University of Colorado - Boulder

ECEN 4840

Independent Study | Summer 2024

Final Report

Software-Defined Instruments

ESP32-S3 ADC Characteristics

Sam Walker

Eric Bogatin

Friday, August 16, 2024

I would like to thank Eric Bogatin for being such an amazing mentor throughout the entire summer despite his busy schedule. I would also like to thank Sam McDiarmid-Sterling and Jonah Yunes for their help throughout this project.

College of Engineering & Applied Science
UNIVERSITY OF COLORADO BOULDER

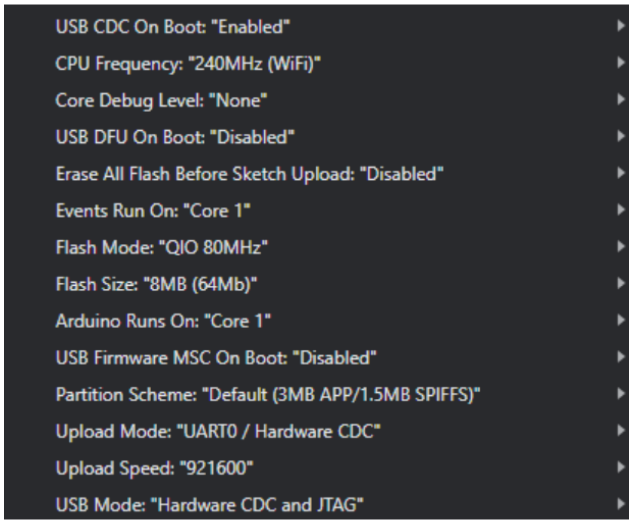
ESP32

The device being used for adc characterization tests outlined in this report is the **Adafruit QT Py ESP32-S3 No PSRAM** or simply QT Py/ESP32 for short. The QT Py requires some specific setup and maintenence to ensure proper working order. We used python scripts as our foundation for testing and analysis. Because of this it was necessary to create a communication protocol over which the ESP32 scripts could communicate with the python scripts.

Setup

To successfully upload code to the ESP32 there are some necessary steps to be taken.

1) Add the board to the list of additional boards available. URL can be found on Adafruit's website. Install the board support package once the URL is added. Here's a step by step guide by espressif: EspressIf Guide
2) When uploading code to the ESP32S3 certain configuration settings must be chosen. These settings under the tools header should match the following for code upload to succeed each time you attempt it.



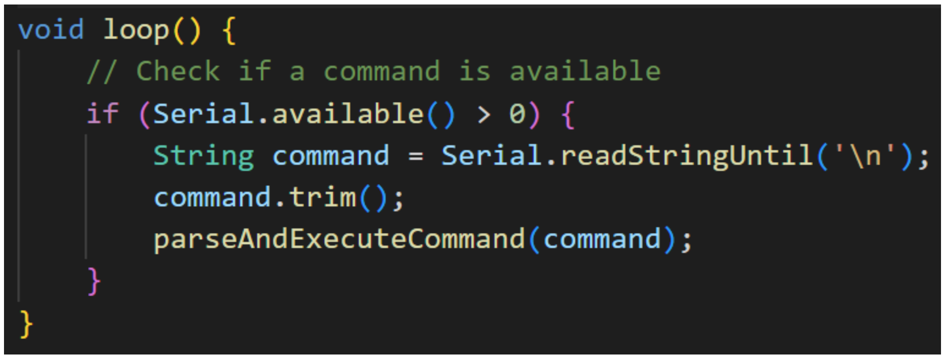
```
USB CDC On Boot: "Enabled"
CPU Frequency: "240MHz (WiFi)"
Core Debug Level: "None"
USB DFU On Boot: "Disabled"
Erase All Flash Before Sketch Upload: "Disabled"
Events Run On: "Core 1"
Flash Mode: "QIO 80MHz"
Flash Size: "8MB (64Mb)"
Arduino Runs On: "Core 1"
USB Firmware MSC On Boot: "Disabled"
Partition Scheme: "Default (3MB APP/1.5MB SPIFFS)"
Upload Mode: "UART0 / Hardware CDC"
Upload Speed: "921600"
USB Mode: "Hardware CDC and JTAG"
```

Figure 1: ESP32 Config

3) To upload code, hold down the 'boot' button on the ESP32 and while holding 'boot' down press 'reset', then release both. This should put the device into programming mode. Now ensure the COM port under tools->port is selected to the ESP32.
4) Now to open the serial monitor (ctrl+shift+m) and receive data it may be necessary to press 'reset' on the QT Py and reselect a new COM port.

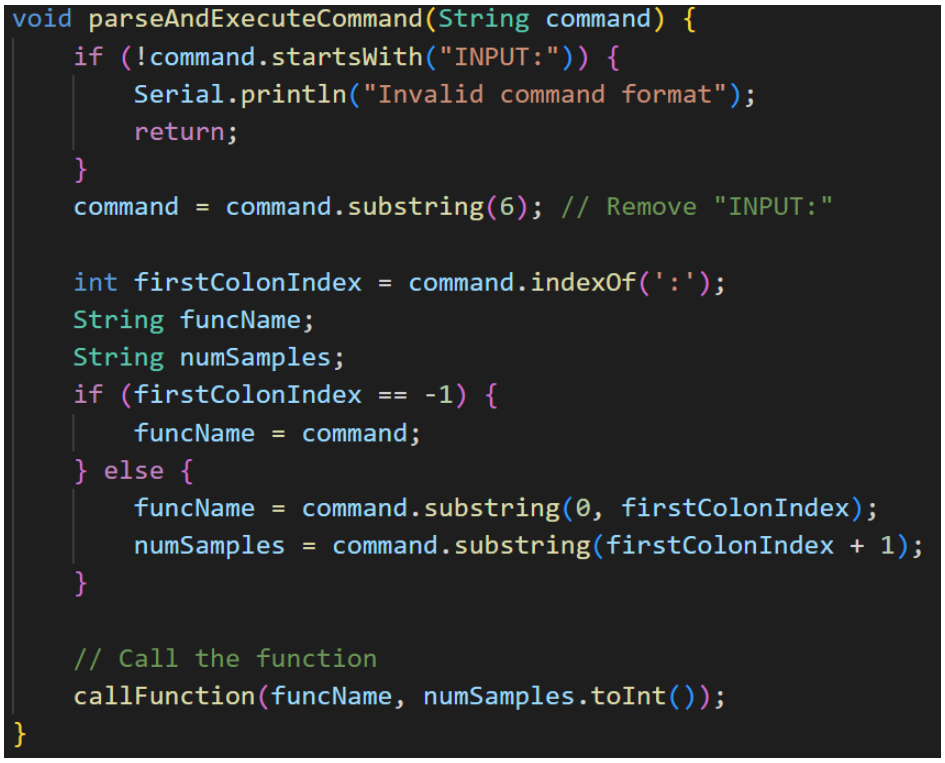
Custom Communication Protocol

In order to communicate between our python and esp32 scripts it was necessary to create a protocol of which messages could be sent. Essentially the format involved sending strings through the serial line in this format: INPUT:FUNCTION:PARAMETER from python to the esp32 and OUTPUT:VALUE from the esp32 to the python script.



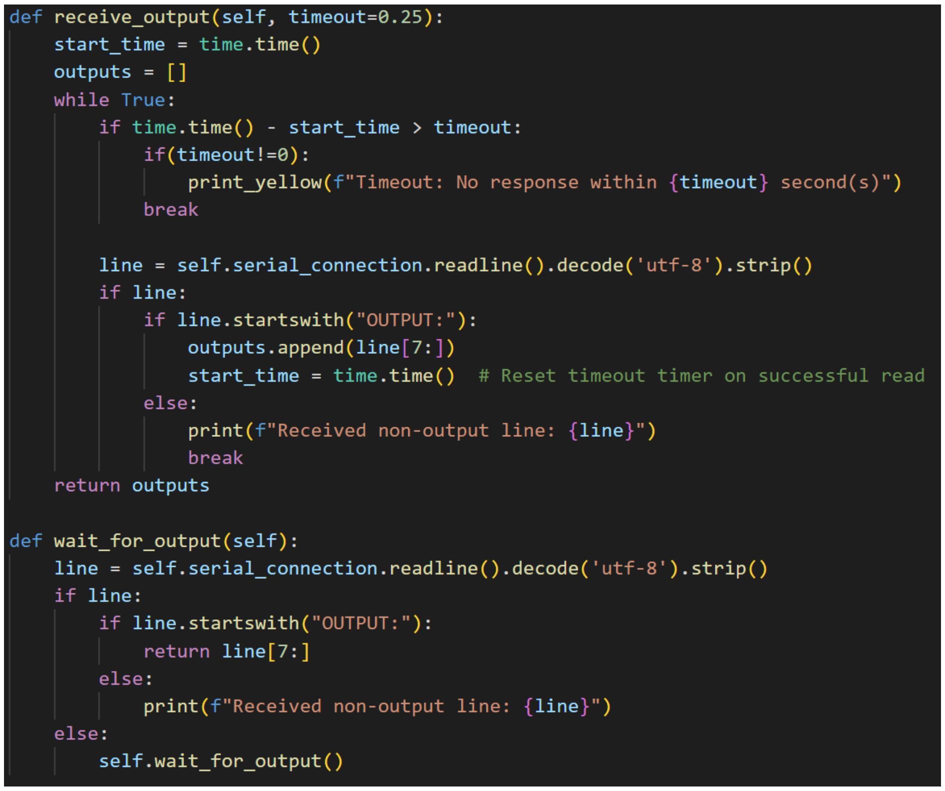
```
void loop() {
    // Check if a command is available
    if (Serial.available() > 0) {
        String command = Serial.readStringUntil('\n');
        command.trim();
        parseAndExecuteCommand(command);
    }
}
```

Figure 2: ESP32 Loop Function - This function was always waiting for commands to come into the esp32, unless it was currently running a previous command.



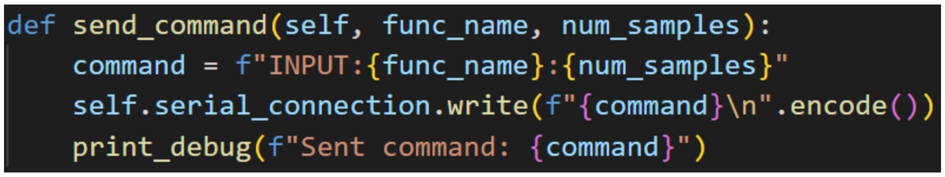
```
void parseAndExecuteCommand(String command) {
    if (!command.startsWith("INPUT:")) {
        Serial.println("Invalid command format");
        return;
    }
    command = command.substring(6); // Remove "INPUT:"

    int firstColonIndex = command.indexOf(':');
    String funcName;
    String numSamples;
    if (firstColonIndex == -1) {
        funcName = command;
    } else {
        funcName = command.substring(0, firstColonIndex);
        numSamples = command.substring(firstColonIndex + 1);
    }

    // Call the function
    callFunction(funcName, numSamples.toInt());
}
```

Figure 3: ESP32 Command Parsing - This function would interpret the INPUT:FUNCTION:PARAMETER and properly call the desired function with the desired parameter.



```python
def receive_output(self, timeout=0.25):
    start_time = time.time()
    outputs = []
    while True:
        if time.time() - start_time > timeout:
            if(timeout!=0):
                print_yellow(f"Timeout: No response within {timeout} second(s)")
            break

        line = self.serial_connection.readline().decode('utf-8').strip()
        if line:
            if line.startswith("OUTPUT:"):
                outputs.append(line[7:])
                start_time = time.time()  # Reset timeout timer on successful read
            else:
                print(f"Received non-output line: {line}")
                break
    return outputs

def wait_for_output(self):
    line = self.serial_connection.readline().decode('utf-8').strip()
    if line:
        if line.startswith("OUTPUT:"):
            return line[7:]
        else:
            print(f"Received non-output line: {line}")
    else:
        self.wait_for_output()
```

Figure 4: Python Output - These were the two functions used for receiving commands. The receive output function continuously receives an output until a certain 'timeout' amount of time passes in which the output stream has ended. wait for output makes use of recursion to continuously check for output making sure it catches just one output signal.

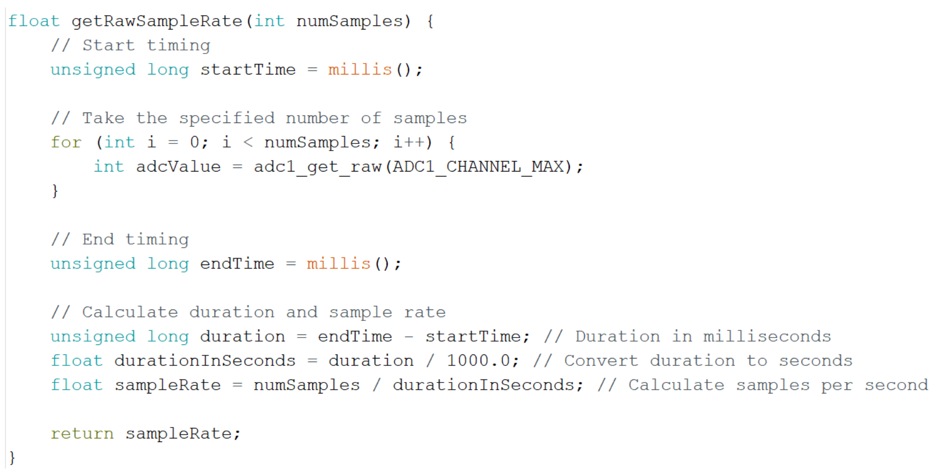


```python
def send_command(self, func_name, num_samples):
    command = f"INPUT:{func_name}:{num_samples}"
    self.serial_connection.write(f"{command}\n".encode())
    print_debug(f"Sent command: {command}")
```

Figure 5: Python Send - This function simply formats the function name and parameter as an input string to be sent to the esp32.

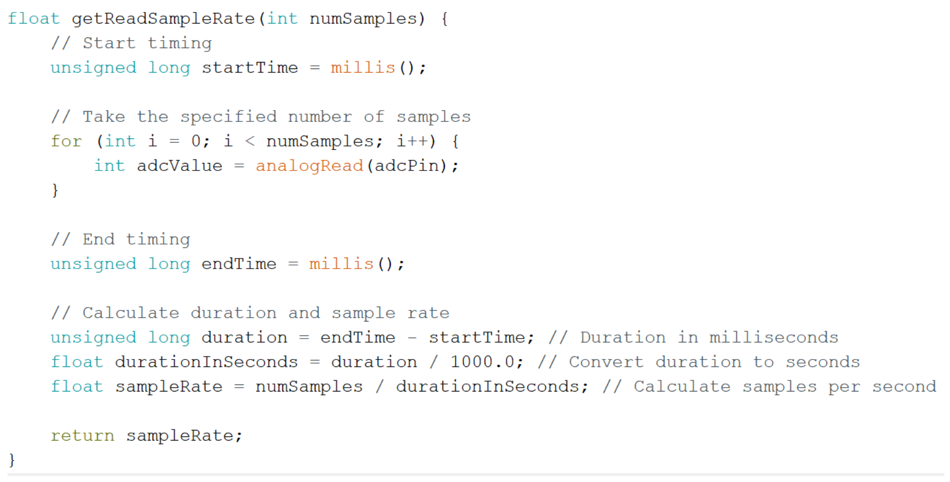
Hardware Testing

There was 3 main hardware tests using just ESP32 and not python. We wanted to confirm our streaming rate and sample rate of the adc and esp32. To do this we created seperate specific scripts that strictly test the sample and streaming rate. The results are more thoroughly described and shown later in the 'Results' section.



```
float getRawSampleRate(int numSamples) {
    // Start timing
    unsigned long startTime = millis();

    // Take the specified number of samples
    for (int i = 0; i < numSamples; i++) {
        int adcValue = adc1_get_raw(ADC1_CHANNEL_MAX);
    }

    // End timing
    unsigned long endTime = millis();

    // Calculate duration and sample rate
    unsigned long duration = endTime - startTime; // Duration in milliseconds
    float durationInSeconds = duration / 1000.0; // Convert duration to seconds
    float sampleRate = numSamples / durationInSeconds; // Calculate samples per second

    return sampleRate;
}
```

Figure 6: Raw Sample Rate



```
float getReadSampleRate(int numSamples) {
    // Start timing
    unsigned long startTime = millis();

    // Take the specified number of samples
    for (int i = 0; i < numSamples; i++) {
        int adcValue = analogRead(adcPin);
    }

    // End timing
    unsigned long endTime = millis();

    // Calculate duration and sample rate
    unsigned long duration = endTime - startTime; // Duration in milliseconds
    float durationInSeconds = duration / 1000.0; // Convert duration to seconds
    float sampleRate = numSamples / durationInSeconds; // Calculate samples per second

    return sampleRate;
}
```

Figure 7: Read Sample Rate

Both RawSampleRate and ReadSampleRate have the same test principle. They start a timer, load the analog value into the buffer and repeat that for a number of samples (1000). They differ in the library they use, RawSampleRate uses adc.h whereas ReadSampleRate uses Arduino.h. Because of small library differences RawSampleRate outperforms ReadSampleRate.

```
float getStreamingRate(int numSamples){
    unsigned long startTime = millis();

    for (int i = 0; i < numSamples; i++) {
        Serial.println("Test Sample");
        Serial.flush();  // Force sending of data
    }

    unsigned long endTime = millis();

    unsigned long duration = endTime - startTime;
    float durationInSeconds = duration / 1000.0;
    float sampleRate = numSamples / durationInSeconds;

    return sampleRate;
}
```

Figure 8: Streaming Rate

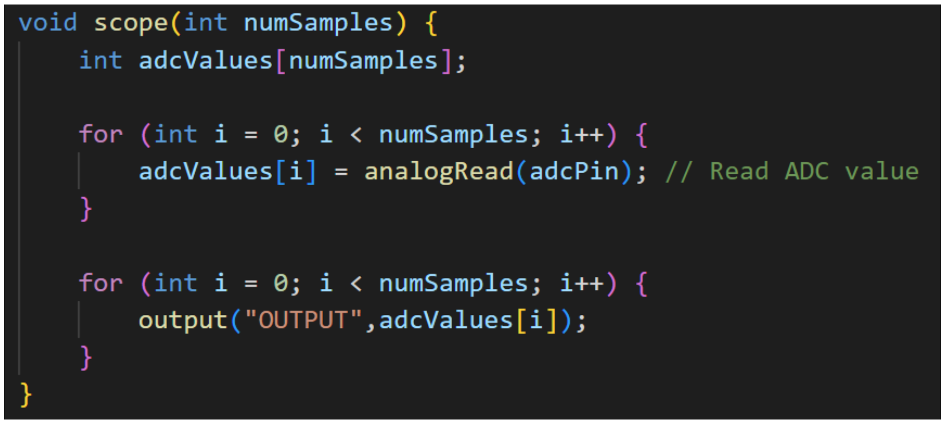
The streaming rate test is very similar to the sample rate tests, except that it prints a test message to the serial port. The interesting thing about the streaming test is that we determined using Serial.Begin(baudRate); has no impact to these results. That line of code is essentially ignored and the fastest baud rate is automatically chosen.

Software-Defined Instruments

Software-defined instruments are virtual instruments implemented through software to perform various measurements and analyses. These instruments are typically more flexible and cost-effective than traditional hardware instruments, as they allow for easy updates and customization. Below is a description of four key software-defined instruments utilized in this context: Scope, Strip Chart, RMS, and DC Calibration.

Scope

The software-defined oscilloscope (Scope) emulates the functions of a traditional hardware oscilloscope. It takes data as fast as possible, stores it in the buffer and then spits out the entire buffer over a serial connection to anything whether its a plot of voltage vs time or even just saving it to a csv. The flexibility of a software-defined scope also enables advanced functionalities such as automatic measurements, and the application of mathematical operations directly to the captured waveforms.

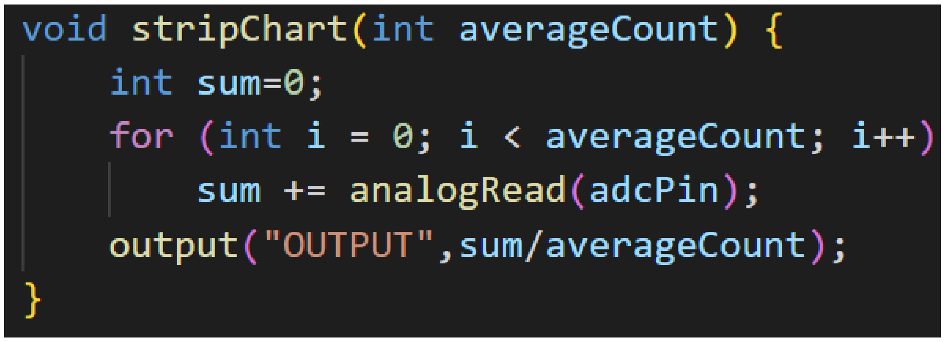


```
void scope(int numSamples) {
    int adcValues[numSamples];

    for (int i = 0; i < numSamples; i++) {
        adcValues[i] = analogRead(adcPin); // Read ADC value
    }

    for (int i = 0; i < numSamples; i++) {
        output("OUTPUT",adcValues[i]);
    }
}
```

Figure 9: Scope Code

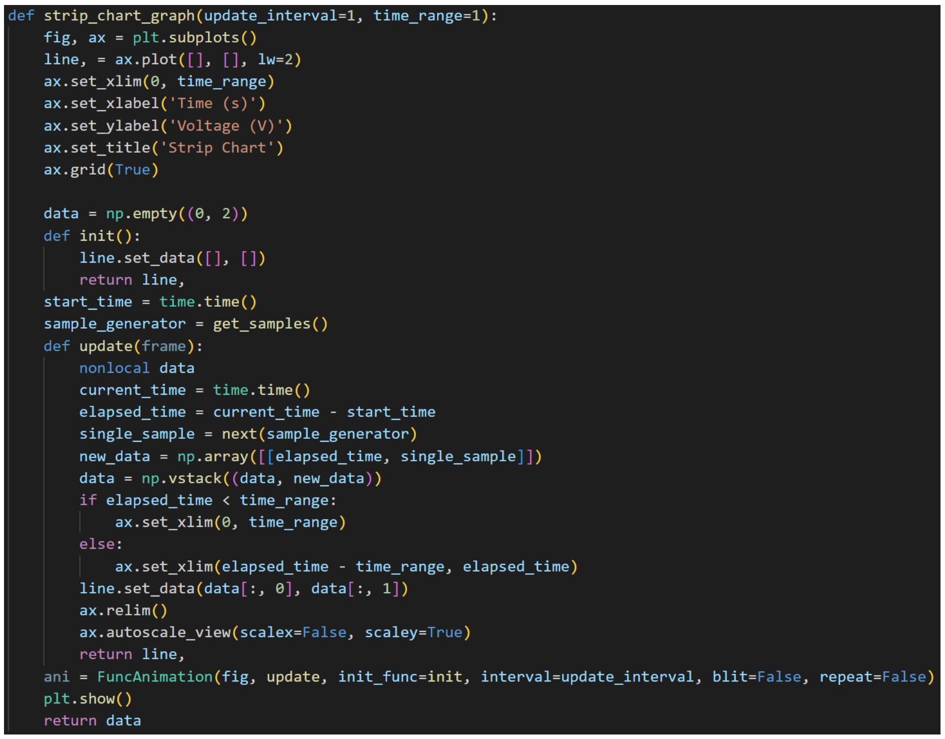
Strip Chart

A software-defined Strip Chart recorder is designed to record data over time and display it in a continuous, scrolling format. Unlike a traditional oscilloscope, which typically captures and displays data in short bursts, the Strip Chart is ideal for long-term monitoring of slowly changing signals. It is particularly useful for tracking trends, observing long-duration events, and logging data for subsequent analysis. The Strip Chart's software implementation allows for customization of the time axis, and data scaling, making it a versatile tool for both live monitoring and retrospective analysis. The strip chart function allows for averaging so that it shows the most accurate data possible because the streaming rate is slower than the sample rate.



```
void stripChart(int averageCount) {
    int sum=0;
    for (int i = 0; i < averageCount; i++)
        sum += analogRead(adcPin);
    output("OUTPUT",sum/averageCount);
}
```

Figure 10: Strip Chart Code



```python
def strip_chart_graph(update_interval=1, time_range=1):
    fig, ax = plt.subplots()
    line, = ax.plot([], [], lw=2)
    ax.set_xlim(0, time_range)
    ax.set_xlabel('Time (s)')
    ax.set_ylabel('Voltage (V)')
    ax.set_title('Strip Chart')
    ax.grid(True)

    data = np.empty((0, 2))
    def init():
        line.set_data([], [])
        return line,
    start_time = time.time()
    sample_generator = get_samples()
    def update(frame):
        nonlocal data
        current_time = time.time()
        elapsed_time = current_time - start_time
        single_sample = next(sample_generator)
        new_data = np.array([[elapsed_time, single_sample]])
        data = np.vstack((data, new_data))
        if elapsed_time < time_range:
            ax.set_xlim(0, time_range)
        else:
            ax.set_xlim(elapsed_time - time_range, elapsed_time)
        line.set_data(data[:, 0], data[:, 1])
        ax.relim()
        ax.autoscale_view(scalex=False, scaley=True)
        return line,
    ani = FuncAnimation(fig, update, init_func=init, interval=update_interval, blit=False, repeat=False)
    plt.show()
    return data
```

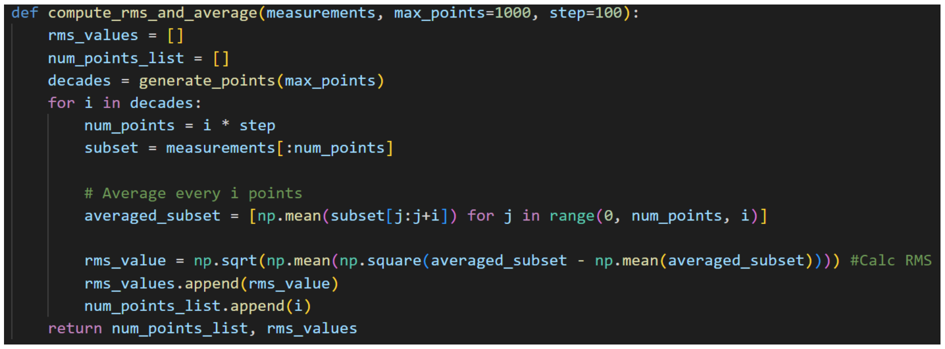
Figure 11: Strip Chart Graph Code

Graphing the strip chart is unique in that it is constantly updating and ’sliding’ as the time scale remains constant but the length of the data is always increasing. To do this it saves a list of the past ’timerange’ seconds data and also keeps the entire dataset. From there it continuously plots the newest data while maintaining a csv of the entire dataset. There are a lot of optimizations that we wanted to make to this function that I will talk more about in the ’What Next?’ section of the report.

RMS

The software-defined RMS (Root Mean Square) instrument is used for calculating the AC RMS value of a signal, which is a measure of its error. By averaging multiple samples into one, we are able to reduce the noise of the signal, this does not infinitely help because other factors such as polynomial drift take over the normal distributed noise that had been averaged out. This leads to one of our main questions during this research: At what point is averaging no longer helpful? I.E. What is the fastest rate that we can receive the most accurate information? Because averaging slows down the output of data, but makes it much more accurate.

Consider the situation where maxpoints is 1000 and step is 100. To calculate the RMS values we take max points value and always take the RMS of the size of step. For example, in this case we take the first 100 points of the 1000 generated points, and we average every 1 consecutive point (effectively not averaging) and then take the RMS. Next we take 200 points and average every 2 consecutive points and then calculate the RMS... all the way until we have used the entire dataset.



```python
def compute_rms_and_average(measurements, max_points=1000, step=100):
    rms_values = []
    num_points_list = []
    decades = generate_points(max_points)
    for i in decades:
        num_points = i * step
        subset = measurements[:num_points]

        # Average every i points
        averaged_subset = [np.mean(subset[j:j+i]) for j in range(0, num_points, i)]

        rms_value = np.sqrt(np.mean(np.square(averaged_subset - np.mean(averaged_subset)))) #Calc RMS
        rms_values.append(rms_value)
        num_points_list.append(i)
    return num_points_list, rms_values
```

Figure 12: RMS Code

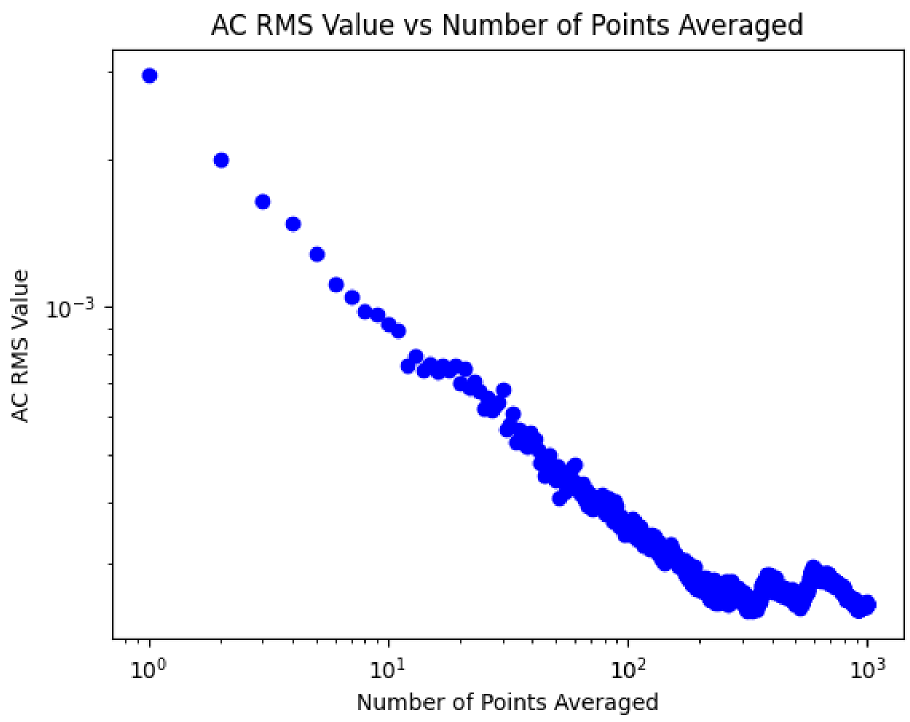


Figure 13: AC RMS - Noise

Our initial results on the ESP32 were very promising, as shown above, the RMS value or noise decreases as we average more points together. In fact, it decreases by a factor of the square root of the number of points averaged. For example, by increasing the number of points by a factor of 100 (from 1 to 100) we see that the RMS value decreases by a factor of 10 (from roughly 3m to .3m). This is as expected and is really cool to see. However, we were confused about the periodic and strange behavior at the end. To better understand this we synthesized data which is shown in the next graph.

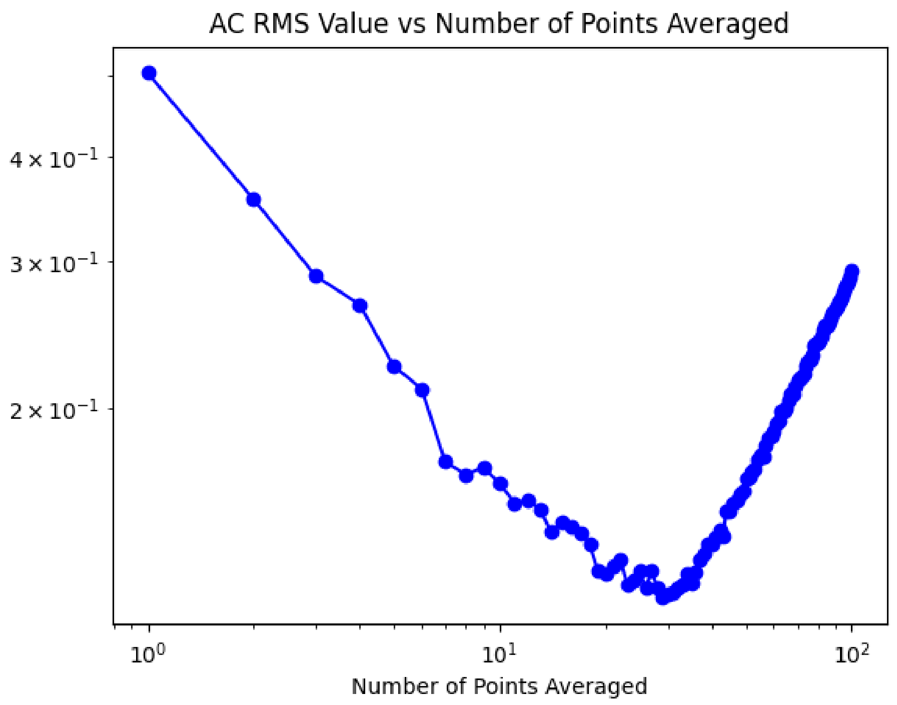


Figure 14: Synthesized Noise Results

For our synthesized dataset we introduced random noise as well as a small drift. This resulted in the data being centered around a linear curve. By using our RMS function on this data we got the graph above. This showed us that the strange periodic results we had got earlier were as expected because in this graph the drift takes over the rms plot and in our original plot on the esp32's data the drift also took over it just wasn't as nominal of a signal.

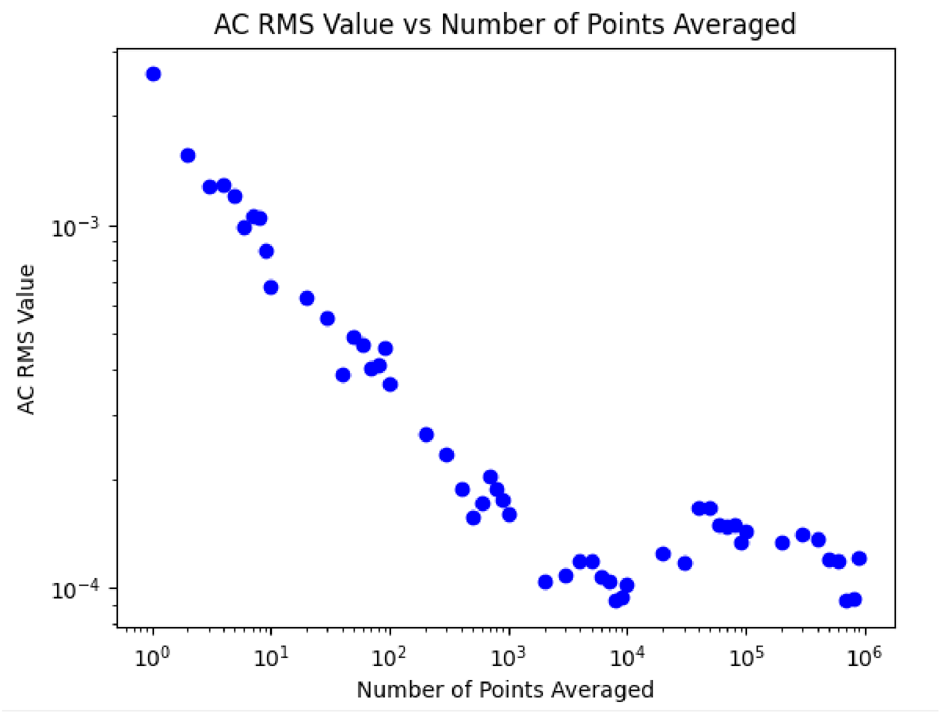


Figure 15: Upgraded Noise

We made one last upgrade to the RMS plotting in order to make the graph look cleaner which was showing only 10 points per decade and ran it with more data to produce the above graph. The end result showing us that around 1000 points is the optimal amount to average on the esp32.

DC Calibration

The software-defined DC Calibration instrument is crucial for ensuring the accuracy, precision and linearity of measurements in a system. It allows for precise calibration of DC signals, correcting any offsets or deviations from the expected values. The software implementation of DC Calibration enables automated and repeatable calibration procedures, enhancing the overall efficiency and consistency of the measurement process.

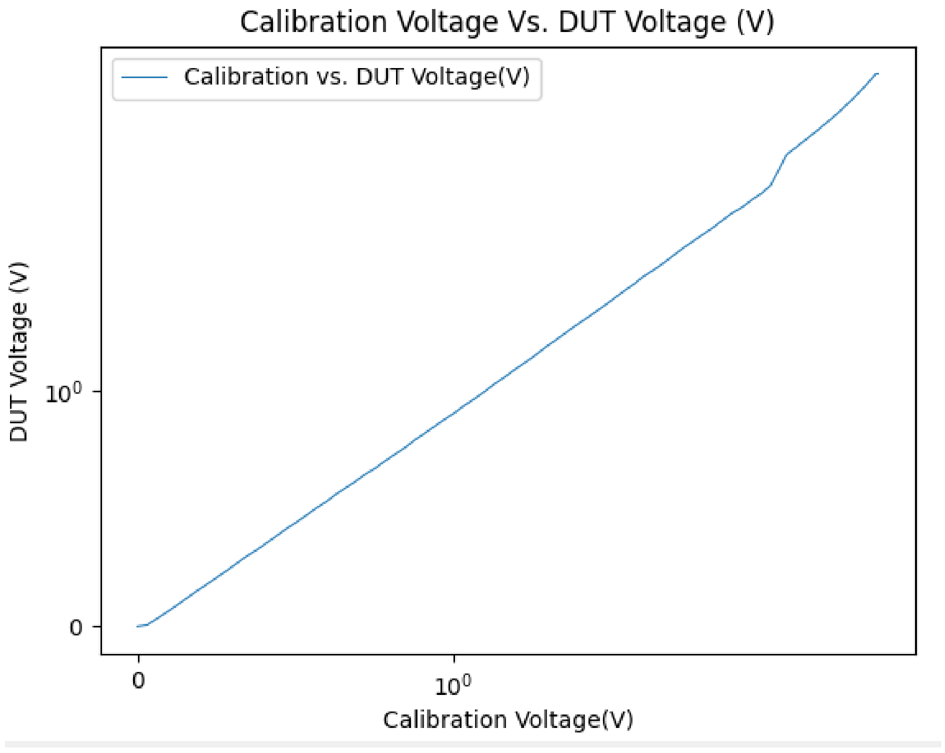


Figure 16: Linearity

Seeing this linearity curve was alarming because this shows that the ESP32 does not have an incredibly linear step and it even has a correction near 2.5 Volts. I will talk more about what could be causing this and possible solutions in the 'What Next?' section of the report.

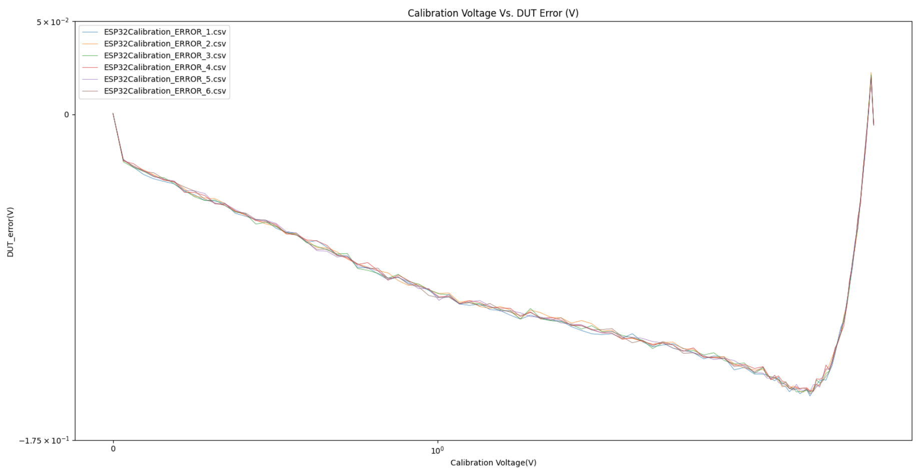


Figure 17: Error Graph with Multiple Curves

After running the linearity test on the same ESP32 6 times and calculating the error of each one we could see that this was a recurring problem and a problem of accuracy rather than precision.

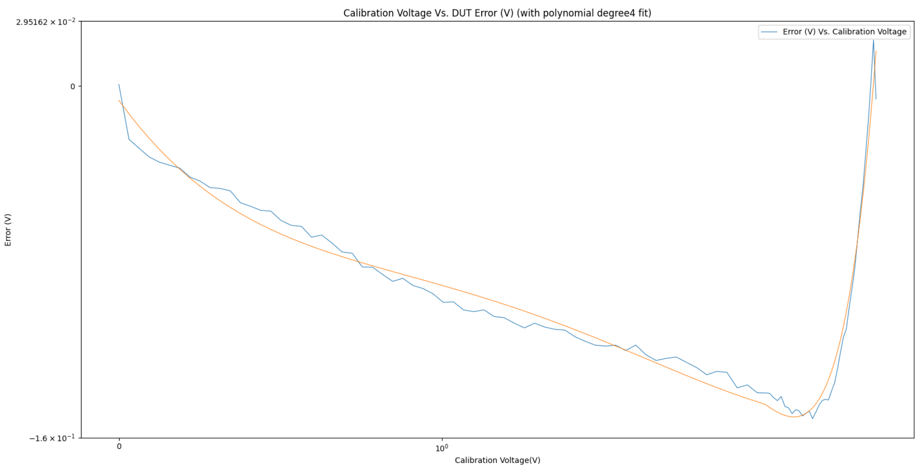


Figure 18: Polynomial Fit

One way that this calibration tool could be very useful is by fitting a polynomial curve to the error and using the inverse function to correct any measurements taken by the instrument. Above is an example of fitting a 4th degree polynomial to my error. However, I expect if my instrument did not have that strange correction it would require a much lower degree polynomial in order to approximate the error.

Important Figure Results

Figures of Merit Table

Figure of Merit	**Specified Value**	**Measured Value**	**Calculated Value**	**Units**
Bit Level	12	12	-	LSB
Internal Buffer	520	-	-	kB
Sample Rate	100	32-38	-	kSPS
Streaming Rate	-	17	0.886	kSPS
Linearity (DNL)	±4	-	-	LSB
Linearity (INL)	±8	-	-	LSB
Range (Atten0)	950	950	-	mV
Range (Atten1)	1250	1250	-	mV
Range (Atten2)	1750	1750	-	mV
Range (Atten3)	3100	3100	-	mV
LSB Voltage (Atten0)	-	-	0.231	mV
LSB Voltage (Atten1)	-	-	0.305	mV
LSB Voltage (Atten2)	-	-	0.427	mV
LSB Voltage (Atten3)	-	-	0.757	mV
Accuracy (Atten3)	-	-	5	%
Noise (Atten3)	-	2.96	-	mV
ENOB (Atten3)	-	10	-	LSB

Table 1: Figures of Merit

Bit Level

Refers to the resolution of the analog-to-digital converter (ADC), indicating the number of bits used to represent the analog input. A higher bit level allows for finer granularity and more precise digital representation of the analog signal. The datasheet specifies the bit level to be 12 meaning that the adc outputs values from 0-4095 because 2^{12} = 4096.

Internal Buffer

This is the amount of memory available within the system to temporarily store data before processing or transferring it, helping to manage the flow of data and avoid bottlenecks. The ESP32 datasheet references 520kB internal buffer. With some calculations we can determine how much data we can expect to be able to store.

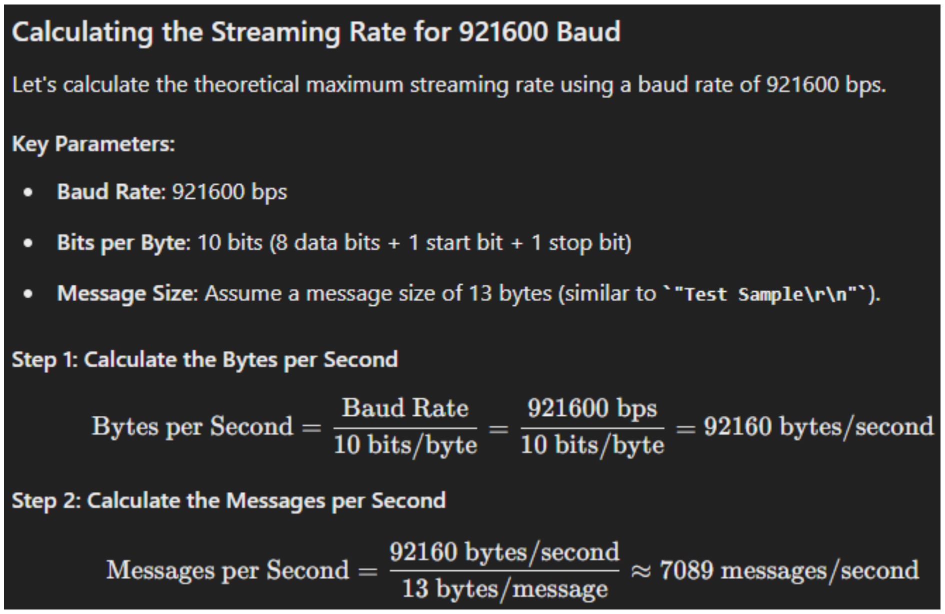
- 520 kB to bytes: There are 1024 bytes per kB so three is 532,480 bytes in 520kB
- Integers: 32 bit integers take 4 bytes each, resulting in a capacity of 133,120 integers.
- Floats: Floats are 64 bit or 8 bytes, resulting in a capacity of 66,560 floats.

Sample Rate

Indicates the number of samples taken per second by the ADC, determining how frequently the analog signal is converted to a digital value. A higher sample rate captures more detail of the analog signal. The datasheet mentions 100kSPS, however, I was never able to get a sample rate that high. Using the Arduino.h library along with analogRead(pin) results in around 32kSPS, whereas using the adc.h library along with ADC1getraw results in around 38kSPS.

Streaming Rate

Refers to the rate at which data can be continuously transferred or processed, often limited by the communication interface or the system's data handling capabilities. To test the streaming rate we can display a test message to the serial port and see how long it takes to display 1000 test samples. This method resulted in a streaming rate of 17kSPS. However, calculating the streaming rate yields:



Calculating the Streaming Rate for 921600 Baud

Let's calculate the theoretical maximum streaming rate using a baud rate of 921600 bps.

Key Parameters:

- **Baud Rate:** 921600 bps
- **Bits per Byte:** 10 bits (8 data bits + 1 start bit + 1 stop bit)
- **Message Size:** Assume a message size of 13 bytes (similar to `` `"Test Sample\r\n"` ``).

Step 1: Calculate the Bytes per Second

$$\text{Bytes per Second} = \frac{\text{Baud Rate}}{10\ \text{bits/byte}} = \frac{921600\ \text{bps}}{10\ \text{bits/byte}} = 92160\ \text{bytes/second}$$

Step 2: Calculate the Messages per Second

$$\text{Messages per Second} = \frac{92160\ \text{bytes/second}}{13\ \text{bytes/message}} \approx 7089\ \text{messages/second}$$

Figure 19: Streaming Rate Calculation

Linearity - Figure 16

Describes how closely the output of the ADC matches a straight line across its range. It includes Differential Nonlinearity (DNL) and Integral Nonlinearity (INL), which indicate deviations in step sizes and overall linearity, respectively.

Range

Defines the span of input voltages that the ADC can accurately convert into digital values. This can vary depending on attenuation settings or reference voltages. Despite being listed on the datasheet, the range can be tested very easily by changing the input signal until the result from reading the analog value reaches its max.

LSB Voltage

The voltage difference represented by the Least Significant Bit (LSB) of the ADC, calculated as the total input range divided by the number of discrete levels. This value can be calculated simply by following the equation below for atten3 where n is the bit level.

$$\frac{range}{2^n} = \frac{3.1}{2^12} = 0.757mV \tag{1}$$

Accuracy - Figure 17

Measures how close the ADC's output is to the true value of the input signal, taking into account all sources of error, including offset, gain, and linearity errors. To determine the accuracy we use our linearity error plot along with the equation below.

$$100 * \frac{error}{range} = 100 * \frac{0.151}{3.1} = 4.8 \approx 5\% \tag{2}$$

Noise - Figure 13

Refers to the random variations or disturbances in the ADC's output that are not related to the input signal, often quantified as the noise level in LSBs or volts. To find the noise of the ESP32's adc we calculated the ac rms (standard deviation) of 1000 points, this resulted in 2.96mV.

$$\sqrt{\frac{1}{N}\sum(V- < V >)^2} \tag{3}$$

(1) Where N is the number of samples in the rms calculation, V is the voltage of each sample and <V> is the mean of all voltages.

ENOB

Effective Number of Bits (ENOB) is a measure of the actual resolution of the ADC after accounting for all sources of error, including noise and distortion, reflecting the ADC's true performance. To determine ENOB we can use this formula $\log_2 \frac{range}{noise}$. For atten3 this results in an ENOB of 10.032, which we will truncate to 10.

What Next?

While a significant amount of progress was made over the summer, working only 10-15 hours per week on such an extensive project made it difficult to accomplish everything I had initially planned. As a result, there are numerous upgrades and next steps I have in mind if this project were to continue. However, with the onset of my senior year, I will not be able to continue working on this project in the upcoming fall. Below, I outline some of the key areas for potential future development.

Missing Figures of Merit Measurements

One of the primary goals moving forward would be to complete the confirmation of all the figures of merit listed in the figures of merit table. Ensuring that each of these metrics is accurately measured and validated would allow for comprehensive testing and comparison against the values specified in the datasheets. Additionally, it would be highly beneficial to develop standardized tests that could compare different ESP32 units with each other, as well as with other microcontroller platforms. Such comparisons would provide valuable insights into the performance consistency and variability of the ESP32, especially in relation to critical parameters like sample rate, noise levels, and accuracy.

Strip Chart Improvements

There are several enhancements that could significantly improve the functionality and performance of the strip chart. Firstly, implementing multi-threading for data acquisition and plotting would allow these tasks to run on separate CPU threads, greatly increasing the strip chart's ability to handle higher frequency data. Currently, the strip chart is limited to displaying low-frequency data in the 1-10 Hz range. By parallelizing these processes, it would be possible to achieve real-time plotting of higher frequency signals.

Additional improvements could include updating the CSV file dynamically as data is acquired, rather than waiting until the acquisition is complete. This would provide more immediate feedback and reduce data loss risks. Allowing the user to adjust the averaging window in real-time would also enable on-the-fly optimization of the data display, making it easier to identify trends or anomalies as they occur. Furthermore, dynamically adjusting the time scale based on the frequency of the incoming data would ensure that the graph remains relevant and informative, regardless of changes in the signal frequency.

RMS Improvements

One of the main challenges with the current RMS function is the need to gather all data points before calculating the RMS value. This approach delays the feedback that the function provides and prevents real-time visualization of the data. A significant improvement would be to enable real-time plotting of the RMS values as the data is gathered. This would allow the test to automatically populate the graph while running, providing immediate insights into the signal characteristics.

Another enhancement involves plotting the RMS values against time rather than just the number of points. This time-based approach could help to smooth out anomalies and provide a clearer representation of the signal's behavior over the duration of the test. By implementing these improvements, the RMS function would become a more powerful tool for analyzing dynamic signals in real time.

DC Calibration Next Steps

The next logical step for improving DC calibration would be to test additional ESP32 units. Unfortunately, I ran out of time to conduct extensive testing, but it would be fascinating to see if other ESP32 units exhibit the same anomalies as the one I tested. Understanding whether these anomalies are consistent across multiple units could provide valuable insights into the inherent characteristics of the ESP32's ADC.

Another avenue for improvement involves adjusting the ESP32's configuration settings during setup to enhance the accuracy of DC measurements. Exploring different configuration options could lead to better calibration results and more reliable data. The most significant upgrade to the DC calibration process would involve using the inverse of a polynomial fit to the error curve. By applying this correction algorithm, it would be possible to automatically correct for any systematic errors in the ESP32's ADC measurements, leading to more precise and accurate data acquisition.

Takeaways

Overall, I am very satisfied with the work I put into this project and I thought that it was very much worth my time and I learned a lot from doing it.

Working with ESP32

Throughout this project, I gained a deeper understanding of the intricacies involved in working with microcontrollers, particularly the ESP32. While I began with a solid foundation in Arduino, my knowledge of ADCs (Analog-to-Digital Converters) was minimal, and I had little experience with the ESP32. This project served as a hands-on learning opportunity, allowing me to explore the nuanced behavior of these devices.

I delved into the specifics of how ADCs function within the ESP32, learning about their resolution, accuracy, and the challenges associated with noise and sampling rates. Through experimentation and debugging, I gained insights into optimizing ADC performance, such as managing signal noise and understanding the trade-offs between speed and precision. This experience not only enhanced my technical skills but also provided me with a better understanding of how to leverage the ESP32's capabilities to improve system performance in real-world applications, preparing me to tackle more advanced microcontroller-based projects in the future.

Project Planning

This project presented a unique challenge in that it was highly free-flowing, which initially led me to underestimate the importance of structured project planning and management. As the project progressed, I realized the critical role that efficient planning plays in guiding a project's success. Without a clear plan, I often found myself sidetracked by tasks that, while interesting, were not the most beneficial to the project's overall objectives. I also spent time polishing aspects of the project that were still in the prototyping stage, which was not the best use of time or resources. For example, I spent over 10 hours working on my own serial plotter extension for vscode after looking on the extension site for 5 minutes. After those 10 hours, I spent 1 more minute on the extension store and found one that was much better than mine. Another distraction I had was with the communication protocol I created. I originally made it overly complicated and super modular accepting multiple parameters and multiple functions. This was all unnecessary for the scope of this project.

The value of weekly meetings became evident as they provided the necessary structure and accountability to keep the project on track. These meetings helped me prioritize tasks, set achievable goals, and make informed decisions about where to focus my efforts. Through this process, I developed a greater appreciation for the skills of project planning and time management. I learned the importance of setting clear milestones, regularly assessing progress, and being flexible enough to adapt the plan as new challenges or opportunities arose.

This experience has taught me that effective project planning is not just about creating a roadmap at the beginning but involves ongoing evaluation and adjustment. By the end of the project, I felt more confident in my ability to manage complex projects, ensuring that my efforts are aligned with the project's goals and timelines. This is a skill that I will carry forward into future projects, both in academic and professional settings.