubic spline interpolation function, which can then be used to compute the spline at each x-axis value.

```
xnew = np.linspace(min(times1), max(times1), 100)
spl = interpolate.make_interp_spline(times1, values1, 3)
ynew = spl(xnew)
```

Listing 4.51. Interpolation code[50].

This (xnew, ynew) pair can now be graphed. The Netstuff constructor is responsible for setting up each of the graphs by producing each window, adding the graph to said window and then showing the window like so:

```
self.win1 = pg.GraphicsLayoutWidget(title="Client 1 Voltage (V) vs Time
(ms)")
self.plot1 = self.win1.addPlot(title="Client 1 Voltage vs Time")
self.win1.show()
```

Listing 4.52. Setting up the graph [51]

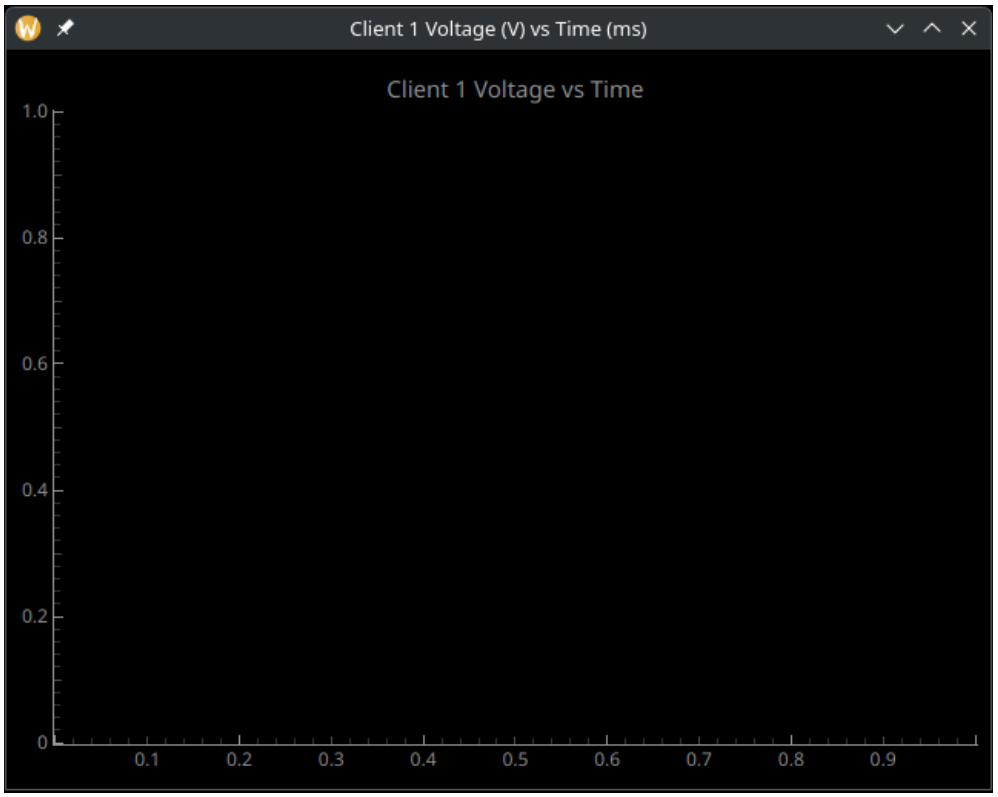


Fig 4.6. The blank graph produced

Using the plot1 handles, one can now use the plot() function to simply plot the values. These will be updated automatically as the values available in the Netstuff QObject are updated (provided the old data is cleared first).

```
self.plot1.clear()
self.plot1.plot(xnew, ynew, pen='r', symbol='o')
```

Listing 4.54. Plotting of the interpolated data points.

4.5.4 Calculating Energy

The central goal of this project is to monitor energy. To do this, one must first derive the power as was done in Chapter 2. Using this derivation, the instantaneous power can be calculated with only the voltage and resistance (provided the power factor ≅ 1). The chosen appliances were a toaster, a halogen light bulb, a kettle and an iron as these all satisfy that condition[]. The resistance was calculated using the power values from the paper combined with the mains voltage of 230 volts. Four helper functions were declared to aid with the voltage to power conversion:

```
    def calcPowerToaster(self, input: float) -> float:
        voltage= input/1000.0
        resistance = 60.80
        return round(((voltage ** 2)/resistance), 4)

    def calcPowerHalobulb(self, input: float) -> float:
        voltage= input/1000.0
        resistance = 18892.29
        return round(((voltage ** 2)/resistance), 4)

    def calcPowerKettle(self, input: float) -> float:
        voltage= input/1000.0
        resistance = 24.05
        return round(((voltage ** 2)/resistance), 4)

    def calcPowerIron(self, input: float) -> float:
        voltage= input/1000.0
        resistance = 26.45
        return round(((voltage ** 2)/resistance), 4)
```

Listing 4.55. Helper functions

These function were used to convert the values parsed from the data list into power values:

```
power_values1 = [self.calcPowerToaster(v) for v in values1]
```

Listing 4.56. Power values calculation

The energy can then be calculated by integrating across each interval and summing these intervals for the total power - both are displayed. SciPy has an integration utility which can numerically integrate using Simpson's rule over the given values and times (since an even number is provided, the simpsons is used for the first N-2 intervals with “the addition of a 3-point parabolic segment for the last interval using equations outlined by Cartwright [1].”)[53]

```
energy1 = integrate.simpson(power_values1, x=times1)

self.Data["client1"].totalenergy = self.Data["client1"].totalenergy +
energy1
```

Listing 4.57. Simpson's rule and summation of intervals for total energy

4.5.5 Exposing Data to Frontend

In order for any of the data calculated to be shown visually, there had to be a way to expose it to the QML. To do this, a separate "DataGui" class was defined.

```
class DataGui(QObject):

    def __init__(self, clientid=0):
        super().__init__()
        self._energy = 0.0
        self._totalenergy = 0.0
        self._clientid = clientid
```

Listing 4.58. DataGui class constructor[54].

Then, using the Signal and Property features of PySide, not only would the data be visible to the frontend, but the data would dynamically update as the change signals were received. Firstly, the signal functions were declared. These would then emit a signal upon the detection of a change (note: the getters and setters are required methods to form a Property).

```
    energyChanged = Signal()
    def getEnergy(self):
            return self._energy

    def setEnergy(self, value):
            if self._energy != value:
                self._energy = value
                self.energyChanged.emit()
```

Listing 4.59. Signal declaration and triggering in the setter[54].

Now that there is a signal, a getter and a setter, one can form a Property. This Property will not only contain the real float value of the field, but will also be notified on the changing of the value.

```
energy = Property(float, getEnergy, setEnergy, notify=energyChanged)
```

Listing 4.60. Property wrapper defined for actual energy value[54].

These steps are then repeated for the totalenergy and clientid fields. However, this only creates the necessary Properties. Now, the data needs to be exposed to the Netstuff backend

so that it can be edited, and to the QML engine. For the Netstuff, four Data instances are constructed and combined into a single Data dictionary - one for each client. This is then passed as an argument to the Netstuff backend:

```
    Data1 = DataGui(clientid=1)
    Data2 = DataGui(clientid=2)
    Data3 = DataGui(clientid=3)
    Data4 = DataGui(clientid=4)

    Data = { #encapsulate all data in a single dictionary
        "client1": Data1,
        "client2": Data2,
        "client3": Data3,
        "client4": Data4
    }

netstuff = Netstuff(Data)
```

Listing 4.61. DataGui object construction and passing to the backend

The QML application engine can then be started up, and rooted to each of the previously defined DataGui objects. This works only because the fields are Properties. Without this, the QML cannot access the individual values.

```
engine = QQmlApplicationEngine()
engine.rootContext().setContextProperty("data1", Data1)
engine.rootContext().setContextProperty("data2", Data2)
engine.rootContext().setContextProperty("data3", Data3)
engine.rootContext().setContextProperty("data4", Data4)
```

Listing 4.62. Exposing the Properties to the QML engine.[55]

4.5.6 Frontend

The energy for each interval and the total energy are displayed via QML. There are four textboxes, one for each client. These are displayed in a single column. Thanks to the Properties defined in the backend, the QML values update dynamically without needing a refresh or inline JavaScript.

```
Window {
    width: 640
```

```
    height: 480
    visible: true
    title: qsTr("Energy Monitor")

    ColumnLayout {
    anchors.centerIn: parent

    Text {
            text: "Client ID: " + data1.clientid + "    Current Energy: "
                + data1.energy + " J" + "    Total Energy: "
                + data1.totalenergy + " J"
            font.pixelSize: 10
        }
    ...
    }

}
```

Listing 4.63. QML frontend layout[55]

This produces a white window, containing fields for each of the corresponding DataGui fields:

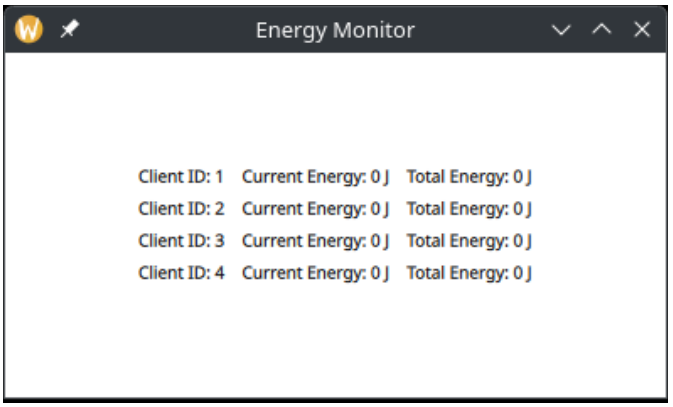


Figure 4.7. Resulting QML before the server is connected (changed during testing)

4.6 Testing

There are a lot of technologies being used for this project. As such, many unit tests were performed during the development, most of which are included in the appendix. This section will only focus on the methods of testing on the pseudo-final version of the project, the results are shown and discussed in Chapter 5. Some changes were made during testing to

either improve the functionality, readability or to correct minor mistakes and edge cases. These changes are also documented in Chapter 5.

4.6.1 Full System Test (End-to-End)

The first test is to confirm that, under normal conditions, the system can operate as expected. In this test, the server is first powered up. Once ready, the clients are all powered up via USB cable. The host device connects to the access point and the GUI is launched. For clients 1, 2 and 4, random test data is generated. Client 3 sends real data generated by scopy through the ADC, which is also powered by the voltage generator.

Figure 4.8. Voltage generator (only positive is enabled here)

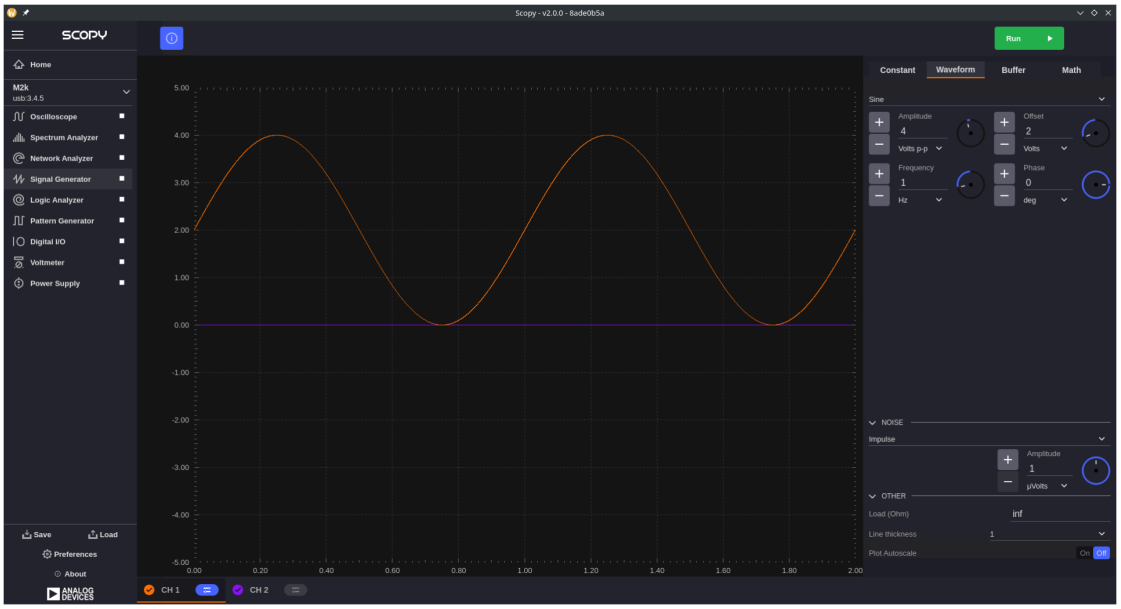


Figure 4.9. Scopy configuration for signal generator

The functioning of the GUI was monitored both before and after the signal generation was enabled.

4.6.2 Determine Maximum Possible Clients

To conduct this test, the server-side code was adapted to see whether or not the number of potential clients could be increased. To achieve this, three parts of the server-side code were considered:

```
let (stack, runner) = embassy_net::new(
    wifi_interface,
    config,
    mk_static!(StackResources<7>, StackResources::<7>::new()),
    seed,
);
```

Listing 4.64. Stack resource allocation

```
static mut RX_CLIENT_BUFFER_POOL: [[u8; 1536]; MAX_CLIENTS] = [[0u8; 1536];
MAX_CLIENTS];
static mut TX_CLIENT_BUFFER_POOL: [[u8; 1536]; MAX_CLIENTS] = [[0u8; 1536];
MAX_CLIENTS];
```

Listing 4.65. Static client buffer pool

```
#[embassy_executor::task(pool_size = MAX_CLIENTS)]
```

Listing 4.66. Client task pool size

These were incremented to see the limit of clients. Each change was monitored in isolation and together to determine the exact point of failure.

4.6.4 Client Disconnect/Reconnect to Server

This is the first of three tests, each of which examine the robustness of the system to disconnects and reconnects. For this first test, the server was monitored while clients were repeatedly reset. Then, a client was monitored to see the effect on both ends.

4.6.5 Server Disconnect/Reconnect to Clients

Similarly to the previous test, the server was disconnected during client transmission and reconnected. First, the client was monitored for the effects. Then, the test was repeated but the server was monitored.

4.6.5 Server Disconnect/Reconnect to GUI

Finally, the GUI was shutdown and reconnected to the server while data was being transmitted. This was tested for a manual shutdown of the Python code, and a disconnect of the host device to the access point.

4.6.6 Data Verification and Optimisation

For this test, the received I2C data was checked when a constant signal is generated rather than a sine wave. Scopy was set to generate a constant 3V signal:

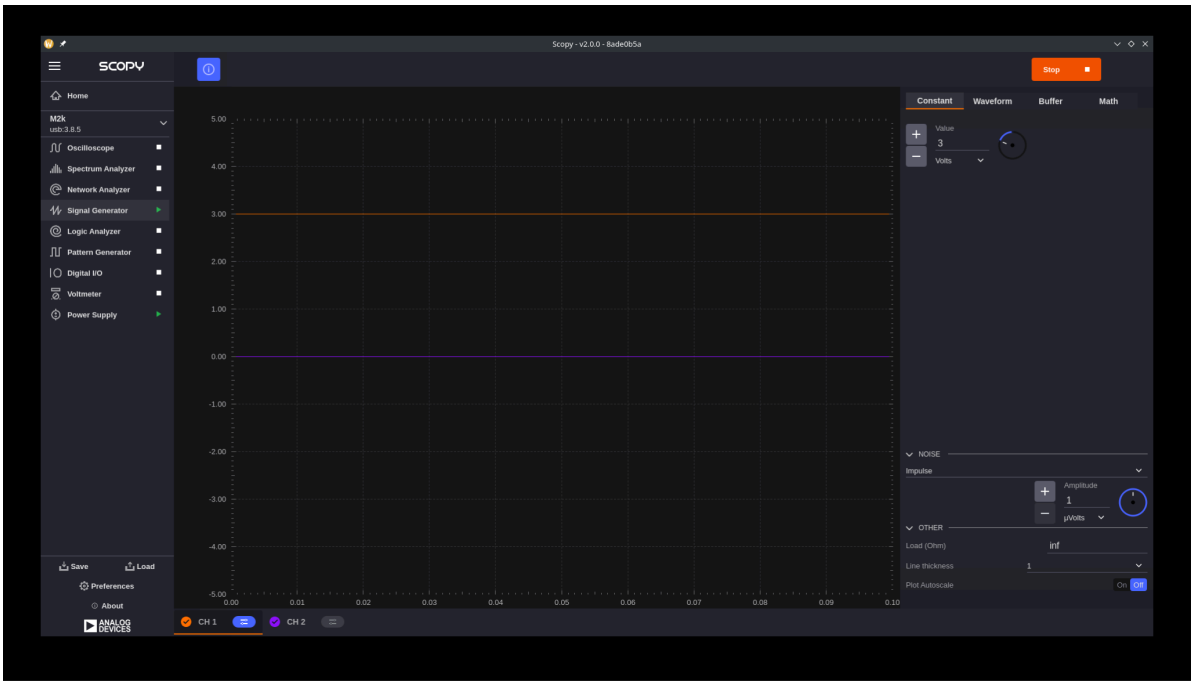


Fig 4.10. Scopy generating a constant value of 3V

The next test which should be conducted is the Total Energy test. In this case, a timer is started when the GUI connects (and therefore starts recording). The data is then left to accrue for two minutes. Then, dividing the Total Energy by the total number of intervals should produce an average value for the energy expenditure per two second interval.

Then, the polling rate was reduced to 100ms. This could potentially reduce data loss, althought the socket still has 2000ms of downtime.

```
timer = QTimer()
timer.timeout.connect(netstuff.sendRequest)
timer.start(2000)
```

Listing 4.67. Target code to optimise in the GUI backend

Next, the socket downtime on the server code was decreased in order to find a minimum.

```
let r = socket.flush().await;
                    if let Err(e) = r {
            println!("flush error: {:?}", e);
        }
```

```
Timer::after(Duration::from_millis(1000)).await;

socket.close();
Timer::after(Duration::from_millis(1000)).await;

socket.abort();
```

Listing 4.68. Target code to optimise on the server-side

The failure conditions were analysed using wireshark.

4.6.7 Penetration Testing

As this project concerns data in IoT, some basic pen testing was conducted with the aim of locating the data without the GUI. The tests were performed assuming the attacker knows nothing about the system.

Chapter 5 - Results and Discussion

This chapter shows the results from the tests described in Chapter four and analyses the results. Any changes to the source code influenced by the results of the testing are also detailed. Each subsection here corresponds to a test outlined in Chapter 4.

5.1 Full System Test (End-to-End)

After the server was started, each client was booted up. The host device was connected to the access point and the GUI successfully started up, dynamically capturing the raw data and processing it. The raw voltages were successfully graphed and interpolated and the interval/total energy values were displayed and updated. The validity of this data is discussed in section 5.6. Firstly, before the signal generation took place:

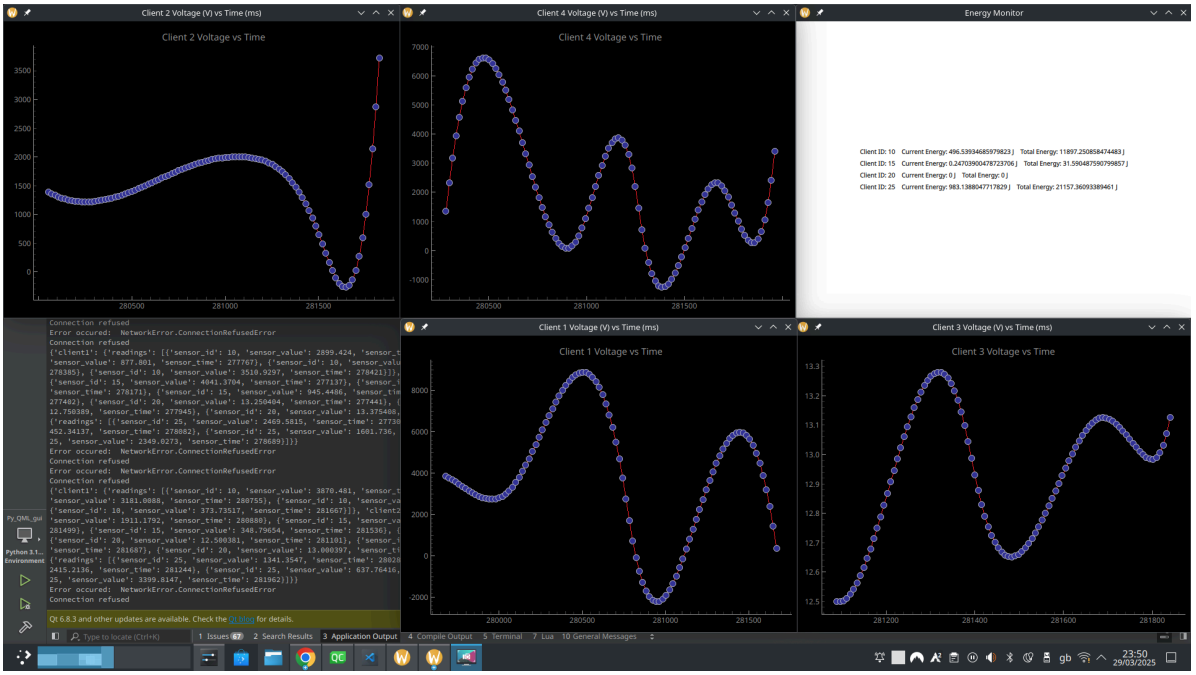


Fig 5.1. I2C inactive

And after the scopy signal generator was activated:

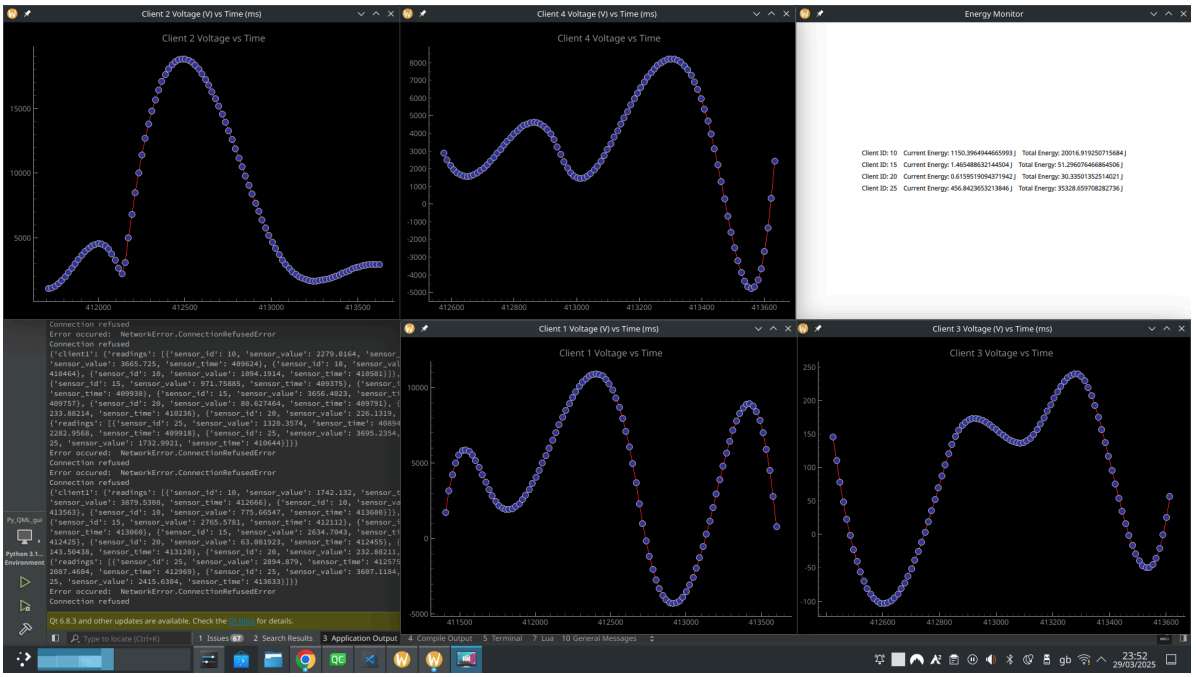


Fig 5.2. I2C active

The differences between these two graphs can be seen in the scaling. Once signal generation begins, the scaling increases from an order of one to an order of around 4. In the bottom left of both images, the received raw JSON values are shown.

During this test, an edge case arose where the fourth client could not connect. This was due to the "building socket" loop continuing to build new sockets after the fourth client connected - causing an out of bounds error on the client buffer pools. To fix this, a simple if condition to break out of the loop to an asynchronous timer is added when the atomic CLIENT_COUNT passes three.

```
TcpSocket::new(stack, &mut RX_CLIENT_BUFFER_POOL[new_client_id], &mut
TX_CLIENT_BUFFER_POOL[new_client_id]);


if new_client_id > 3 {break;}
...
loop{
    Timer::after(Duration::from_secs(5)).await;
    }
```

Listing 5.1. Edge case cause and solution.

Then, a buffer full error was thrown on the JSON payload string. This was fixed by increasing the jsonpayload from 2kB to 4kB.

```
let jsonpayload:String<4000>  =
serde_json_core::to_string(&totalreadings).unwrap();
```

Listing 5.2. Increased jsonpayload.

5.2 Determine Maximum Possible Clients

Instead of simply increasing the MAX_CLIENTS static to determine if another client could be added, each entry was individually increased so as the determine the exact cause of the error:

```
    let (stack, runner) = embassy_net::new(
        wifi_interface,
        config,
        mk_static!(StackResources<8>, StackResources::<8>::new()),
        seed,
    );
```

Listing 5.3. Stack resource allocation change

```
static mut RX_CLIENT_BUFFER_POOL: [[u8; 1536]; MAX_CLIENTS+1] = [[0u8;
1536]; MAX_CLIENTS+1];
static mut TX_CLIENT_BUFFER_POOL: [[u8; 1536]; MAX_CLIENTS+1] = [[0u8;
1536]; MAX_CLIENTS+1];
```

Listing . Static client buffer pool changes

```
#[embassy_executor::task(pool_size = MAX_CLIENTS + 1)]
```

Listing 5.4. Client task pool size change

Firstly, the MAX_CLIENT buffers were increased by 1. This was done separately to the increased stack resources. Upon further testing, the stack resources could be increased to 8 and the client_handler pool could be increased by 1, but as long as there was an extra buffer declared, the following error occurred:

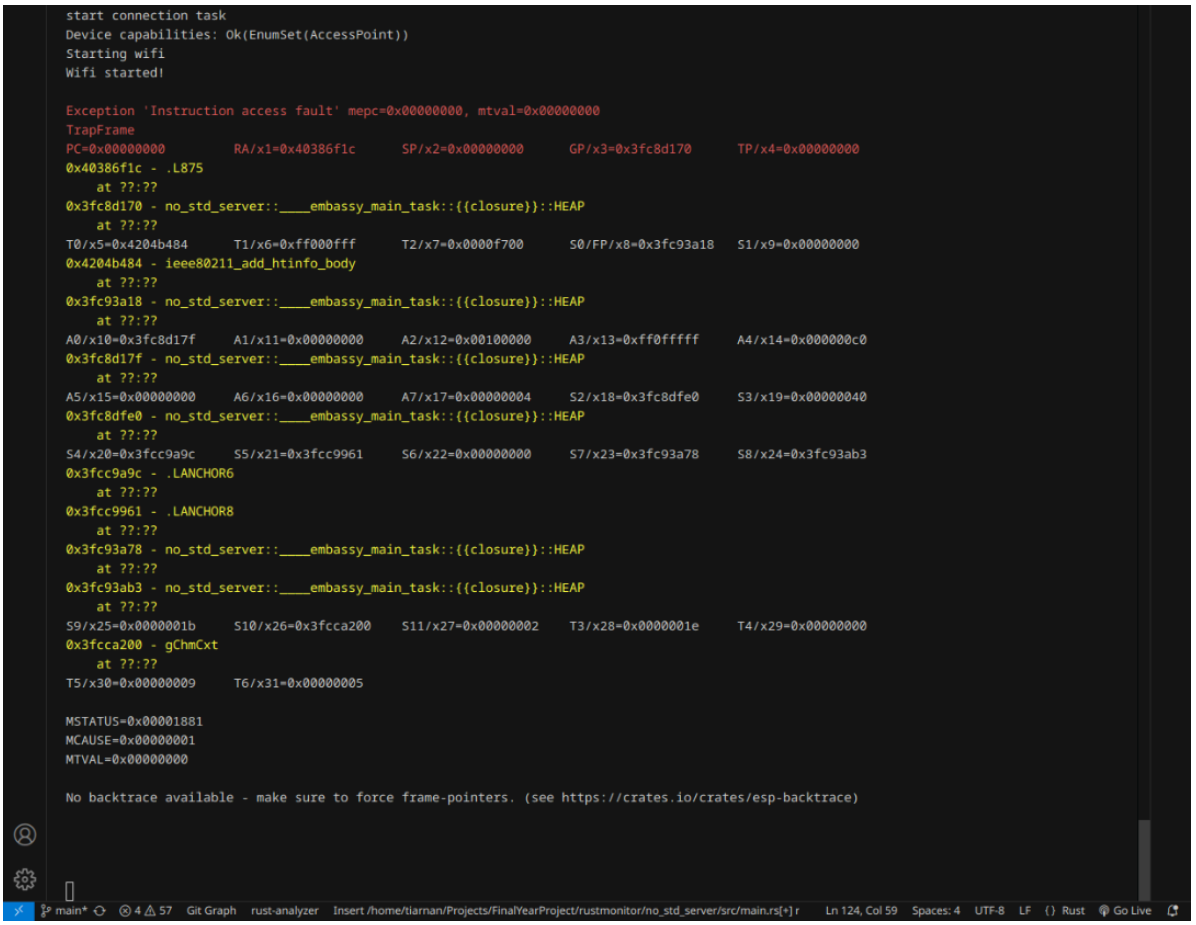


Fig 5.3. Error thrown on extra buffer pool (unaffected by stack resource change)

In RISC-V, the instruction access fault indicates that an area reserved for M (machine access mode) was attempted to be accessed by a U (user mode) program. This is part of the physical memory protection provided by the RISC-V architecture[56]. This likely indicates that after increasing the buffers, the ROM was encroached upon by the program. This suggests a hard limit of four clients for this implementation of the server on an ESP32C3.

5.3 Client Disconnect/Reconnect to Server

In each case, the disconnected client was able to reconnect with the server after the set socket timeout had elapsed - but not before then.

Fig 5.4. Point at which final client was reset (server)

Then, the same client was repeatedly connected and disconnected to check if the buffers became overloaded, but in each case the client count was successfully decremented by the shutdown code in the client side.

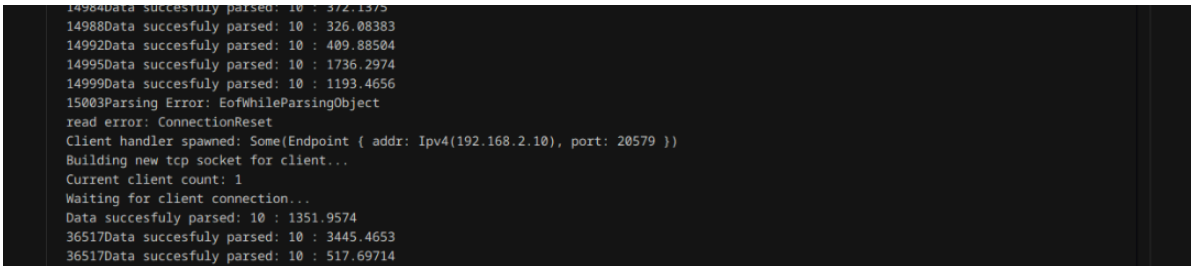


Fig 5.5. Successful reconnection.

However, when the same procedure was done with two clients simultaneously, the buffers conflicted resulting in neither being able to reconnect. This occurred because there is no guarantee the correct buffers are freed up in the code. Therefore, two clients can be told to access the same buffer resulting in a data race. This would likely not occur if a locking mutex was implemented for the static mutable buffers, or if there was a data structure keeping track of the currently busy buffers to avoid reassignment.

Fig 5.6. Buffers conflicting.

When a client was monitored, no change was seen client-side between the first and second connection attempt:

Fig 5.7. On reconnect, the client successfully connected to the server.

5.4 Server Disconnect/Reconnect to Client

This time, it's simply a reverse of the last test, where the server is shut down while the clients are kept alive.

Fig 5.8. Client monitor paused with no error or visual indication before resuming transmission after reconnect.

Next, the server was monitored:

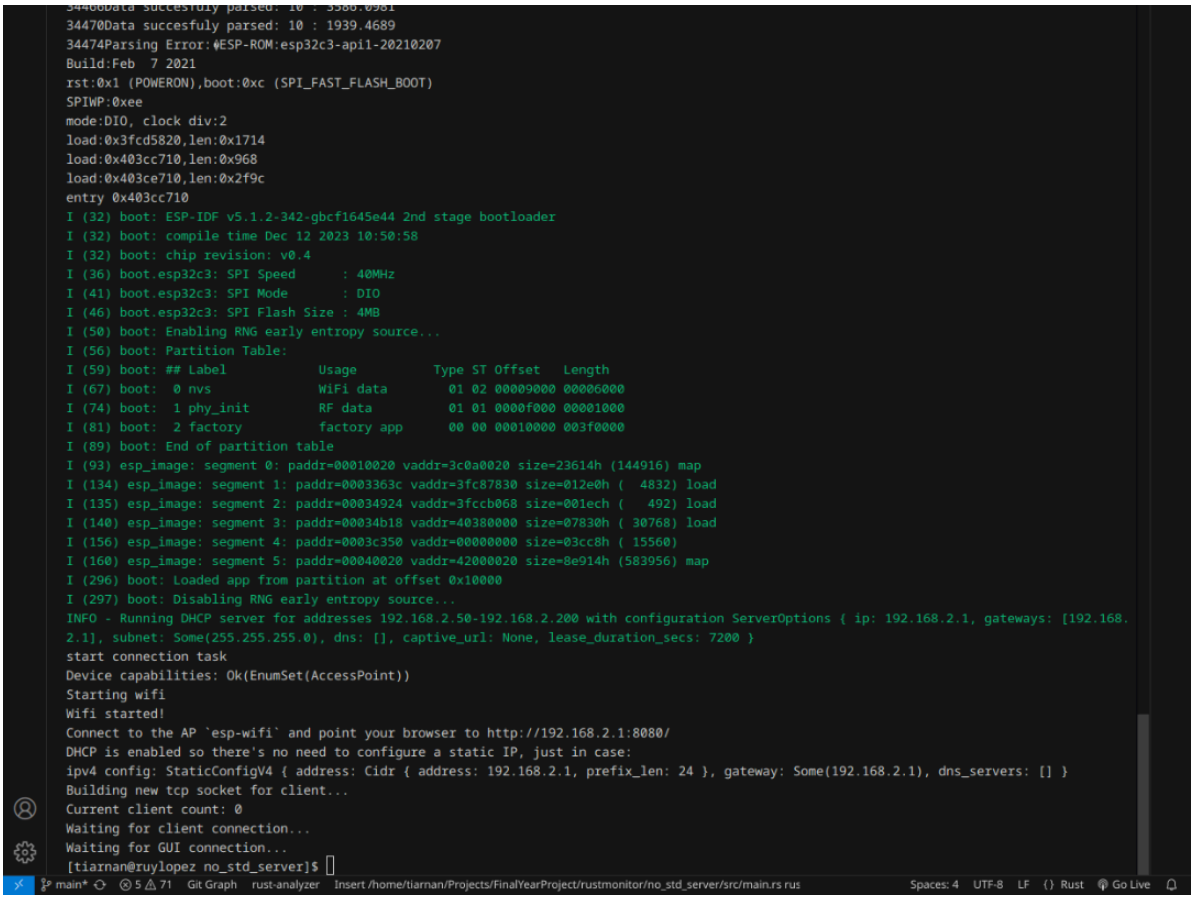


Fig 5.9. The server failed to reconnect to the disconnected clients

This occurs due to a lack of retry code on the client end. If the error was handled rather than unwrapped, the client could conditionally retry the connect code.

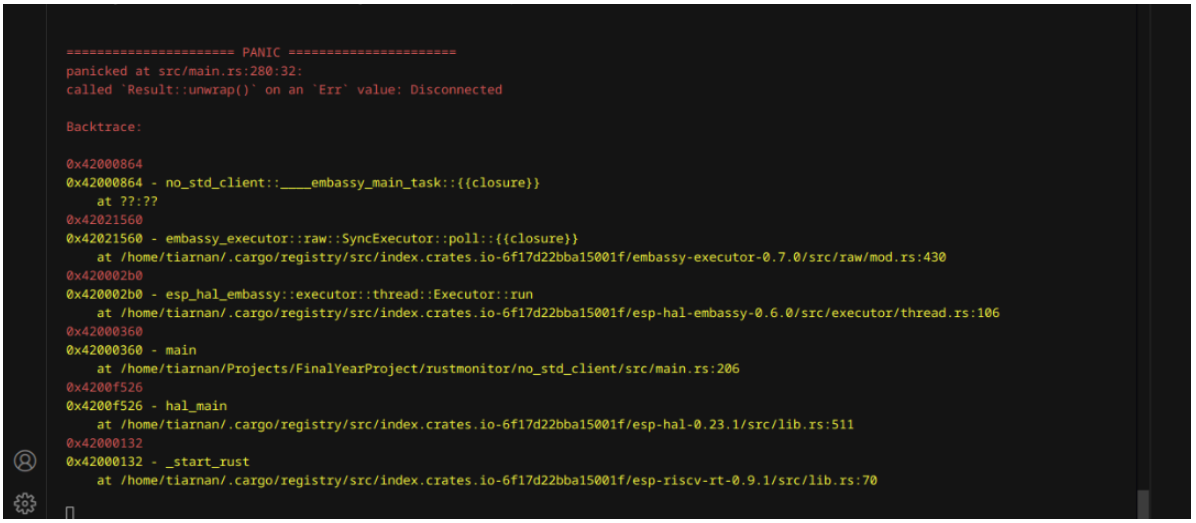


Fig 5.10. Disconnection error

5.5 Server Disconnect/Reconnect to GUI

The GUI successfully reconnected to the server and no change to the server monitor was seen. The total energy, however, was reset to zero - the previous data being lost. This could be fixed by sending the current total energy back to the server for storage and then retrieving the total as the constructor value to the DataGui class.

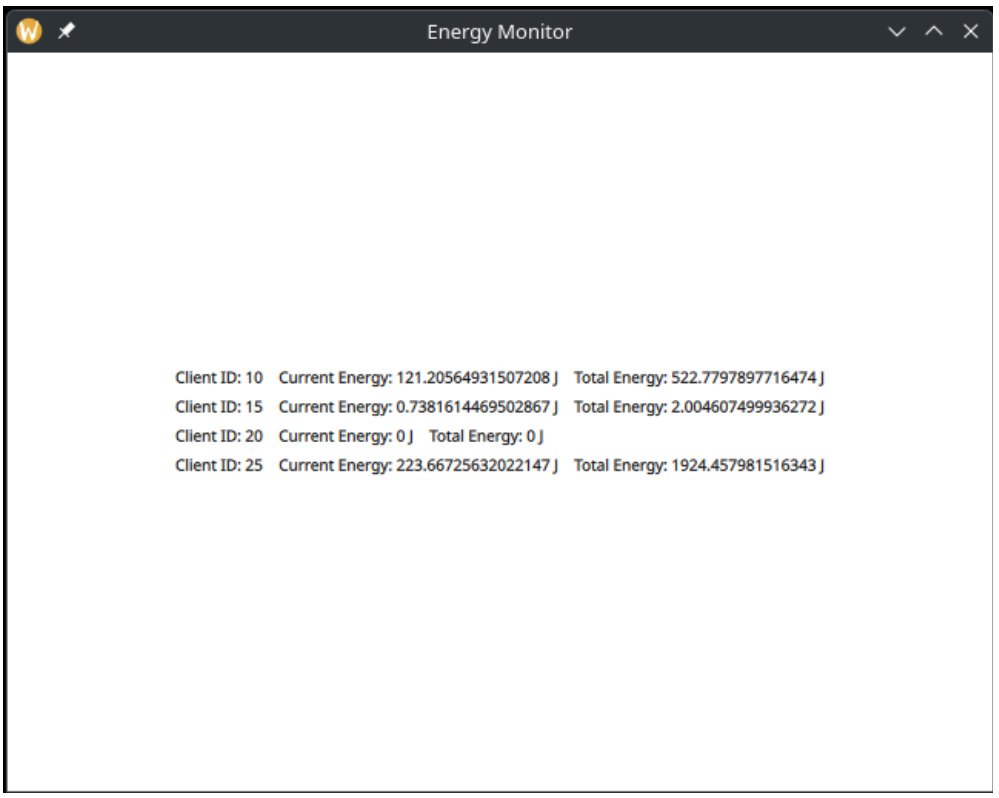


Fig 5.11. After reconnect, old energy was lost (low values for total)

These results were the same whether the GUI was shutdown manually by ending and restarting the task, or by disconnecting and reconnecting the host device from the access point. This is because the HTTP endpoint on the server is not reliant on a continuous connection like the client-server code. This allows the GUI to simply reconnect whenever it wants.

5.6 Data Verification and Optimisation

This section details the results and fixes which were implemented to ensure that correctly scaled data was being processed and displayed. First, the energy for a single interval was checked:

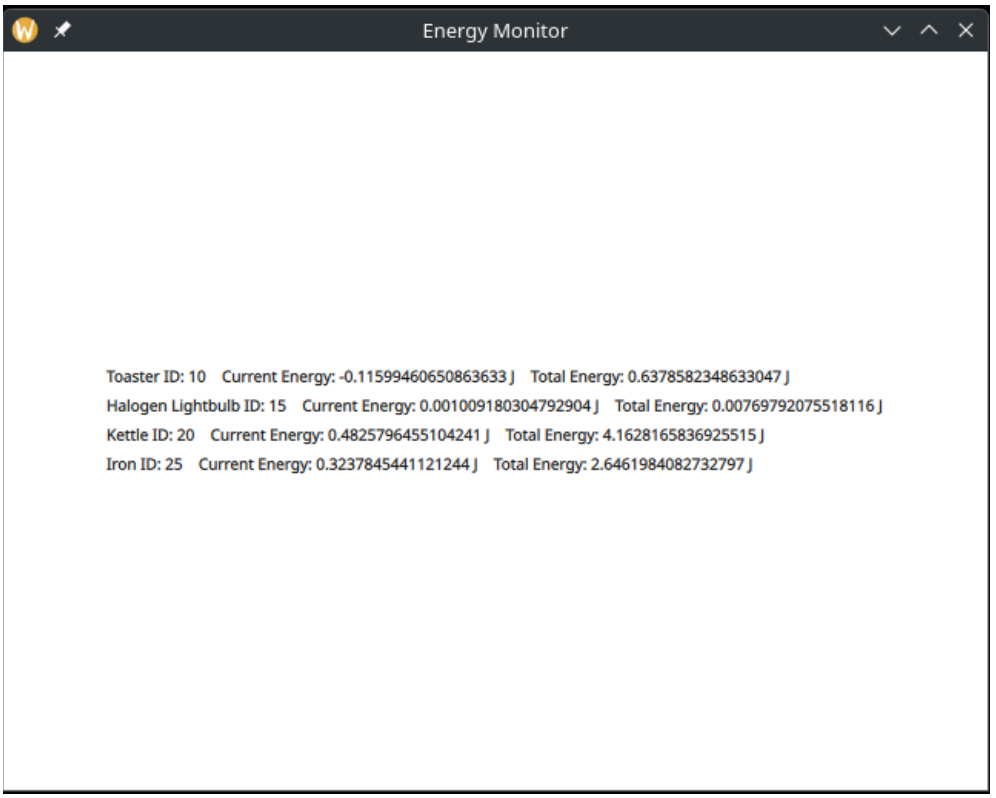


Fig 5.12. 0.48J output observed in this interval.

Given these results, and an approximate interval of 2000ms
Kettle R = 24.05 Ω
Instantaneous voltage = 3V

E = Pt = (3)^2/24.05 = 0.374W * 2 = 0.674J.

This is close but not equal to the 0.48J listed, however it does seem to hover around 0.6J but this is entirely subjective.

Toaster ID: 10 Current Energy: -0.05869704577983634 J Total Energy: 4.158601152199311 J
Halogen Lightbulb ID: 15 Current Energy: 0.0002462609576001001 J Total Energy: 0.01748543473578408 J
Kettle ID: 20 Current Energy: 0.3644051701020816 J Total Energy: 16.64111543036611 J
Iron ID: 25 Current Energy: 0.17166946429378654 J Total Energy: 12.34101922905483 J

Fig 5.13. Total energy after two minutes

After two minutes, 16.64J of energy was expended by the kettle client.
16.64/(120/2) = 0.185J per interval

This resulted in the following graph output of roughly 3000mV which is correct.

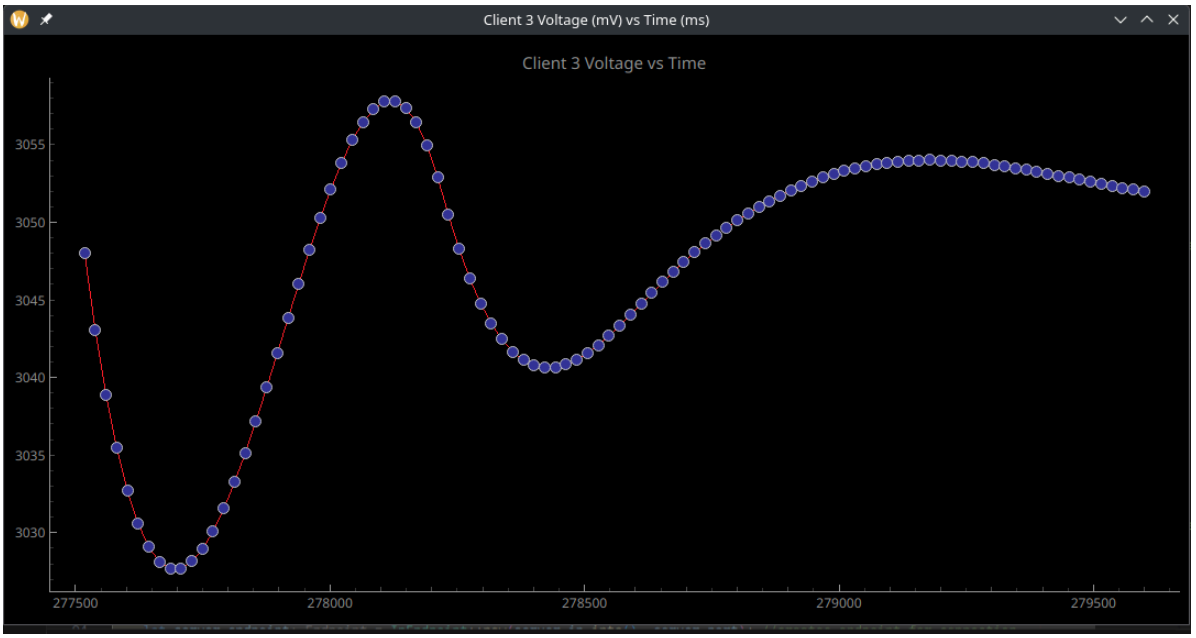


Fig 5.14. Voltage to time graph (mV, ms) displaying 3000mV

The real data is from the third client with ID 25, this is shown to be a kettle. Curiously, the interval data seems to change quite erratically - shifting from as much as 0.8J to as little as 0.3J in an interval whereas the voltage data only shifts +-50mV.

```
timer = QTimer()
```

```
timer.timeout.connect(netstuff.sendRequest)
timer.start(100)
```

Listing 5.5. Target code to optimise in the GUI backend

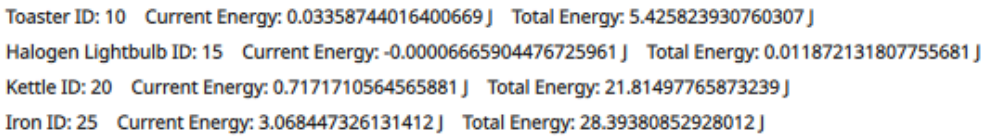


Fig 5.15. Increased polling rate to once per 100ms now, 21.8/60 = 0.363J per interval

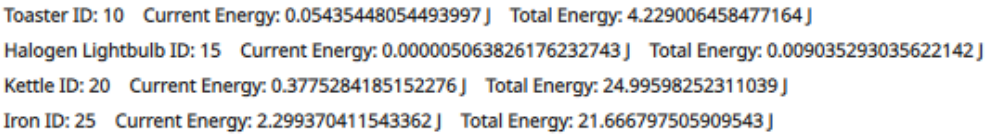


Fig 5.16. Increased again to once per 10ms. 25/60 = 0.41J per interval

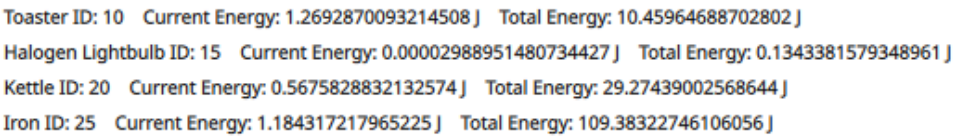


Fig 5.17. Increased to once per 1ms. 29/60 = 0.48J

This trend shows that with increasing polling rate, more data is captured every interval by acting earlier. This makes sense as the socket can have an irregular delay before it is ready to respond with the data, such as TCP retransmissions.

Stalls occurred if the values were dropped below the ones shown below:

```
let r = socket.flush().await;
                    if let Err(e) = r {
            println!("flush error: {:?}", e);
        }
        Timer::after(Duration::from_millis(100)).await;

        socket.close();
        Timer::after(Duration::from_millis(100)).await;

        socket.abort();
```

Listing 5.6. Minimised socket downtime

To observe the network activity during the server stalls, wireshark (or tshark) was used to scan for TCP packets:

```
sudo tshark -f tcp
```

```
  735 20.812902027 192.168.2.50 → 192.168.2.1  TCP 74 53234 → 8000 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TS
val=2253144391 TSecr=0 WS=128
  736 20.923730279  192.168.2.1 → 192.168.2.50 TCP 66 8000 → 53234 [SYN, ACK] Seq=0 Ack=1 Win=1440 Len=0 MSS=1438 WS
=1 SACK_PERM
  737 20.923762299 192.168.2.50 → 192.168.2.1  TCP 54 53234 → 8000 [ACK] Seq=1 Ack=1 Win=64256 Len=0
  738 20.923842970 192.168.2.50 → 192.168.2.1  HTTP 209 GET / HTTP/1.1
  739 21.215935027  192.168.2.1 → 192.168.2.50 TCP 1492 HTTP/1.0 200 OK
  740 21.215995671 192.168.2.50 → 192.168.2.1  TCP 54 53234 → 8000 [ACK] Seq=156 Ack=1439 Win=67072 Len=0
  741 21.325247015  192.168.2.1 → 192.168.2.50 TCP 152 8000 → 53234 [PSH, ACK] Seq=1439 Ack=156 Win=1452 Len=98
  742 21.325274016 192.168.2.50 → 192.168.2.1  TCP 54 53234 → 8000 [ACK] Seq=156 Ack=1537 Win=67072 Len=0
  743 21.342898764  192.168.2.1 → 192.168.2.50 TCP 655 8000 → 53234 [PSH, ACK] Seq=1537 Ack=156 Win=1452 Len=601
  744 21.342930985 192.168.2.50 → 192.168.2.1  TCP 54 53234 → 8000 [ACK] Seq=156 Ack=2138 Win=69888 Len=0
```

Fig 5.18. Output of wireshark command upon stall (GUI IP is 192.168.2.50)

From this response, we can see that the issue stems from the server. This is because the final response is an acknowledgment of the received data by the GUI. If the server was still transmitting, the final packet would show 8000 -> 53234 (port numbers) [PSH,ACK]. This would imply that port 8000 is providing the prepared data to the GUI which is not the case.

Another phenomenon that occurred was the retransmission of the same data. The total energy could be seen to be increasing, whereas the readings remained constant which is not what one would expect from random data.

The reason for this was likely the overwhelming of the WiFi. Combined with TCP's built-in congestion control, this could have caused disconnects or endless retransmissions. The increased traffic is due to the decreased delays on both the GUI and server end. When operating at the lower downtime and increased polling rate, the GUI clearly did not update

once every 200ms. Below, there are 11 rejected GET requests at a 100ms polling rate. If operating correctly, there should only have been at most three before the socket became available.

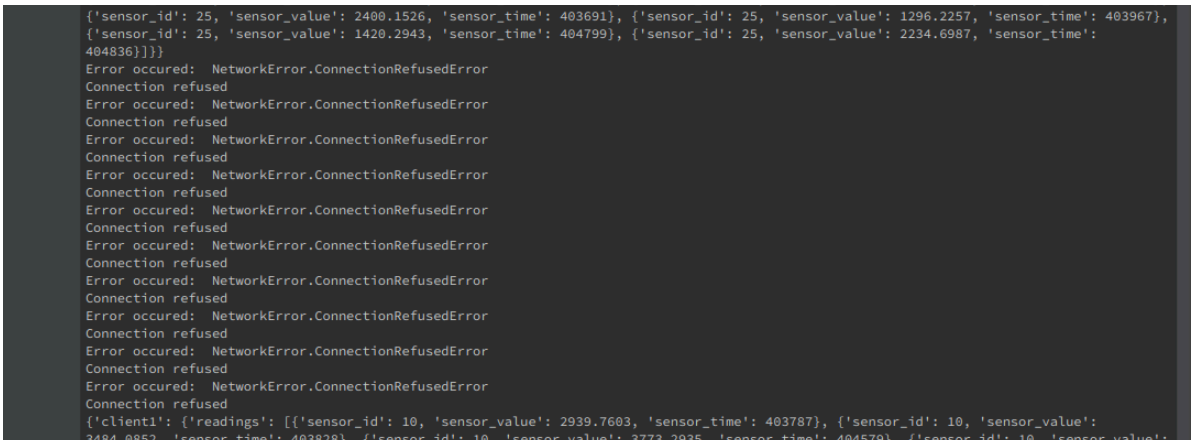


Fig 5.19. Number of GET requests sent at 100ms before socket is ready (200ms downtime)

Below are the number of rejected GET requests at 1000ms while the sockets were unchanged (2000ms). Two rejected requests can be expected here.

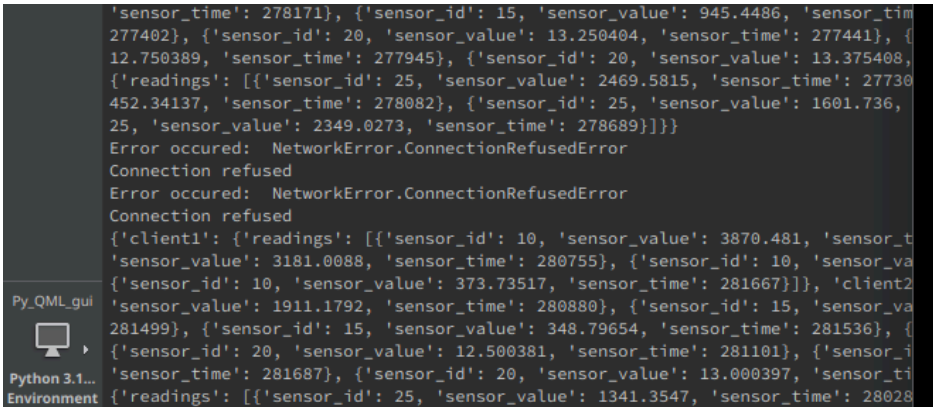


Fig 5.20. Number of rejected GET requests - once per 1000ms (2000ms downtime)

This is in contrast to the higher downtime delays, where there were far fewer rejected requests relatively speaking. This means that the increased traffic could have caused as much as 900ms of delay.

The resulting total energy after the socket was run for a minute is shown below. The server would stop transmitting some time after a minute at this sampling/socket delay pair.

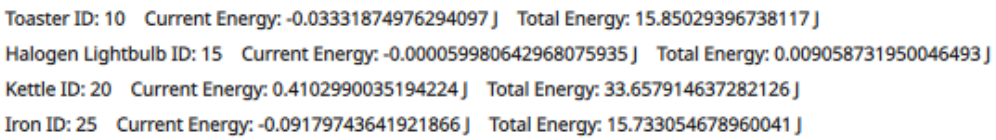


Figure 5.21. After an interval of 200ms downtime.

From the raw JSON sensor_time: 235205 - 237464 = 2259ms per interval

33/(60/2.259) = 1.269J per 2.259s interval

E = 3^2/24.05 * t = 0.374W*2.259 = 0.844 J

This, combined with the earlier results, shows that the total power calculation overestimates the power at the 200ms downtime, 100ms polling interval, rather than underestimating it. This seems to show that the packet loss and retransmission delays have a significant impact on the shown total energy.

When testing this section, an error was found where the ms received were not converted to seconds before being integrated. This caused ridiculously high energy readings which can be seen in earlier screenshots.

```
times1 = [item["sensor_time"]/1000.0 for item in data1]
```

Listing 5.7. Incorrect time fix

Another change made was the adding of labels based on the device the client was simulating. This was not done dynamically and the label is purely superficial for the sake of reading and understanding the data better.

```
text: "Toaster ID: " + data1.clientid + ... data1.totalenergy+ " J"
```

Listing 5.8. Replaced “Client” with device type

5.7 Penetration Testing

Once connected, the IP can be found by running:

```
ip neighbour
```

Which returns:

```
192.168.2.1 dev wlan0 lladdr f2:f5:bd:8f:f8:b0 REACHABLE
```

With this IP address, the port can be found by using the network map utility, an open source port scanner. The command below, with the above IP address and using the TCP connect flag returns:

```
nmap -sT 192.168.2.1 --system-dns
```

```
[tiarnan@ruylopez FinalYearProject]$ nmap -sT 192.168.2.1 --system-dns
Starting Nmap 7.95 ( https://nmap.org ) at 2025-03-30 07:44 IST
Nmap scan report for _gateway (192.168.2.1)
Host is up (0.089s latency).
Not shown: 999 closed tcp ports (conn-refused)
PORT     STATE SERVICE
8000/tcp open  http-alt

Nmap done: 1 IP address (1 host up) scanned in 45.22 seconds
```

Figure 5.22. Open port at 8000 found after only 45 seconds.

From here, one can simply type 192.168.2.1:8000 into the search bar which shows:

```
{"client1":{"readings":[{"sensor_id":10,"sensor_value":317.0683,"sensor_time":569522},
{"sensor_id":10,"sensor_value":2057.9749,"sensor_time":569559},
{"sensor_id":10,"sensor_value":189.3644,"sensor_time":570510},
{"sensor_id":10,"sensor_value":1309.1205,"sensor_time":571256},
{"sensor_id":10,"sensor_value":375.67697,"sensor_time":571296},
{"sensor_id":10,"sensor_value":162.06592,"sensor_time":572858},
{"sensor_id":10,"sensor_value":1561.6929,"sensor_time":604601},
{"sensor_id":10,"sensor_value":3042.36,"sensor_time":604637}]},"client2":{"readings":
[{"sensor_id":15,"sensor_value":1659.3994,"sensor_time":566866},
{"sensor_id":15,"sensor_value":1498.9054,"sensor_time":566906},
{"sensor_id":15,"sensor_value":1320.8949,"sensor_time":568106},
{"sensor_id":15,"sensor_value":2337.967,"sensor_time":569395},
{"sensor_id":15,"sensor_value":8.483811,"sensor_time":569431},
{"sensor_id":15,"sensor_value":1335.9827,"sensor_time":571172},
{"sensor_id":15,"sensor_value":1140.1525,"sensor_time":572811},
{"sensor_id":15,"sensor_value":3505.9143,"sensor_time":572848}]},"client3":{"readings":
[{"sensor_id":20,"sensor_value":182.0,"sensor_time":569682},
{"sensor_id":20,"sensor_value":180.0,"sensor_time":569925},
{"sensor_id":20,"sensor_value":178.0,"sensor_time":571383},
{"sensor_id":20,"sensor_value":180.0,"sensor_time":571423},
{"sensor_id":20,"sensor_value":180.0,"sensor_time":571967},
{"sensor_id":20,"sensor_value":178.0,"sensor_time":572684},
{"sensor_id":20,"sensor_value":178.0,"sensor_time":572720},
{"sensor_id":20,"sensor_value":180.0,"sensor_time":573401}]},"client4":{"readings":
[{"sensor_id":25,"sensor_value":1025.2784,"sensor_time":564667},
{"sensor_id":25,"sensor_value":2510.161,"sensor_time":566236},
{"sensor_id":25,"sensor_value":564.2441,"sensor_time":568585},
{"sensor_id":25,"sensor_value":3126.5586,"sensor_time":568621},
{"sensor_id":25,"sensor_value":1711.6227,"sensor_time":569695},
{"sensor_id":25,"sensor_value":37.07232,"sensor_time":571656},
{"sensor_id":25,"sensor_value":3932.6047,"sensor_time":571691},
{"sensor_id":25,"sensor_value":583.86975,"sensor_time":572870}]}}
```


Figure 5.23. Raw JSON data written to port 8000.

Given this, an attacker could write a simple python file which, like the actual GUI implementation, queries the port and captures the resulting JSON.

It should be noted, however, that not much conclusive data is given here. The sensor_id doesn't reveal the nature of the device, nor does the sensor_value give any indication as to the purpose or scale of what is being measured.

Chapter 6 – Ethics

There are many ethical considerations that come into play in this project. The first consideration is related to the memory safety of Rust. Rust provides innate protection against buffer overflow attacks with its strict memory safety features. These attacks work by overloading a given buffer with data which can lead to the execution of code or unexpected behaviour. However, through the use of the unsafe feature, one accepts that Rust is no longer providing the same guarantees as normal code which can lead to such attacks. In the code, there are indeed instances where the unsafe label is used. If used in production, this can be ameliorated by minimising these occurrences.

```
unsafe { core::str::from_utf8_unchecked(&buffer[..(pos + len)]) };
```
Listing 6.1. Unsafe code example from Chapter 4 which takes an input

Another ethical consideration is the wireless communication of the data. The network provided by the server ESP is not password protected which means that, in theory, an attacker could monitor the activity. This could be done by connecting to the network and scanning the network with a tool such as nmap to find the open port. Then, the attacker could simply parse the incoming JSON to extract the energy usage information. In a smart home, this could be used to plan a burglary by analysing the data for times when the homeowner is away. A more in depth analysis and example of this can be seen in Chapter 5.7. For this reason, WPA2 Personal should be added before release.

There are also legal and safety issues worth considering. There was a case where a man in Cork was fined four thousand euros for performing electrical works on his home without being an electrician[57]. Since installing this type of device can be dangerous, it should be made clear on release that only a registered electrician should be installing this device on the mains or any other appliance. When working with alternating current and high voltages/currents (as are found in mains), there is a risk of fibrillation which could result in death. This should be emphasised in the README.

While the code for this project will be open sourced, it should be made clear that the product is unfinished and not safe for deployment in its current state. As such, it will use the Apache-2.0 license which does not provide a warranty, but still allows people to use and

redistribute the work[58]. This ensures credit for the work is given properly without leaving contributors liable.

There is also a positive ethical effect worth considering. This energy monitor could be used to cut down on energy waste by monitoring activity and figuring out how much energy is being wasted. It is also possible that by monitoring the energy usage of a device, malfunctions can be detected or predicted. For example, if a device is using electricity even when in sleep mode, it could mean that a device is failing to enter low power mode.

Chapter 7 - Conclusions and Further Research

7.1 Conclusions

The goal of this project was to demonstrate the effectiveness of Rust by producing a Rust-powered multi-channel energy monitor for smart home applications. The title requirements were fulfilled - the entire client-server backend is indeed using rust, and nicely highlights some of the useful features such as the Result type, memory safety and type safety. It also shows how to break these rules when necessary, leveraging the unsafe{} scope with care.

The project also achieved multi-channel functionality. A maximum of four clients can be connected at the same time asynchronously. While only one of these clients had real voltage data, the code can easily be adapted to support multiple I2C interfaces - each with their own client. The I2C also supports multiple channels, meaning that the current and voltage could both be measured on the same client device without much hassle.

Care was taken to account for its usage in a smart-home environment. The ADC sampling frequency was selected with an AC frequency of 50Hz, and the clients are protected from surges from the mains by the ADC. In regards to the “energy monitoring”, energy was successfully calculated on the front end, although the accuracy can be disputed. This was done for multiple different household appliances with nearly purely resistive loads assumed.

This work adds a much needed up-to-date implementation of Embassy using Rust in the current version. There are other projects using Embassy on ESPs but if the project is as much as a few months old the dependency structure could be completely different. This can be seen here: REF. This changes not only the Cargo.toml file but also the usage of any of the crates which were restructured. For example, this project is released around the same time as the esp-hal 1.0.0 beta announcement. This crate was used extensively with version 0.23.1 (current stable version). This shows that the no_std ecosystem is ever evolving and only in its infancy. As such, this project will be open-sourced so it can be used as a template by others when starting their own Embassy projects.

7.2 Reflections and Further Research

While the goals of the project are satisfied, there are many opportunities for further research and refinement.

7.2.1 Hardware

In terms of hardware choices, I think selecting a RISC-V cored device was the right choice. It promotes the use of open source architectures and can lead to cheaper electronics (ARM licenses can be as much as one million euros plus a commission on each device!). I also got to use an opportunity to showcase some of the knowledge gained from working on a RISC-V core at Infineon - the PMP (or MPU) was a focal point of some of my work (see Chapter 5.2).

On the other hand, more thought could go into the capabilities of each device. While the client devices were barely pushed, the server had an enormous load for a single core processor with only 400kB of SRAM. The client and server devices could be different, with perhaps a less powerful/cheaper client and a more powerful server device.

Time permitting, more focus could be put on the gathering of data. I intended to have a setup with a current transformer which could monitor actual readings, rather than simulated ones. On top of this, if both the current and voltage were measured (using two channels of the same ADC), the instantaneous power could be calculated for any device, not just devices with purely resistive loads and no reactance.

7.2.2 Software

A lot of compromises were made in regards to the software for the sake of time and complexity. For example, a spinlock/mutex could be implemented rather than using static mutables, which are not covered by Rust's memory safety. On top of this, more robust measures for keeping track of buffer pools could allow for better reliable reconnects. Also, further use of Rust’s Result error control could be leveraged to further improve robustness (pattern matching rather than simply unwrap()-ing the Result).

Zigbee was a better choice for a project like this. This wasn't chosen for complexity reasons, as the provided embassy libraries do not include a Zigbee one, meaning that a different library would need to be used which may cause compatibility issues. Zigbee also could provide better practical benefits such as efficient low-power control - as discussed in Chapter 3.

Within WiFi, UDP may have been a better option. The robust error correction used by TCP ended up slowing down the data transmission (Chapter 5). This type of socket is not necessarily appropriate, as, unlike with a file transfer, if packets are missing it's not necessarily a big deal.

Speaking of networking, something which was planned but never implemented was WPA2-Personal encryption. While this might induce an even higher load on the server, having some sort of encryption for a network is a must (as seen in Chapter 5.7). Furthermore, some kind of response could be implemented - for example triggering an LED on the ESP32 when a certain energy threshold was passed at a minimum. There's a lot that could be done with fully duplex communication from GUI to client.

7.2.3 Frontend

The UI was quite rushed as this was one of the final things implemented. Due to time constraints, there were no interactive or even aesthetic additions. This could be fleshed out by adding buttons, network statistics and statuses for each client to show their connection. There could also be dynamic choosing of the device being monitored. It was shown that multiple functions can be used to alter the voltage to the correct power, this could be implemented as a dropdown menu where the client ID and energy values are shown.

References

[1] "ESP32C3 Series Datasheet UltraLowPower SoC with RISCV SingleCore CPU Supporting IEEE 802.11b/g/n (2.4 GHz WiFi) and Bluetooth ® 5 (LE) Including." https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf, (14/11/2024). 31/03/2025

[2] "esp-rs docs," *Esp-rs.org*, 2025. https://docs.esp-rs.org/esp-hal/ (accessed Mar. 31, 2025).

[3] "esp_idf_hal - Rust," *Esp-rs.org*, 2025. https://docs.esp-rs.org/esp-idf-hal/esp_idf_hal/ (accessed Mar. 31, 2025).

[4] alldatasheet.com, "ADS1015 Datasheet(PDF)," *Alldatasheet.com*, 2025. https://www.alldatasheet.com/datasheet-pdf/pdf/292738/TI/ADS1015.html (accessed Mar. 31, 2025).

[5] Embassy, "Embassy," *Embassy*, Jun. 02, 2024. https://embassy.dev/ (accessed Mar. 31, 2025).

[6] T. Q. Company, "Embedded Software Development Tools | Cross Platform IDE | Qt Creator," *www.qt.io*. https://www.qt.io/product/development-tools (accessed Mar. 31, 2025).

[7] Mayank C, "Java vs Rust: How faster is machine code compared to interpreted code for JWT sign & verify? | Tech Tonic," *Medium*, Nov. 18, 2023. https://medium.com/deno-the-complete-reference/java-vs-rust-how-faster-is-machine-code-compared-to-interpreted-code-for-jwt-sign-verify-fa6aeeabff58 (accessed Mar. 31, 2025).

[8] T. Heartman, "Understanding the Rust borrow checker," *LogRocket Blog*, Mar. 25, 2024. https://blog.logrocket.com/introducing-rust-borrow-checker/ (accessed Mar. 31, 2025).

[9] "Memory safety," *www.chromium.org*. https://www.chromium.org/Home/chromium-security/memory-safety/ (accessed Mar. 31, 2025).

[10] "TOML: Tom's Obvious Minimal Language," *toml.io*. https://toml.io/en/ (accessed Mar. 31, 2025).

[11] “Introduction - The Cargo Book,” *doc.rust-lang.org*. https://doc.rust-lang.org/cargo/ (accessed Mar. 31, 2025).

[12] “Why Async? - Asynchronous Programming in Rust,” *Github.io*, 2025. https://rust-lang.github.io/async-book/01_getting_started/02_why_async.html (accessed Mar. 31, 2025).

[13] “The Future Trait - Asynchronous Programming in Rust,” *rust-lang.github.io*. https://rust-lang.github.io/async-book/02_execution/02_future.html (accessed Mar. 31, 2025).

[14] P. Oppermann, “Async/Await | Writing an OS in Rust,” *Phil-opp.com*, Mar. 27, 2020. https://os.phil-opp.com/async-await/ (accessed Mar. 31, 2025).

[15] “I2C Primer: What is I2C? (Part 1) | Analog Devices,” *Analog.com*, 2024. https://www.analog.com/en/resources/technical-articles/i2c-primer-what-is-i2c-part-1.html (accessed Mar. 31, 2025).

[17] “ads1x1x - Rust,” *Docs.rs*, 2025. https://docs.rs/ads1x1x/0.3.0/ads1x1x/ (accessed Mar. 31, 2025).

[18] “nb - Rust,” *Docs.rs*, 2025. https://docs.rs/nb/latest/nb/ (accessed Mar. 31, 2025).

[19] GeeksforGeeks, “TCP/IP Model,” *GeeksforGeeks*, Aug. 07, 2019. https://www.geeksforgeeks.org/tcp-ip-model/ (accessed Mar. 31, 2025).

[20] Noction, “ACK and NACK in Networking,” *Noction*, Apr. 08, 2024. https://www.noction.com/blog/ack-and-nack-in-networking (accessed Mar. 31, 2025).

[21] “embassy_net - Rust,” *Embassy.dev*, 2025. https://docs.embassy.dev/embassy-net/git/default/index.html (accessed Mar. 31, 2025).

[22] “ESP32 Wi-Fi Basics Getting Started | ESP32,” *Electronicwings.com*, 2019. https://www.electronicwings.com/esp32/esp32-wi-fi-basics-getting-started (accessed Mar. 31, 2025).

[23] esp-rs, “esp-hal/examples/src/bin/wifi_embassy_access_point_with_sta.rs at main · esp-rs/esp-hal,” *GitHub*, 2021. https://github.com/esp-rs/esp-hal/blob/main/examples/src/bin/wifi_embassy_access_point_with_sta.rs (accessed Mar. 31, 2025).

[24] *Http.dev*, 2023. https://http.dev/1.0 (accessed Mar. 31, 2025).

[25] "serde_json - Rust," *docs.rs*. https://docs.rs/serde_json/latest/serde_json/ (accessed Mar. 31, 2025).

[26] "PyQtGraph - Scientific Graphics and GUI Library for Python," *www.pyqtgraph.org*. https://www.pyqtgraph.org/ (accessed Mar. 31, 2025).

[27] M. Fitzpatrick, "Creating PyQt6 desktop applications with QtQuick/QML," *Python GUIs*, Apr. 07, 2021. https://www.pythonguis.com/tutorials/pyqt6-qml-qtquick-python-application/ (accessed Mar. 31, 2025).

[28] "What is the Best Language for Embedded Systems? | KO2 Recruitment," *KO2 Recruitment*, Jan. 31, 2024. https://www.ko2.co.uk/best-language-for-embedded-systems/ (accessed Mar. 31, 2025).

[29] [1]A. Obregon, "Java in Embedded Systems: Opportunities and Limitations," *Medium*, Dec. 18, 2023. https://medium.com/@AlexanderObregon/java-in-embedded-systems-opportunities-and-limitations-cd0128c9f2c2 (accessed Mar. 31, 2025).

[30] Plauska, A. Liutkevičius, and A. Janavičiūtė, "Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller," *Electronics*, vol. 12, no. 1, p. 143, Dec. 2022, doi: https://doi.org/10.3390/electronics12010143.

[31] H. Ayers *et al.*, "Tighten rust's belt: shrinking embedded Rust binaries," *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, Jun. 2022, doi: https://doi.org/10.1145/3519941.3535075.

[32] D. Seeman, "WiFi and Bluetooth: How Do They Compare and Differ?," *IoT For All*, May 17, 2022. https://www.iotforall.com/bluetooth-vs-wifi (accessed Mar. 31, 2025).

[33] EETimes, "Implementing ZigBee networks," *EE Times*, Oct. 04, 2006. https://www.eetimes.com/implementing-zigbee-networks/ (accessed Mar. 31, 2025).

[34] “Using the Core Library (no_std),” *Esp-rs.org*, 2025. https://docs.esp-rs.org/book/overview/using-the-core-library.html#when-you-might-want-to-use-the-core-library-no_std (accessed Mar. 31, 2025).

[35] “RTIC and Embassy - Real-Time Interrupt-driven Concurrency,” *Rtic.rs*, 2025. https://rtic.rs/dev/book/en/rtic_and_embassy.html (accessed Mar. 31, 2025).

[36] “Embassy Book,” *Embassy.dev*, 2023. https://embassy.dev/book/#_getting_started (accessed Mar. 31, 2025).

[37] P. C. L. updated P. G. W. Library, “Which Python GUI library should you use in 2023?,” *Python GUIs*, Jul. 24, 2022. https://www.pythonguis.com/faq/which-python-gui-library/ (accessed Mar. 31, 2025).

[38] esp-rs, “GitHub - esp-rs/rust-build: Installation tools and workflows for deploying/building Rust fork esp-rs/rust with Xtensa and RISC-V support,” *GitHub*, Jan. 10, 2025. https://github.com/esp-rs/rust-build (accessed Mar. 31, 2025).

[39] “esp_hal::i2c::master - Rust,” *Esp-rs.org*, 2025. https://docs.esp-rs.org/esp-hal/esp-hal/0.23.1/esp32s2/esp_hal/i2c/master/index.html (accessed Mar. 31, 2025).

[40] G. Wright, “What is the maximum transmission unit (MTU)?,” *SearchNetworking*. https://www.techtarget.com/searchnetworking/definition/maximum-transmission-unit (accessed Mar. 31, 2025).

[41] “IpEndpoint in embassy_net - Rust,” *Embassy.dev*, 2025. https://docs.embassy.dev/embassy-net/git/default/struct.IpEndpoint.html (accessed Mar. 31, 2025).

[42] esp-rs, “esp-hal/examples/src/bin at main · esp-rs/esp-hal,” *GitHub*, 2021. https://github.com/esp-rs/esp-hal/tree/main/examples/src/bin (accessed Mar. 31, 2025).

[43] O. Hiari, “Sharing Data Among Tasks in Rust Embassy: Synchronization Primitives,” *The Embedded Rustacean Blog*, Jan. 09, 2023. https://blog.theembeddedrustacean.com/sharing-data-among-tasks-in-rust-embassy-synchronization-primitives (accessed Mar. 31, 2025).

[44] “circular_buffer - Rust,” *Docs.rs*, 2025. https://docs.rs/circular-buffer/latest/circular_buffer/ (accessed Mar. 31, 2025).

[45] “Circular Buffer | Baeldung on Computer Science,” *Baeldung on Computer Science*, Jan. 25, 2023. Available: https://www.baeldung.com/cs/circular-buffer (accessed Mar. 31, 2025).

[46] “Online JSON Size Calculator Tool (In Bytes) | JavaInUse,” *Javainuse.com*, 2025. https://www.javainuse.com/bytesizejson (accessed Mar. 31, 2025).

[47] Shalahuddin Al-Ayyubbi, “QNetworkAccessManager Process Finished With Exit Code 139 (Interrupted by signal 11: SIGSEGV),” *Stack Overflow*, Jul. 16, 2019. https://stackoverflow.com/questions/57059997/qnetworkaccessmanager-process-finished-with-exit-code-139-interrupted-by-signal (accessed Mar. 31, 2025).

[48] “PySide6.QtCore.QTimer - Qt for Python,” *Doc.qt.io*, 2025. https://doc.qt.io/qtforpython-6/PySide6/QtCore/QTimer.html (accessed Mar. 31, 2025).

[49] Data Journal, “How to Parse JSON Data With Python (EASY) | Medium,” *Medium*, May 29, 2024. https://medium.com/@datajournal/how-to-parse-json-data-with-python-99069a405e2b (accessed Mar. 31, 2025).

[50] py-unicorn, “How to smooth a graph in pyqtgraph?,” *Stack Overflow*, Oct. 16, 2020. https://stackoverflow.com/questions/64394172/how-to-smooth-a-graph-in-pyqtgraph (accessed Mar. 31, 2025).

[51] J. L. L. updated P. Graphics and Plotting, “Plotting in PyQt - Using PyQtGraph to create interactive plots in your GUI apps,” *Python GUIs*, Jan. 15, 2024. https://www.pythonguis.com/tutorials/plotting-pyqtgraph/ (accessed Mar. 31, 2025).

[52] J. Hannagan, R. Woszczeiko, T. Langstaff, W. Shen, and J. Rodwell, “The Impact of Household Appliances and Devices: Consider Their Reactive Power and Power Factors,” *Sustainability*, vol. 15, no. 1, p. 158, Dec. 2022, doi: https://doi.org/10.3390/su15010158.

[53] “scipy.integrate.simpson — SciPy v1.13.1 Manual,” *Scipy.org*, 2024. https://docs.scipy.org/doc/scipy-1.13.1/reference/generated/scipy.integrate.simpson.html#id1 (accessed Mar. 31, 2025).

[54] Magikman008, “No connection between qml and python in pyside6,” *Stack Overflow*, Nov. 23, 2023. https://stackoverflow.com/questions/77539922/no-connection-between-qml-and-python-in-pyside6 (accessed Mar. 31, 2025).

[55] “Your First QtQuick/QML Application - Qt for Python,” *Doc.qt.io*, 2025. https://doc.qt.io/qtforpython-6/tutorials/basictutorial/qml.html (accessed Mar. 31, 2025).

[56] Wassim Dhokar, “Analyzing Faults and Exceptions in RISC-V Processors: Techniques and Approaches,” *Medium*, Jul. 18, 2024. https://medium.com/@wassimdhokkar/techniques-to-use-to-analyze-software-faults-exceptions-on-riscv-processors-1d7a14fe494c (accessed Mar. 31, 2025).

[57] Safe Electric, “Cork Man Given Criminal Conviction and Fined €4,000 for Illegally Undertaking Electrical Works” *safeelectric.ie*, 2019. https://safeelectric.ie/cork-man-given-criminal-conviction-and-fined-e4000-for-illegally-undertaking-electrical-works/ (accessed Mar. 31, 2025).

[58] Apache, “APACHE LICENSE, VERSION 2.0,” *Apache.org*, 2019. https://www.apache.org/licenses/LICENSE-2.0 (accessed Mar. 31, 2025).

Appendix 1: GitHub Repository

Source code: https://github.com/Tiarban/rustmonitor