DESIGN OF A LOW-COST,

MICROCONTROLLER-BASED

PHASOR MEASUREMENT UNIT

A Project

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Electrical & Computer Engineering

By

Joseph Gozum

2021

SIGNATURE PAGE

PROJECT:	DESIGN OF A LOW-COST MICROCONTROLLER-BASED PHASOR MEASUREMENT UNIT
AUTHOR:	Joseph Gozum
DATE SUBMITTED:	Spring 2021
	Department of Electrical & Computer Engineering

Dr. Sean Monemi
Project Committee Chair
Electrical & Computer Engineering

Dr. Thomas Ketseoglou
Project Committee Member
Electrical & Computer Engineering

Dr. Solomon Oldak
Project Committee Member
Electrical & Computer Engineering

ACKNOWLEDGEMENTS

I'd like to thank, Melissa Santos. You always believed in me when I never did and helped me to persevere through the tougher times of my project.

To my family for always encouraging me and being there to support me when I really needed it.

ABSTRACT

The electrical grid is due for an upgrade with many portions of the transmission lines and power plants going on for decades of service. These systems are not built with the increasing loads, changing energy sources, and global climate change. While it is not feasible to replace the aging components all at once, the grid can be upgraded to better minimize power failures and deficiencies of those older parts. PMUs are part of this upgrade solution, they provide real-time, two-way communication about power events along transmission and distribution lines by sending magnitude and phase of voltages and currents ~30 or more times per second. This helps to immediately identify issues and allows either manual or ideally automated systems to adjust. They also have the added benefit of time-tagging their data so events over a wide area can be simultaneously compared. While they play an important role, their associated development, installation, and maintenance costs impede even wider deployment. This project will implement a low-cost, microcontroller-based PMU. The microcontroller in this implementation is the based on an Arm Cortex-M4 processor with a floating-point unit and hardware accelerated Digital Signal Processing (DSP) instructions. This makes it very capable in the calculation of the fast Fourier transform that will be used to calculate the phasor forms of the sampled electrical signals. The phasors will then be time-tagged with a UTC timestamp coming a GPS receiver. The data will be then sent to a PC for visualization. The main emphasis of this project is to attempt to show that such a low-cost implementation that can meet the metrics in the IEEE standards governing PMUs is possible. However, due to inconsistent capturing of the input sinusoids the phase angle could not be calculated properly. Several outputs such as the frequency and rate of

change of frequency are derivatives of the phase angle and so they to would also be incorrect. The input signals were also sent through a breadboard which may have introduced more delay into the signals through extra capacitance or inductance. Future work would improve on the sampling and move away from breadboards and implement on a PCB where capacitance and inductance can be more tightly controlled.

TABLE OF CONTENTS

SIGNATURE PAGE ii

ACKNOWLEDGEMENTS iii

ABSTRACT iv

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER 1: INTRODUCTION 1

CHAPTER 2: THEORETICAL BACKGROUND 3

- Synchrophasor 3
- Phasor Representation: 3
- Discrete Fourier Transform 4
- Complex Magnitude 8
- Phase Angle 8
- Rate of Change of Frequency (ROCOF) 10

CHAPTER 3: IMPLEMENTATION 11

- Time Source 11
- Processing 17
- Data Acquisition 23
- Transmission 42
- Timings 45

CHAPTER 4: EVALUATION METRICS 47

- Total Vector Error (TVE) 47
- Frequency and Rate of Change of Frequency (ROCOF) Error 48
- Measurement Response and Delay Time 48
- Overshoot and Undershoot 49
- Reporting Latency 50
- Operational Errors 50

CHAPTER 5: METHODOLOGY 51

CHAPTER 6: RESULTS & CONCLUSION ... 54

CHAPTER 7: IMPROVEMENTS AND FUTURE WORK ... 57

REFERENCES ... 59

APPENDIX A: testGPS.ino ... 62

APPENDIX B: testADC.ino ... 64

APPENDIX C: testDFT.ino ... 66

APPENDIX D: testADC_v2.ino ... 67

APPENDIX E: testPWM.ino ... 69

APPENDIX F: PMU.ino ... 70

APPENDIX G: genSampleCLK.ino ... 77

LIST OF TABLES

Table 1: Atmega 328P Internal Registers for PWM Generation 26

Table 2: Input Signal Characterization 35

Table 3: ADC Requirements.. 36

Table 4: ADC Characterization 36

Table 5: External Capacitor Requirements 37

Table 6: External Resistor Requirements 37

Table 7: Primary Op-Amp Requirements 38

LIST OF FIGURES

Figure 1: 2-point Butterfly Operation [29] 6

Figure 2: 8-point Decimation-in-Time FFT [29] 7

Figure 3: System block diagram of PMU 11

Figure 4: Adafruit Ultimate GPS Breakout. Photographed by Adafruit [10] 12

Figure 5: Active GPS antenna. Photographed by Adafruit [12] 13

Figure 6: Signal information after ~1000 pulses from GPS receiver 14

Figure 7: Signal information at ~1000 pulses from GPS receiver 15

Figure 8: Example of NMEA Output Message 16

Figure 9: Adafruit Grand Central M4. Photographed by Adafruit [7] 17

Figure 10: Frequency chart of 256-point DFT 18

Figure 11: Bootstrap Analysis of 256-point DFT Median 19

Figure 12: Frequency chart of 128-point DFT 20

Figure 13: Percent difference of normalized, single-sided amplitudes of FFT outputs 21

Figure 14: Flowchart of Processing Block 22

Figure 15: AD9544 Functional Block Diagram. Image by Analog Devices. [13] 23

Figure 16: Custom AD9544 PCB 24

Figure 17: ADC Sampling Signal Generated by Atmega 328P 25

Figure 18: analogRead with no oversampling 29

Figure 19: analogRead with oversampling 30

Figure 20: Analog Interface LTSpice Simulation 32

Figure 21: Active Low-pass Filter Op-amp Protection 34

Figure 22: Voltage signal at different points of analog interface to ADC........................ 39

Figure 23: Op-amp input and output tracking, Oscilloscope Channel 1........................... 40

Figure 24: Op-amp input and output tracking, Oscilloscope Channel 2........................... 40

Figure 25: Adafruit Ethernet Feathering. Photographed by Adafruit [18]. 42

Figure 26: Flowchart for Transmission Block [19] 43

Figure 27: Histogram of Times to Send Full PMU Message... 44

Figure 28: Reduced Histogram of Length of Time to Send Full PMU Message 45

Figure 29: Microcontroller Utilization Per Sampling Period Using Mean Execution…...45

Figure 30: System-block Diagram of PMU system... 51

Figure 31: Actual System Under Test... 52

Figure 32: Photo of Actual Setup.. 53

Figure 33: Phase angles Graph 54

Figure 34: Phase Angle Channel 0 in Serial Monitor.. 55

Figure 35: Values captured on ADC CH0 56

CHAPTER 1: INTRODUCTION

The current electrical generation and distribution system in the United States is aging and is being forced beyond its original use case. The grid faces the challenges of interfacing with new and evolving loads, changing energy sources and demands, and climate change affecting the reliability and quality of power [1],[2]. Solar, wind, and hydro sources are being added to the grid to encourage renewable energy, but they typically generate energy far away from the consumers. The challenges of climate change are growing every year, the increase in the number of hurricanes, flooding, droughts, increasing temperatures, and more have catastrophic affects by shutting down energy generation or destroying the transmission infrastructure [3]. Several countries including the United States, Canada, China, and parts of Europe have been pushing for the Smart Grid to address these growing concerns.

The smart grid improves on the current grid by allowing two-way communication between the electrical service providers and the consumers, as well as sensing along transmission lines, and automated control of generation and distribution. With this two-way communication and more visualization of the grid, optimized generation and loading can occur and quicker restoration since problems are easily pinpointed [4]. Phasor Measurement Units (PMUs) are one of the most important components of this smart grid. They provide the required visibility by calculating the phasor-equivalents of the power signals and time-tagging them at high speed. The time-tagging allows data from over a wide-area to be combined to give a more accurate view of the overall system [5]. The high-speed sampling (typically ~30 samples per second) provides data on the real-time changes in the transmission lines. With this data, automated controls can be appropriately

activated. The challenge then faced by PMUs are their high initial costs for research and development and installation but also continuous cost for maintenance and repair [6]. Current PMUs are also typically closed source, so it is impossible for an electrical service provider to use them beyond their prebuilt purpose. The closed source philosophy also affects deeper understanding of the data received from those PMUs,

There are several projects that have attempted and successfully implemented an open source PMU in response, such projects include OpenPMU, GridTRAK PMU, and Dtu PMU [7]. OpenPMU is of considerable note as they have collected and published resources for developing a PMU. Initially, they produced a PMU using a closed source National Instruments Data Acquisition (DAQ) hardware but have gone on to implement a PMU based around the BeagleBone Black [8]. GridTRAK and Dtu PMU are also good examples of academically developed PMUs that ran on PCs with Dtu PMU even being introduced to the Danish electricity grid. However, while GridTRAK was a good design, the method it used to calculate phasors incurred a loss of point-on-wave and harmonic information [7]. Dtu also used LabVIEW proprietary software from National Instruments which is also a continual cost to pay for licensing.

The overall objective of this project is then to implement and verify a working PMU according to the IEEE standards governing PMU functionality to show a cheaper and open-source PMU based on a microcontroller can be developed. This project will also help to gather important design considerations and implementations so others may improve upon it.

CHAPTER 2: THEORETICAL BACKGROUND

Synchrophasor

Synchrophasors are time-tagged phasors in polar form representing the magnitude and phase angle of sinusoidal electrical power signals.

Rate of change of frequency (ROCOF), an output of PMUs, can be obtained as the derivative of the frequency with respect to time.

Phasor Representation:

A general sinusoidal waveform can be represented as follows in equation 1.1:

$$x(t) = A \cdot \cos(\omega t + \phi) \qquad (1.1)$$

A is the maximum amplitude of the signal with a radian frequency ω which is equal to $2\pi f_o$, f_o being the nominal frequency of the signal. Then ϕ is the phase shift of the signal. In a typical 3-phase power system each sinusoidal signal would have a phase shift difference of 120 degrees or $\frac{2\pi}{3}$ radians between each other as shown below

$$x_1(t) = A_1 \cdot \cos(\omega t) \qquad (1.2)$$

$$x_2(t) = A_2 \cdot \cos\left(\omega t + \frac{2\pi}{3}\right) \qquad (1.3)$$

$$x_3(t) = A_3 \cdot \cos\left(\omega t + \frac{4\pi}{3}\right) \qquad (1.4)$$

Using Euler's formula

$$e^{jx} = \cos(x) + j \cdot \sin(x) \qquad (1.5)$$

where j is the imaginary number. The general sinusoid defined earlier in equation 1.1 is shown to be related to this complex exponential function as the real portion when

expressed in trigonometric terms. In order to get to phasor representation of the sinusoid, it's converted into its complex exponential form and removing the radian frequency for now, the conversion is as follows:

$$A \cdot \cos(\omega t + \phi) = \Re e\{A \cdot e^{j(\omega t + \phi)}\} = \Re e\{Ae^{j\phi} \cdot e^{j\omega t}\} \quad (1.6)$$

$$A \cdot \cos(\omega t + \phi) = Ae^{j\phi} = A\angle\phi \quad (1.7)$$

Equation 1.7 shows the conversion from cosine to exponential and the subsequent representation conversion to the desired polar form. Creating a synchrophasor can now be easily done with equation 1.7, by associating a phase with a time signal in Coordinated Universal Time.

Discrete Fourier Transform

The Discrete Fourier Transform (DFT) is used to perform Fourier analysis on a finite-sequence of equally spaced samples of a function in time, converting a signal from the time domain to frequency domain. This allows one to see the component frequencies present in the time domain signal.

There exist real and complex versions of the DFT. In the complex version of the DFT, each time-domain datapoint is complex with a real and imaginary component. While the real DFT only deals with real valued data. For the sampled data that will be collected from this PMU, it would be sufficient to use the real DFT/FFT but for fullness this section will continue to consider the complex DFT/FFT in calculations.

Transforming from the time to frequency domain is known as the forward complex DFT which is seen in equation 1.8 and the inverse DFT in equation 1.9 [9], [10]

$$e^{-\frac{j2\pi}{N}} = W_N \; ; \; X_k = \sum_{n=0}^{N-1} x_n \cdot W_N^{nk} = \sum_{n=0}^{N-1} x_n \cdot \left[\cos\left(\frac{2\pi kn}{N}\right) - j\sin\left(\frac{2\pi kn}{N}\right)\right] \qquad (1.8)$$

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k \cdot e^{\frac{j2\pi kn}{N}} \qquad (1.9)$$

There are several things to note, one is the replacement of the exponential term for another variable W, this is commonly called the Twiddle factor. Also, this strict implementation of the DFT is computationally inefficient and does not scale well. Looking at equation 1.8 it can be seen for each index of the output DFT requires N number of multiplications. This occurs for each index of the output which is of the same length of the input data, so an input of length N would require N times N multiplications. For increasingly large sizes, this becomes infeasible to calculate. Algorithms that increase the calculation speed of the DFT are broadly known as Fast Fourier Transforms (FFTs).

Fast Fourier Transform

The most common method to efficiently calculate the DFT is through the Cooley-Tukey FFT algorithm, which splits the calculation of the DFT into its even and odd indexed input values [11], [12]. Equation 1.10 shows how the formal DFT in equation 1.8 can be rearranged into the calculation of the FFT

$$X[k] = \sum_{n=0}^{N-1} x(n) \cdot W_N^{nk} = \sum_{n=0}^{\frac{N}{2}-1} x(2n) \cdot W_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) \cdot W_N^{(2n+1)k}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} x(2n) \cdot W_N^{2nk} + W_N^{k} \cdot \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) \cdot W_N^{2nk} \qquad (1.10)$$

Before it was mentioned the variable W is known as the Twiddle factor. These are also known as nth roots of unity and exhibit symmetric and periodic properties that are leveraged by the FFT [13]. The meaning of these properties are as follows:

$$\text{Symmetry: } W_N^{k+\frac{N}{2}} = -W_N^k$$

$$\text{Periodicity: } W_N^{k+N} = W_N^k$$

With these properties, it allows the FFT to be split into the even and odd indexed values of the input. Take for example, a 4-point FFT, the 0th and 2nd even index values of the input share the same Twiddle factor with only a difference in sign between the two. This example also showcases another speed increase in the FFT in the calculation of the DFT. The input can be continuously reduced to smaller and smaller subsets of even and odd indices with the same Twiddle factor differing only in sign. In fact, the Cooley-Tukey FFT is extended until the DFT needs to be calculated for two input values with the same twiddle factor of a different sign. This becomes what is commonly known as a "butterfly" operation named in part because when the dataflow is visualized, the inputs and outputs are connected in what looks like a butterfly. Figure 1 shows the butterfly operation for two values visualized

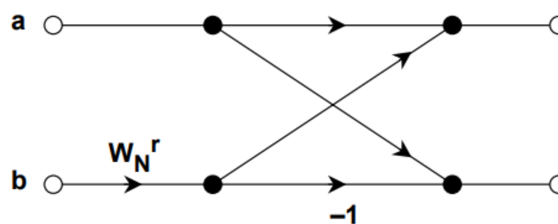

Figure 1: 2-point Butterfly Operation [14]

When the DFT is calculated on pairs of values, this is known as the Radix-2 Cooley-Tukey Decimation-in-time FFT. The number of inputs used in one butterfly operation can change (e.g., 2, 4, and 8) creating Radix-n FFTs. Then once the outputs are computed, these values can be used in another stage that would combine the Radix-n butterfly outputs and this is done until the full DFT is calculated. That is the speed increase of the FFT, once the first stage is calculated those values can be used in the next stage, this saves the next stage N/2 calculations. In this manner, the solution can be solved recursively and reducing the required number of multiplications. Figure 2 visually shows the flow of data and butterfly calculations for an 8-point DFT in three stages [11]

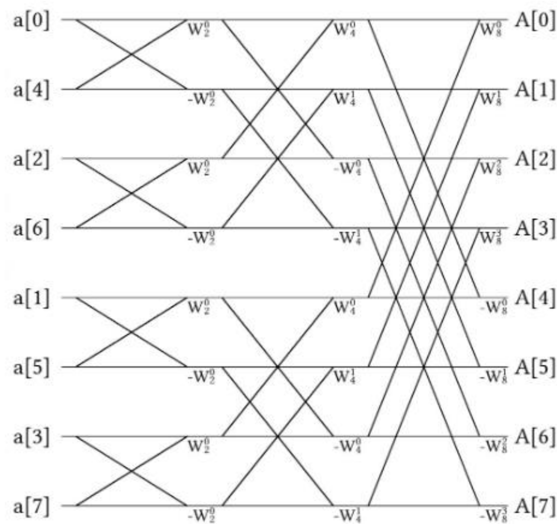

Figure 2: 8-point Decimation-in-Time FFT [14]

The first stage does a butterfly operation on pairs of data of even or odd indices. Figure 2 shows that for each of the 3 stages, only 4 or N/2 butterfly operations are required. That means only a total of 12 complex multiplications are required as opposed to the 64 required in the definition of the DFT for an input of length 8.

Real Fast Fourier Transform

Another speed increase is that for real-valued inputs, the FFT only needs to calculate the DFT of size N/2+1. This is because X[N-k] turns out to be the complex conjugate of X[k] saving time from redundantly calculating the DFT for k greater than N/2+1.

Sampling Frequency

According to the Nyquist-Shannon sampling theorem, to properly discretize a signal for a given frequency, it should be sampled at twice the frequency or higher [9].

Frequency Resolution

The frequency resolution is the ability of the FFT to differentiate between signals of specific frequencies. This is determined by the sampling frequency divided by the size of the FFT that is to be calculated.

Complex Magnitude

The complex magnitude of X[k] has the same units as the input signal and is calculated below in equation 1.14

$$\text{Mag}\,X[k] = \sqrt{(\text{Re}\,X[k]^2 + \text{Im}\,X[k]^2)} \qquad (1.14)$$

and it is calculated as the square root of the sum of the squares of the real and imaginary component [9], [15].

Phase Angle

The phase angle of the DFT output is in units of radians or degrees. This value relates the difference in phases of the input signal in relation to a reference cosine signal

at the frequency specified by the frequency bin. In equation 1.15 is the calculation for the phase angle for a given frequency bin

$$\phi[k] = \arctan\left(\frac{\text{Im } X[k]}{\text{Re } X[k]}\right) \text{ radians} \quad (1.15)$$

To find the phase angle, calculate the arctangent of the imaginary component over the real component of a specified frequency bin [9],[15]. This calculation outputs a value in radians which can be converted to degrees.

Calculation of the arctangent is not a trivial function to solve especially on microcontrollers. Using rational expressions or polynomials to approximate the arctangent create varying execution times and overhead costs [16], [17]. Typically, rational expressions provide the best accuracy between the two at the cost of higher computational resources to calculate division operations [16]. In [16], it provides a comparison study of both rational and polynomial approximations of varying degrees and providing their maximum errors and computational requirements. In conclusion, the following equation 1.16 is recommended and will be used in this implementation

$$x = \left(\frac{\text{Im } X[k]}{\text{Re } X[k]}\right);\ \arctan(x) \approx \frac{\pi}{4}x + 0.273x(1 - |x|), -1 \leq x \leq 1 \quad (1.16)$$

Equation 1.15 provided the best compromise in terms of accuracy and computational cost of all the expressions and polynomials in the study in [16]. It should be noted that arctan is only defined in the range between [-1, 1]. A common function that can be implemented easily is arctan2 which takes into consideration the sign of the inputs to properly return a value in radian or degrees with the defined range of arctan. The considerations are in equation 1.17 as follows:

$$\mathrm{arctan2}(y,x) = \begin{cases} \arctan\left(\frac{y}{x}\right), \text{if } x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi, \text{if } x < 0 \text{ and } y \geq 0 \\ \arctan\left(\frac{y}{x}\right) - \pi, \text{if } x < 0 \text{ and } y < 0 \\ +\frac{\pi}{2}, \text{if } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2}, \text{if } x = 0 \text{ and } y < 0 \\ \text{undefined}, \text{if } x = 0 \text{ and } y = 0 \end{cases} \quad (1.17)$$

Frequency

The frequency is the angular velocity of the AC power signal in units of Hz and is defined in equation 1.18 [18]

$$f(t) = \frac{1}{2\pi} \cdot \frac{d\theta}{dt} = f_0 + \frac{1}{2\pi} \cdot \frac{d[\phi(t)]}{dt} \qquad (1.18)$$

Rate of Change of Frequency (ROCOF)

The angular acceleration of a signal is in units of Hz/s. This measured value relates to the angular velocity of the sinusoidal input signal below in equation 1.19 [18]

$$ROCOF(t) = \frac{df(t)}{dt} = \frac{1}{2\pi} \cdot \frac{d^2\phi(t)}{dt^2} \qquad (1.19)$$

in the above equation, ROCOF(t) is the second derivative of the phase angle with respect to time (t), multiplied by one over two times π. Calculation of the phase angle from the DFT is required in the calculations of ROCOF.

CHAPTER 3: IMPLEMENTATION

In Figure 3 is a system block diagram of the different functional blocks of the PMU.

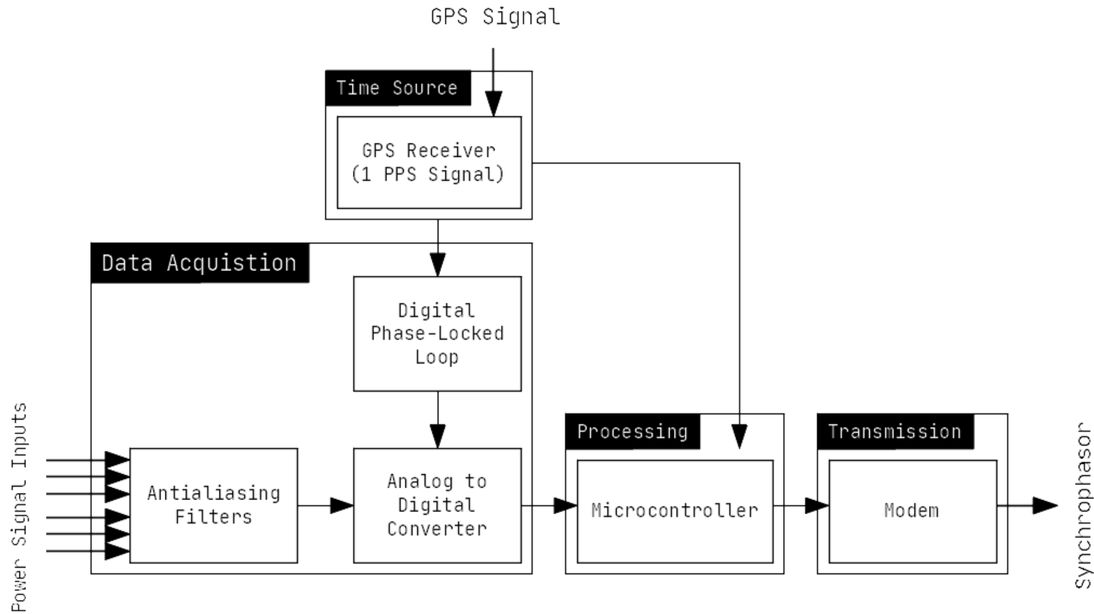

Figure 3: System block diagram of PMU

The following sections will go into detail of the design and implementation of each block, and some of the key features of chosen components.

Time Source

The PMU uses the MediaTek MTK3339 GPS receiver. This receiver has several features ideal for this implementation [19]:

- Ultra-High Sensitivity at -165 dBm (without a patch antenna) tracking
    - Typical received signal power from a GPS satellite is -127.5 dBm
- Acquisition sensitivity: -148 dBm (cold) / -163 dBm(hot)
    - Cold being the first acquisition after startup
- High accuracy 1 PPS single-ended signal with +/-10 ns jitter

- Low-power consumption (Acquisition: 25 mW; Tracking: 18 mW)
- NMEA0183 Output (3V logic level)

The GPS receiver comes installed on a breakout board provided open source by Adafruit. Figure 4 shows the GPS receiver on the breakout board.

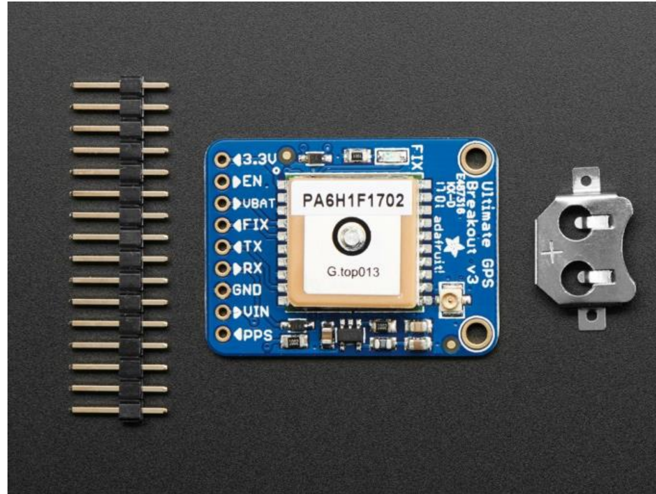

Figure 4: Adafruit Ultimate GPS Breakout. Photographed by Adafruit [20]

The two important outputs used directly by the PMU are the 1 PPS signal that is sent to the digital PLL to generate the ADC disciplining signal and the NMEA0183 output (via asynchronous UART).

NMEA0183 RMC is set as the output and is sent serially to the microcontroller to properly time-tag the calculated phasors. RMC is known as the recommended minimum specific GPS/Transit data and provides the required information for our synchrophasor without adding too much excess (e.g., Timestamp, validity of signal, Latitude, North/South, Longitude, East/West, Date, and a couple unused specifics).

While the receiver sensitivity is high, using an external active GPS antenna increases acquisition/tracking sensitivity. In this implementation an active antenna by

Chang Hong Information Co., Ltd. is used. The active antenna draws an additional 8.6 mA typically at 3V [21]. An image of the antenna is shown in Figure 5.

Figure 5: Active GPS antenna. Photographed by Adafruit [22]

Any active antenna can be used in place of the one used in this implementation. It needs to interface with the u.FL connector located on the breakout board.

The GPS receiver breakout was tested to verify functionality. Below are screen captures from an oscilloscope connected to the 1 PPS output, looking at peak voltage, settling minimum, rise time, and settling time. The image below is the statistical information after the oscilloscope received ~1000 pulses from the receiver with an attached active antenna. Environmental conditions were clear skies, temperature at ~75 F, humidity ~66%, with the antenna placed outside of a window on the second story of a building.

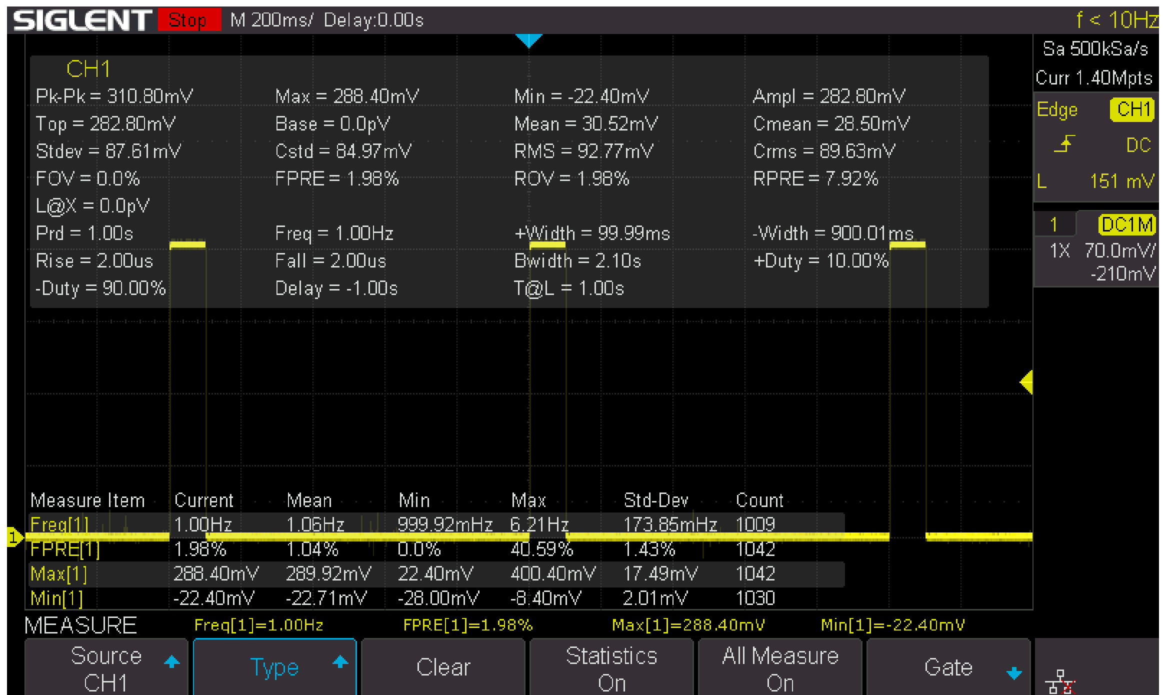

Figure 6: Signal information after ~1000 pulses from GPS receiver

Seen in Figure 6 is the 1 PPS signal after 1009 instances, with a mean of 1.06 Hz with a standard deviation of 0.17385 Hz. Therefore, with 95.4% confidence the portion of 1 PPS frequencies vary between 0.7123 Hz and 1.4077 Hz. The same test was conducted on another day with the same setup. The environmental conditions of the second day were overcast and foggy, temperature ~72 F, and humidity of 99%. Results of the test are shown below in Figure 7.

Figure 7: Signal information at ~1000 pulses from GPS receiver

After 1118 instances of the 1 PPS signal, the mean is 1.00 Hz with a standard deviation of 0.01718 Hz. Therefore, with 95.4% confidence the portion of 1 PPS frequencies vary between 0.96564 Hz and 1.03436 Hz. The differences in the confidence intervals can most likely be attributed to the ability of the receiver to receive a GPS signal from the satellites overhead at the time of testing.

The NMEA output message that is sent serially to the microcontroller from the GPS receiver was also verified. In Figure. 8, an example of the NMEA message and broken down into its component parts are shown.

```
===========================================================================
$GPRMC,161454.000,A,3704.8858,N,07624.7176,W,0.04,174.92,220321,,,D*7A

Time: 16:14:54.000
Date: 22/3/2021
Fix: 1 quality: 0
Location: 3704.8857N, 7624.7178W
Speed (knots): 0.04
Angle: 174.92
Altitude: 0.00
Satellites: 0
===========================================================================
```

Figure 8: Example of NMEA Output Message

A typical NMEA message is started with a '$' and then each section is separated with a comma. Immediately following after the '$' is the NMEA output type, in this case is recommended minimum specific GPS/Transit data (RMC). The most important portions of this message are the time, date, fix/quality, and location. Even the location is only needed once the PMU is first turned on or location is changed. The GPS receiver used in this implementation has an accompanying library provided by open source by Adafruit, that allows easy capture of the GPS information and accessing the component parts of the message.

The GPS receiver used in this implementation is one possible solution for the Time Source of the PMU. Any receiver can be used that meets the desired requirements for a given installation. The most important abilities are to output an accurate 1 PPS single-ended or differential tied to UTC time and reporting GPS information.

Processing

Processing will be done on chip in the Microchip ATSAMD51 ARM Cortex-M4 microcontroller. The microcontroller used in this implementation is broken out on the Adafruit Grand Central M4 express development board. An image of the development board is in Figure 9.

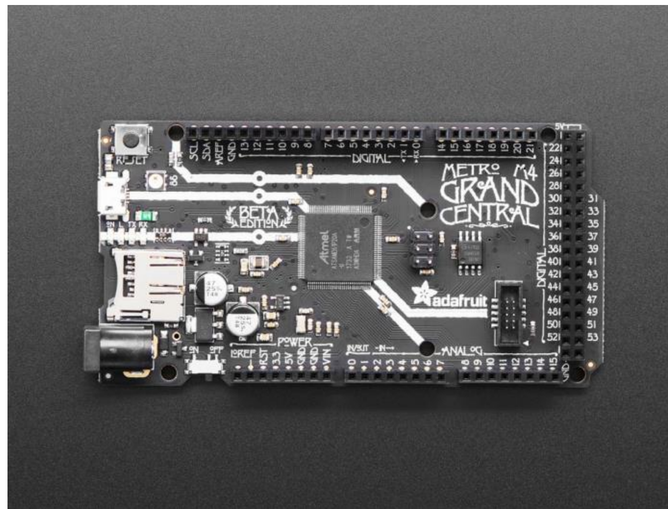

Figure 9: Adafruit Grand Central M4. Photographed by Adafruit [23]

The Cortex-M4 SoC is used due to its high clocks speeds of 120 MHz, 32-bit RISC architecture, floating-point unit, integrated SAR-ADC, and the inclusion of Digital Signal Processing (DSP)-specific instructions [24]. Tests on 256-point DFT calculation speed on microcontroller using DSP instructions on floating-point values were conducted and the results can be seen in Figure 16. The 256-point DFT is used because this is the desired DFT size for this implementation. The 256 floating-point values were generated in MATLAB for a 3 $V_{p\text{-}p}$ at 60 Hz with 30dBW noise with a 1.65 V DC bias over 1 period then used in the test code.

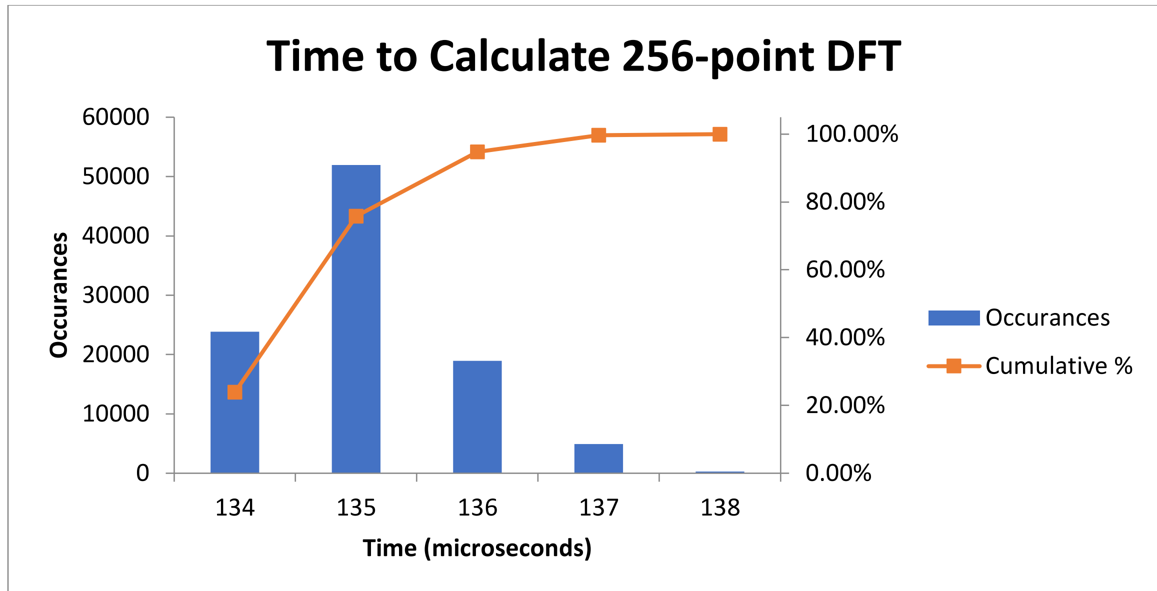

Figure 10: Frequency chart of 256-point DFT

Running analysis on the data of 100,000 samples yields that ~80% of execution times falls on 135 microseconds and below and the mean is 135.0588 microseconds while the median value is 135 microseconds. The difference in the mean and median this indicates there is a right-skew present in the data, calculation of the skew shows a value of 0.59961 meaning the data is moderately right skewed. A normal distribution function cannot be applied, instead the median will be used for the skewed data to calculate a confidence interval. However, the median of 135 needs to be confirmed. To further confirm that 135 is the median of the dataset, bootstrap analysis was used. 1000 subsets of 20 randomly sampled datapoints were generated, the median value for each subset was then calculated. In Figure 11, the results of the bootstrap analysis are shown.

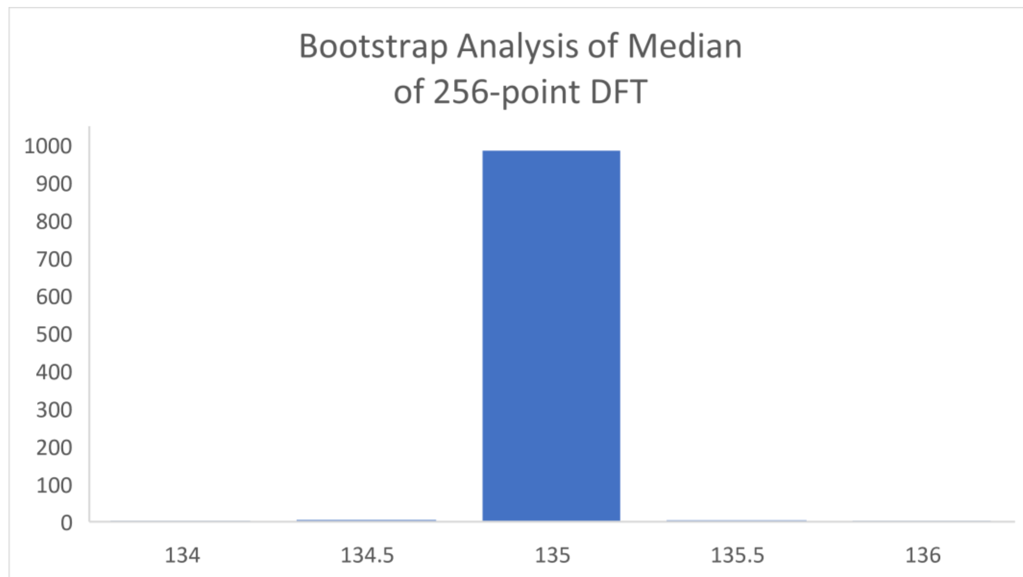

Figure 11: Bootstrap Analysis of 256-point DFT Median

The bootstrapped 95.4% confidence-interval for the median lies between [134.808, 135.192]. Now a 95.4% confidence interval for the execution time of all the samples can be given with a median of 135 and a standard deviation of ~0.806 which yields [132.388, 136.612] microseconds. From the previous section on the ADC, using the non-oversampled read time, it would take ~23724 microseconds to acquire 256 samples and calculate the DFT. That estimate is also only based on a single-phase line. Other factors like the structure and organization of the code, sampling of the 3-phase voltage and current lines will change output speed. Further speedups related to the microcontroller itself would be to decrease DFT size or implement DMA transfers of ADC data to memory. Decreasing the size of the DFT will also decrease the number of required calculations and therefore decrease execution time. Fixed-point integers could also increase speed of execution. Caution should be considered with the use of floating-point integers since they may lose precision in cases of overflow of arithmetic operations.

DMA transfer can help increase output by freeing up the microcontroller to perform DFT calculations while DMA controller directs data to memory.

Tests were also conducted on the 128-point DFT over 100,000 samples and are shown in Figure 18 to show the difference in execution times. The 128 floating-point values were generated in MATLAB for a 3 V_{p-p} at 60 Hz with 30 dBW noise with a 1.65 V DC bias.

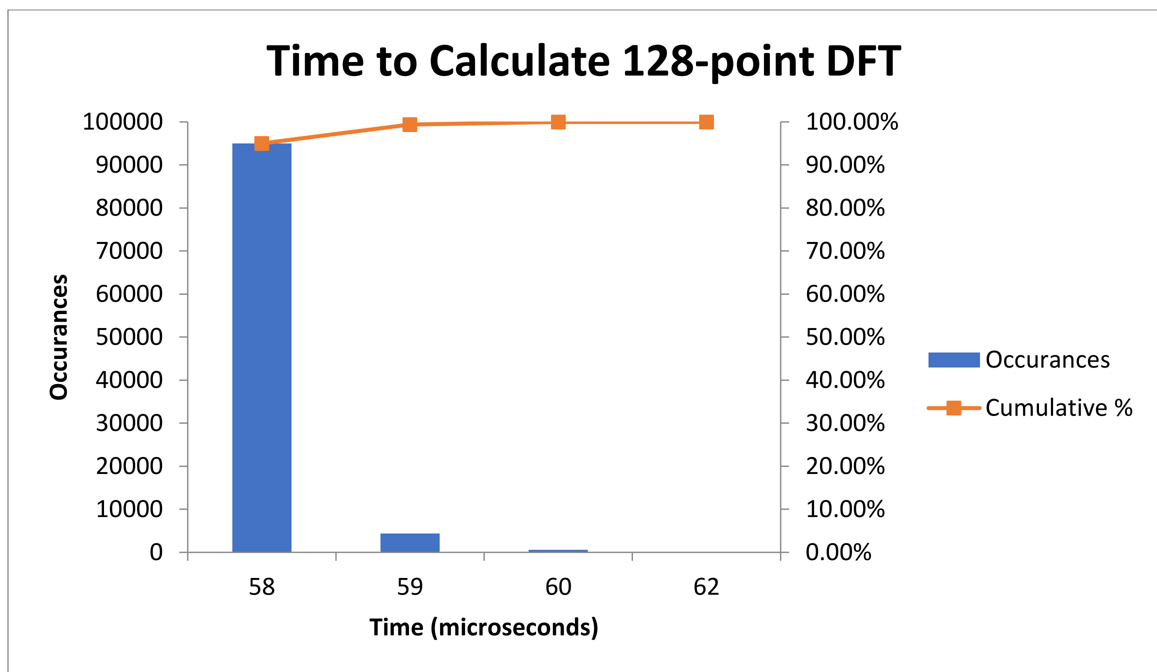

Figure 12: Frequency chart of 128-point DFT

Even before conducting further analysis on the data, the data is right skewed, but ~95% of all execution times complete in 58 microseconds per calculation of the DFT. This is roughly half the time required to calculate the 256-point DFT. While good in terms of speed, the tradeoff is a decrease in frequency resolution of a smaller DFT. However, in the Timings section, the choice of DFT size is limited by a few other factors. These factors are the time to execute a read on an ADC channel and the period between

sampling signal pulses for the ADC. Timings will further discuss how it is not viable to calculate the 256-point DFT or higher given the time to execute the read commands on the ADC channels which are discussed in Analog to Digital Converter and Analog Interface and Antialiasing Filter subsections of Data Acquisition.

The accuracy of the DFT on the microcontroller was also tested, this was compared to an DFT computed in MATLAB. The same datapoints used in execution speed of the 256-point DFT is used for this testing. Again, 256 datapoints were collected for one period of the signal and then given to the microcontroller and MATLAB to calculate the FFT. In Figure 13, the percent difference of the normalized, single-sided amplitudes of the outputs is shown.

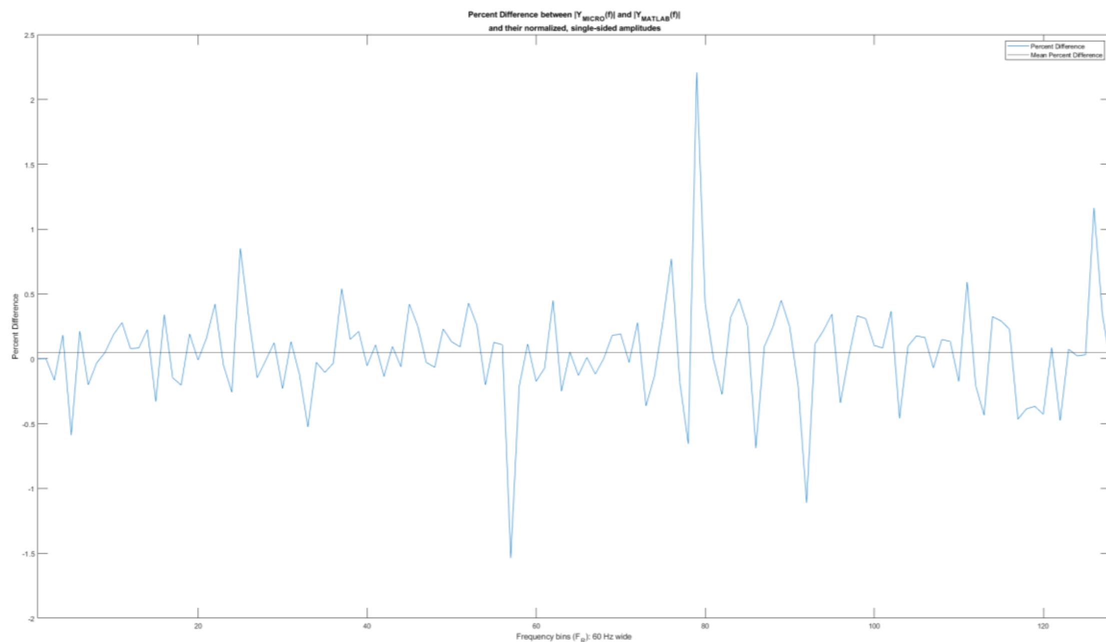

Figure 13: Percent difference of normalized, single-sided amplitudes of FFT outputs

Figure 13 shows that the mean of the percent difference between the microcontroller and MATLAB is calculated to be 0.0475 percent. This shows the microcontroller DFT is comparable to the MATLAB DFT.

Another thing to consider when thinking about FFT size is the frequency resolution of the FFT which is dependent on FFT length and the sampling frequency. The frequency resolution/bin sizes of the FFT will be limited by this value. This will be the ability of the FFT to differentiate between frequencies.

The flowchart for this section is shown in Figure 14.

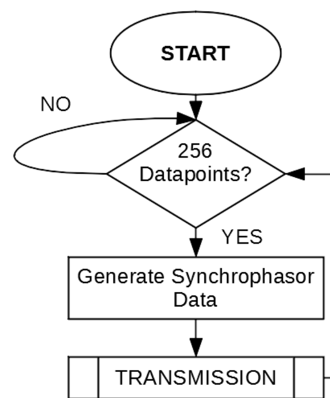

Figure 14: Flowchart of Processing Block

Once the desired number of datapoints per cycle is reached, the phasor is calculated and then time-tagged to create the synchrophasor. The data is then sent over ethernet using the User Datagram Protocol (UDP).

Data Acquisition

Phase Locked Loop (PLL)

The AD9544 is used in this PMU implementation. This chip is typically used in clock generation and can accept the single-ended 1 Hz, 1PPS signal coming from the GPS [25]:

- 5 pairs of clock output pins useable in differential LVDS/HCSL/CML or as 2 single-ended outputs
- 2 differential or 4 single-ended input inferences
- Single 1.8 V power supply operation with internal regulation
- Typical current draw ~400 mA
- 3.3V operation for I/O
- Built in temperature monitor/alarm and temperature compensation for enhanced zero delay performance

Below in Figure 15 is the functional block diagram of the IC

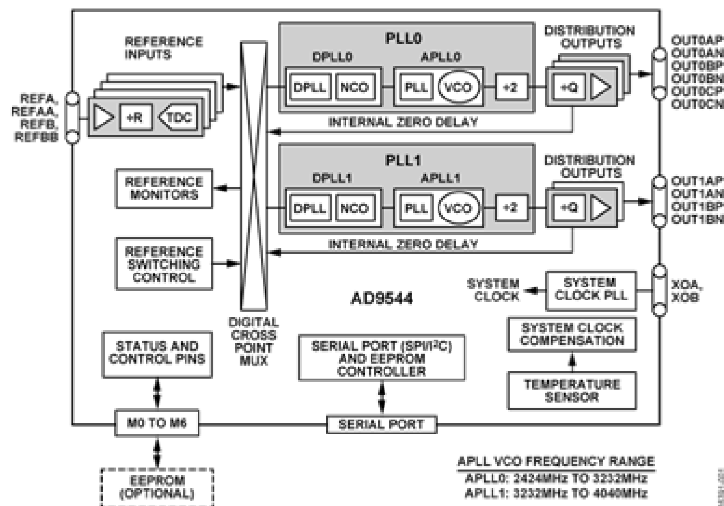

Figure 15: AD9544 Functional Block Diagram. Image by Analog Devices. [26]

There is an evaluation board that can be purchased that implements all functionality of this IC with supporting circuitry. Documentation on the schematics and PCB are openly available to the public on Analog Devices website. This implementation instead used a custom PCB to implement only the desired functionality while taking design considerations from the evaluation board. Below is an image of the custom PCB in Figure 16.

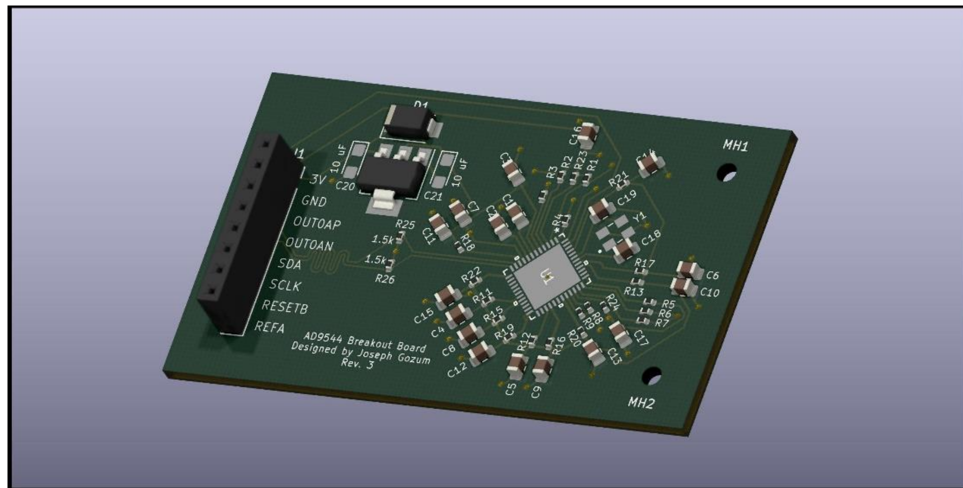

Figure 16: Custom AD9544 PCB

Using either an evaluation board or custom PCB requires programming of the digital PLL over I2C or SPI. Specific registers locations and value descriptions can be found on the website, but Analog Devices also offers free software that lets you set up the IC to desired specifications. After setting up the IC in software, it can output the locations and required values to write to the physical chip so that it matches with the software implementation. In this case, the digital PLL will take a single-ended 1 PPS signal from the GPS receiver and output two phase matched single-ended signals of the desired sampling frequency.

While using the digital PLL to generate the proper sampling signal from the 1 PPS signal is ideal, unfortunately hardware debugging of the custom PCB is too time-consuming for the scope of this project. Instead, the 1 PPS signal will instead be fed to Atmega 328P that is implemented on the Adafruit Metro (similar implementation to an Arduino Uno) and then a PWM signal of the chosen sampling frequency will be generated for use as the sampling signal for the ADC. This configuration is similar to one described in [8] to generate the sampling signal. An example of the PWM signal generated by the Atmega 328P is shown in Figure 17.

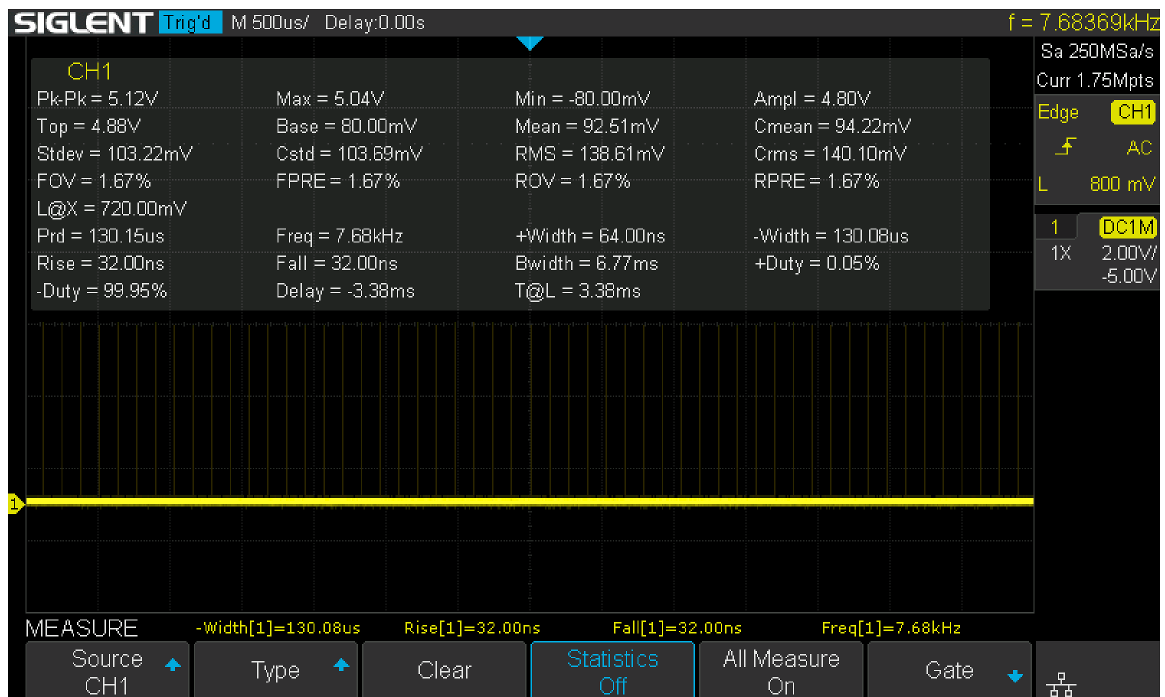

Figure 17: ADC Sampling Signal Generated by Atmega 328P

The measurements generated by the oscilloscope show that the PWM signal is at 7.680 kHz and a corresponding period of 130.15 microseconds.

Arduino Unos and their variants like the Adafruit Metro are known to produce PWM signals but are limited by the software implementations provided by Arduino in

their libraries. In order to generate the PWM signal shown in Figure 17, several internal registers must be properly set. The code and information on PWM generation was derived from [27] which shows how to generate non-typical PWM signals that can be produced by the Arduino libraries. However, that implementation in [27] used only the 8-bit timers and counters while this PMU implementation uses the 16-bit timers and counters to generate the sampling signals. However, the same steps can be followed while adjusting them for use with the 16-bit timers. Using the Atmega 328P datasheet [28] shows what specific registers to adjust and what bits to set to get the desired output using the 16-bit timers and counters. Table 1 shows the internal registers that must be set and their values to generate the desired sampling signal of 7.680 kHz.

Register	Binary	Hex
TCCR1A	1010 0011	0xA3
TCCR1B	0001 1001	0x19
OCR1A	0000 1000 0010 0010	0x0822

Table 1: Atmega 328P Internal Registers for PWM Generation

Given these register values, the Timer/Counters were set to operate in FastPWM mode with non-inverting output, using the system clock with no prescaling. The timer upper limit is then set by OCR1A which is equal to decimal value 2082. This value of 2082 can be found using formulas found in [28] and Section 15.9.3 Fast PWM Mode in [27].

Analog to Digital Converter (ADC)

When considering what ADC to use, there are several things to consider. This includes the desired resolution of the measured data which is determined by the bit

resolution. Ideally, there should be multiple channels that can simultaneously sample and hold the values. There is also the type of ADC, Successive Approximation (SAR) and Sigma-Delta (ΣΔ) are the most common. Of the two, SAR ADC have a much faster conversion rate, but the resolution range is smaller compared to sigma-delta. The sigma-delta can reach resolutions of 32-bits while SAR typically reach 18-bits. However, only SAR-based can have multiple channels that can sample so SAR-ADCs will be used. Specifically, the internal SAR-ADC on the ATSAMD51 on the Adafruit Grand Central M4 will be used.

Another thing to consider before choosing an ADC is to determine what is the desired sampling frequency. The sampling frequency determines how many points per period there is which also determines the FFT size. The FFT size and sampling frequency have a proportional relationship, when trying to compute the FFT per period, with a constant of proportionality equal to the operating frequency of the electrical power signals (~60 Hz typically in the US).

$$f_{sampling} = \mathrm{k} \cdot \mathrm{S}$$

Where

$f_{sampling}$	is the required sampling frequency for the desired FFT size in Hz
k	is the nominal frequency of the given input samples
S	is the desired FFT size

In the case of this implementation, the desired FFT size is 256-points and the operating frequency of the electrical power signals is 60 Hz. Therefore, the required sampling frequency is 15.36 kHz. However, later on it will be shown that at the 15.36 kHz

sampling clock is not possible because it is not enough time to sample on more than one or two channels of the ADC.

Originally the AD7606B was chosen over the on chip one on the microcontroller. However, setup and verification of the external ADC proved to be difficult. This implementation now uses the on-chip ADC. The ADC is similarly capable as the external at a lower resolution. The AD7606B has a 16-bit resolution without oversampling while the on chip has a 12-bit resolution. There are several additional benefits to using the internal ADC, while it has a lower resolution it can be increased through oversampling and averaging, higher sampling of 1 Msps over the 800 ksps of the external ADC, and finally the ADC registers can be directly linked to Direct Memory Access (DMA) transfers immediately after sampling which can free up instruction cycles for FFT calculation.

The oversampling method mentioned above to increase the resolution of the data does not come without some tradeoffs. When oversampling is used, it averages the ADC values over several samples, and this increases the time to get a fully averaged sample. The time can vary depending on the number of values that are being averaged. In Figure 17, the results for the execution time of an ADC channel read using the analogRead function with no oversampling are shown. By default, analogRead is not implemented with oversampling but can be adjusted with several lines of code seen in Appendix B. The following data does not include any data transfer and only concerns reading an individual sample from a single ADC channel. The ADC channel is this test case is connected directly to the 3 V power output header on the Adafruit Grand Central development board. Also take note that the times shown may not accurately represent

only the execution time of analogRead but will give a good estimate. This is due to the micros function used in the code in Appendix B for calculating elapsedtime which adds additional software overhead in time. However, it was found the micros function only adds between [0,1] microseconds to the execution time seen in the graphs.

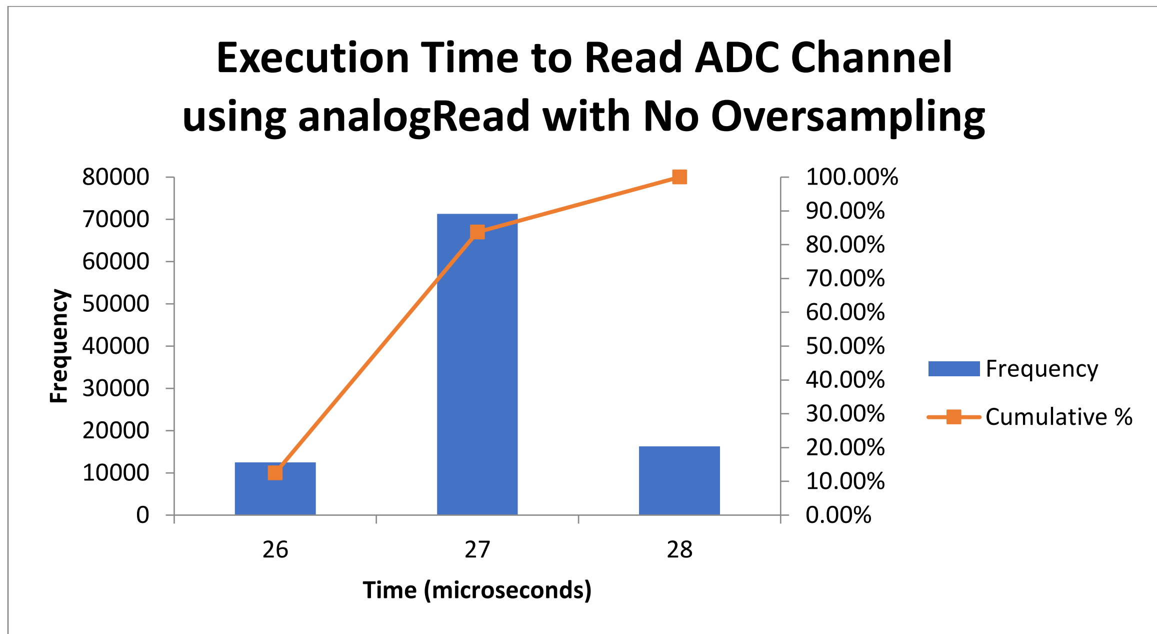

Figure 18: analogRead with no oversampling

The data execution time is approximately normal. Analysis of the mean and median show that the values are 27.0381 and 27, respectively. The skewness of the data is 0.035264 and can therefore be considered approximately symmetric. With this, a 95.4% confidence-interval for execution of a non-oversampled ADC reading can be expected to be within [25.96903, 28.10717] microseconds. Next, in Figure 18, the execution time for an ADC channel read using the analogRead function with oversampling is shown, 16 samples are averaged for the oversampling implementation.

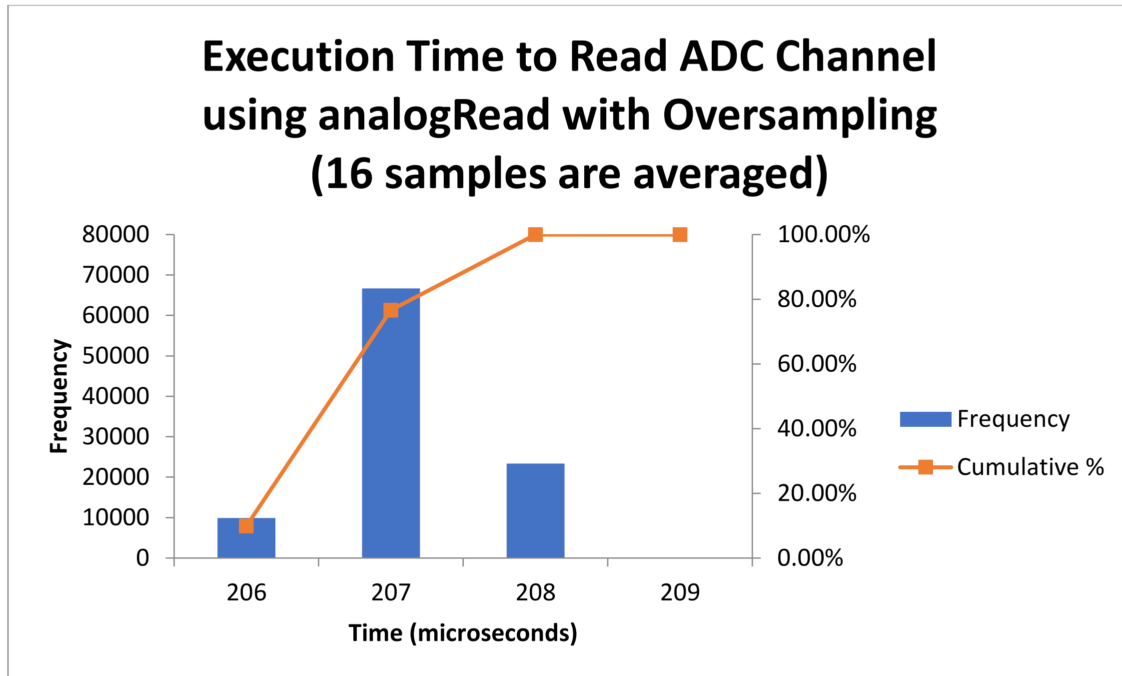

Figure 19: analogRead with oversampling

With this data, the mean, median, and skew are found to be 207.13525 microseconds, 207 microseconds, and 0.032692 respectively. This allows the data to be considered approximately symmetric. A 95.4% confidence-intervals for execution time of oversampled ADC reading yields [206.0127, 208.2578] microseconds. This increase in time to oversample is not viable as the period between sampling pulses for the 128-point and 256-point (i.e., 130 and 65 microseconds respectively) would not allow even one full channel capture on the ADC to complete before the next pulse. Therefore, the non-oversampled implementation of analogRead will be used. The Timings section will further elaborate on this.

Analog Interface and Antialiasing Filter

The analog interface from the power lines to the ADC channels is a transformer. It couples together the power lines and the inputs to the ADC and provides electrical isolation between them. In this implementation, a step-down signal transformer takes a 120 VAC and steps it down to 12 VAC, with a max current output of 1.6 A. While these voltage and current values are more manageable, the microcontroller used is not able to handle such values. In order to further reduce the voltage, a basic resistive voltage divider is used to step down 12 V_{p-p} to ~3.3 V_{p-p}. However, the voltage signal is centered around 0 V, which means the signal oscillates between positive and negative values. A DC bias is added to bring up the whole signal into the positive region. Figure 19 shows the output after the voltage divider and DC biasing on the secondary of the transformer for three phase lines at 120 VAC at 60 Hz with some non-nominal frequency components in LTSpice.

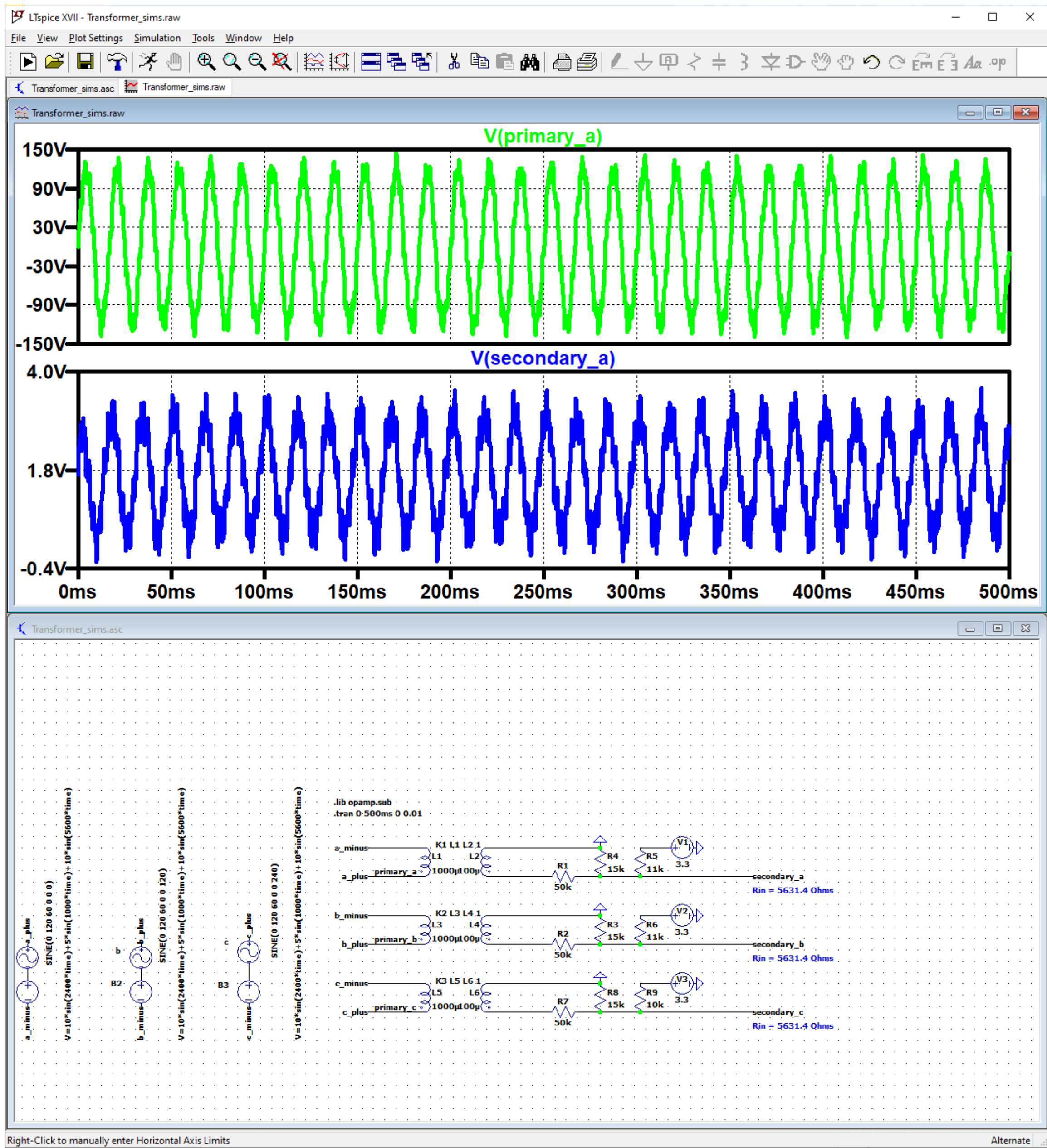

Figure 20: Analog Interface LTSpice Simulation

The previous section produced a resistive network that further lowered the voltage from 12 V_{p-p} to ~3.3 V_{p-p} and added a DC bias. However, another concern is that realistic power signals may have lower and/or higher frequency components that are introduced by the different mechanisms used to generate and distribute electricity. Therefore, a low-

pass filter that can attenuate at least the higher than nominal frequency (60 Hz) components is required. This implementation will use an active low-pass filter at unity gain. An active low-pass filter attenuates the higher frequency components and separates the high impedance created from the voltage divider, pull-up resistor, and RC circuit and creating a low impedance output [29]. Low output impedance is required because this has a direct effect on sampling time and accuracy for the SAR-ADC. A SAR-ADC like the one used in this implementation has an internal RC circuit per channel that is used to measure the voltage. However, if a large external impedance (>10 kΩ) is added, this will cause the charging time of the capacitor to be longer than any known sampling times stated in the datasheet. If the capacitor is not properly charged in the allotted time, then the reading will be inaccurate [30]. Therefore [24] provides known sampling times for specific resistance ranges. To get the shortest sampling time at 12-bit precision, the external impedance going to the ADC channel must be less than 147 Ω.

The active low-pass filter will consist of an RC circuit placed before an op-amp that will be set to unity gain. The cutoff frequency is set according to desired specifications and is set to 65 Hz in this case. From there, the values of the resistor and capacitor are set until the desired cutoff frequency is reached. Below is the equation to determine the cutoff frequency (65 Hz), and an equation that was derived to find the value of capacitance [29].

$$f_C = \frac{1}{2\pi RC}\ \mathrm{Hz}$$

$$C = \frac{1}{2\pi f_C R}\ \mathrm{F}$$

The first equation is used to determine the cutoff frequency, in this implementation several values must be chosen. The desired cutoff frequency (i.e., fc or f_{-3dB}) is 65 Hz, then the equation is rearranged to calculate the required capacitance. In an RC circuit it is much easier to adjust the resistance. In the second equation, it is shown that R and C also have an inverse relationship. Therefore, R can be made arbitrarily large to reduce the value of C. Smaller capacitance means physically smaller capacitors that can be used. R here is equal to the series connection of the arbitrarily chosen 20 kΩ resistor, and the 5357.14 Ω equivalent resistance of the divider and pull up resistor. The total resistance in the cutoff frequency calculation is 25357.14 Ω. To get a 65 Hz cutoff frequency with the chosen resistance, the capacitor would need to have a value of 0.1 μF. This is an acceptable value of capacitance, capacitors in this range are small, widely available, and affordable. From here, the voltage then goes through the op-amp with unity gain. Figure 20 shows a circuit implementation of the active low-pass filter at unity gain with the chosen values.

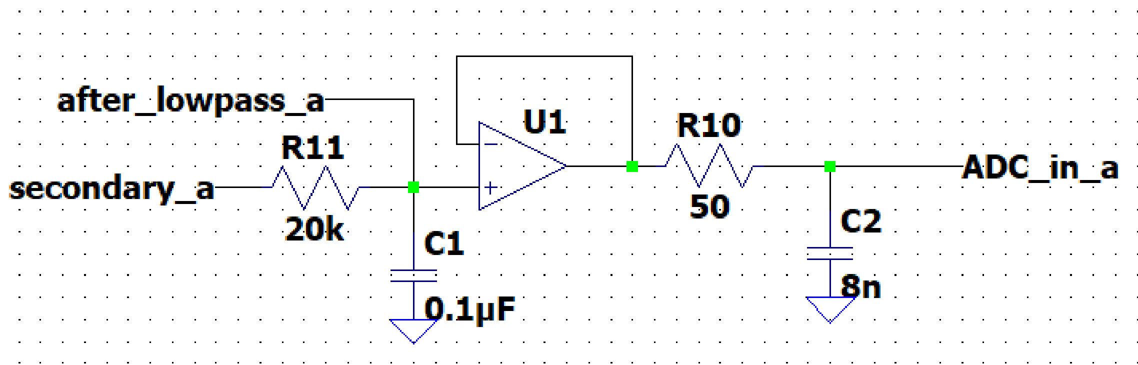

Figure 21: Active Low-pass Filter Op-amp Protection

In Figure 20 the active low-pass filter for a single phase was shown but on the output side there is another RC circuit. This portion has the objective of stabilizing the op-amp voltage and current to correctly the drive the SAR-based ADC. While the unity gain op-amp ideally outputs the input signal at lower impedance, without additional supporting circuitry it is vulnerable to currents from the internal RC sampling circuit. In [30], it provides a methodology in designing a RC protection circuit for the op-amp while maintaining low impedance, that circuit is seen in Figure 18 at the op-amp output. This methodology involves, characterizing the input signal, ADC requirements, choosing the values for the external protective RC circuit, and finally op-amp requirements. Using the methodology stated in [30], it produces the following characterizations, requirements, and values for the ADC and its input(s) for this implementation seen in Tables 1, 2, 3, 4, 5, and 6.

Input Signal	
Highest Frequency	60 Hz (single channel)
Largest Voltage Swing	0 to 3.3 V
Accuracy	805.7 µV LSB size / 12-bit with range of 3.3

Table 2: Input Signal Characterization

ADC Requirements	
Minimum Requirements	• Min. sampling freq. two times higher than max. signal freq. • Additional 10 to 20 x multiplier
Sampling Rate	> 2.4 ksps

Table 3: ADC Requirements

The internal SAR-based ADC on the ATSAMD51 used in this PMU sufficiently meets the minimum requirements desired. Table 2 provides important characteristics of the internal ADC. ADC specifications from [24] will be used for further calculations.

ADC Characteristics	
t_{ACQ} (ADC Acquisition Time)	1 TAD = [62.5, 6250] ns
C_{SH} (ADC Input Capacitance)	[min, max] = [2,3] pF
k (12-bit time constant multiplier)	12
V_{FSR} (Full-scale input range of ADC)	3.3 V

Table 4: ADC Characterization

The information provided on the ADC characteristics are important for the further calculation of the resistor and capacitor values that are placed between the op-amp and internal sampling circuitry. It is also of important note that for increasing values of

Rsource, [24] provides increased sampling times. This implementation chose to use an Rsource ≤ 147 Ω to minimize sampling time.

External Capacitor (C_{FLT})	
$C_{FLT} = Q_{INTERNAL} / (805.7\ \mu V)$	[8, 12] nF

Table 5: External Capacitor Requirements

In [30], a comparison of different capacitor dielectrics (X7R, Z5U, Y5V, and Silver Mica) shows that different dielectrics materials introduce increasing total harmonic distortion and noise (THD+N) at increasing frequencies. While at the 60 Hz of the PMU inputs, the different dielectrics have the same THD+N. So, any of the dielectrics listed are suitable for use. This implementation uses a C0G, 8200 pF/8.2 nF capacitor.

External Resistor (Rsource/R_{FLT})	
$R_{FLT} \geq (0.60 * t_{ACQ})/(k * C_{FLT})$	[0.39, 26] Ω

Table 6: External Resistor Requirements

As long as the Rsource is greater than or equal to 26 Ω it will be suitable for any value of C_{FLT}. Another limiting factor is the sampling time, this implementation stipulated a desired Rsource value to be less than 147 Ω. Therefore, Rsource is arbitrarily chosen to be a common resistor value of 50 Ω.

Primary Op-Amp Buffer Specs	
Gain Bandwidth Product (GBWP) • $GBWP > 4 * f_{FLT\ f\text{-}3dB}$ • $f_{FLT\ f\text{-}3dB} = 1/[2\pi*50*8.2nF]$	$f_{FLT\ f\text{-}3dB} = 388.183$ kHz $GBWP > 1.552731$ MHz
Output Impedance of Op-amp (R_O) • Desired: $R_o \leq 9 * R_{FLT}$	$R_O \leq 450\ \Omega$
Closed Loop Gain Bandwidth (f_{CL}) • Desired: $f_{CL} > 2 * f_{FLT\ \text{-}3dB}$	$f_{CL} > 776.366$ kHz
Slew Rate to Track 60 Hz input (SR_{OPA})	Minimum = 0.0006 V/us Optimal = 0.0012 V/us

Table 7: Primary Op-Amp Requirements

Using these values, the Analog Devices OP279 op-amp was chosen. Simulation results after choosing all resistor and capacitors values with an ideal op-amp in LTSpice are shown in Figure 21. The simulations also use the same input signals shown in Figure 19.

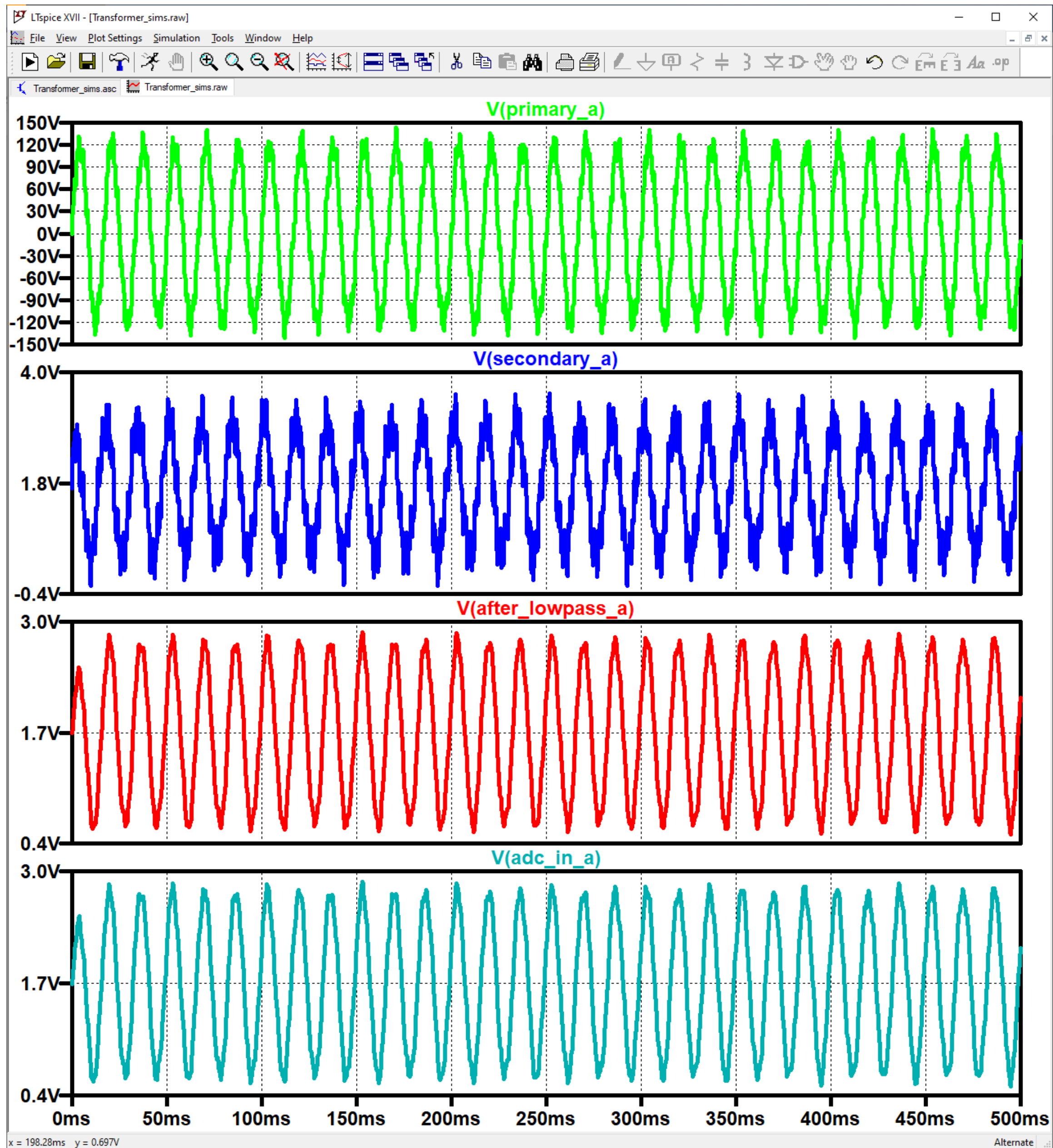

Figure 22: Voltage signal at different points of analog interface to ADC

To verify the simulations, a physical implementation of the design was tested. First, a test of chosen op-amp was conducted. Oscilloscope readings in Figure 21 and Figure 22 show the input and output respectively, are tracking well and show little to no signal

attenuation. The signal is a 3.3 $V_{p\text{-}p}$ sine wave at 60 Hz with a 1.65 V DC bias that is generated by a waveform generator.

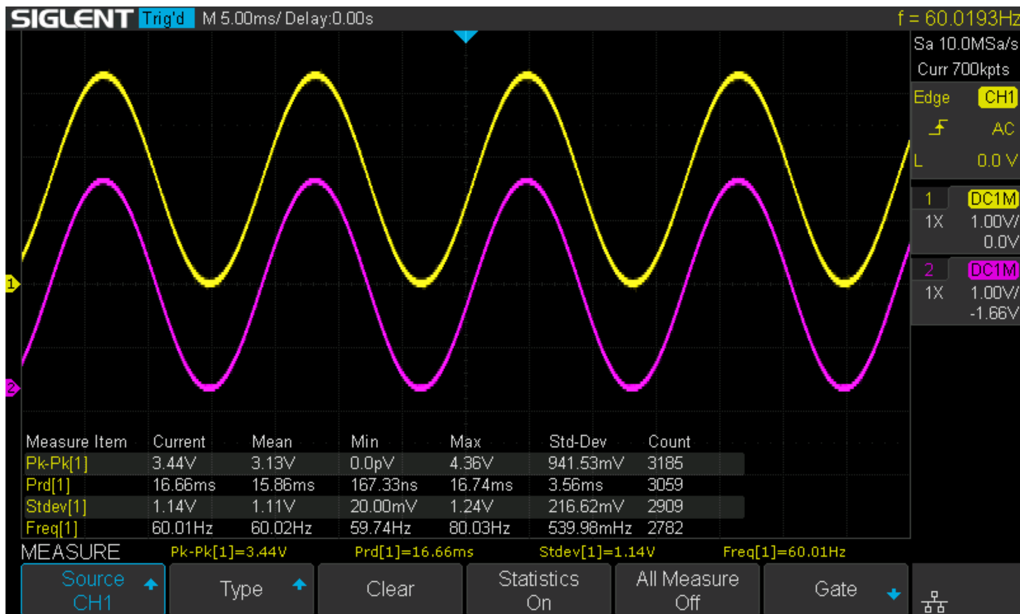

Figure 23: Op-amp input and output tracking, Oscilloscope Channel 1

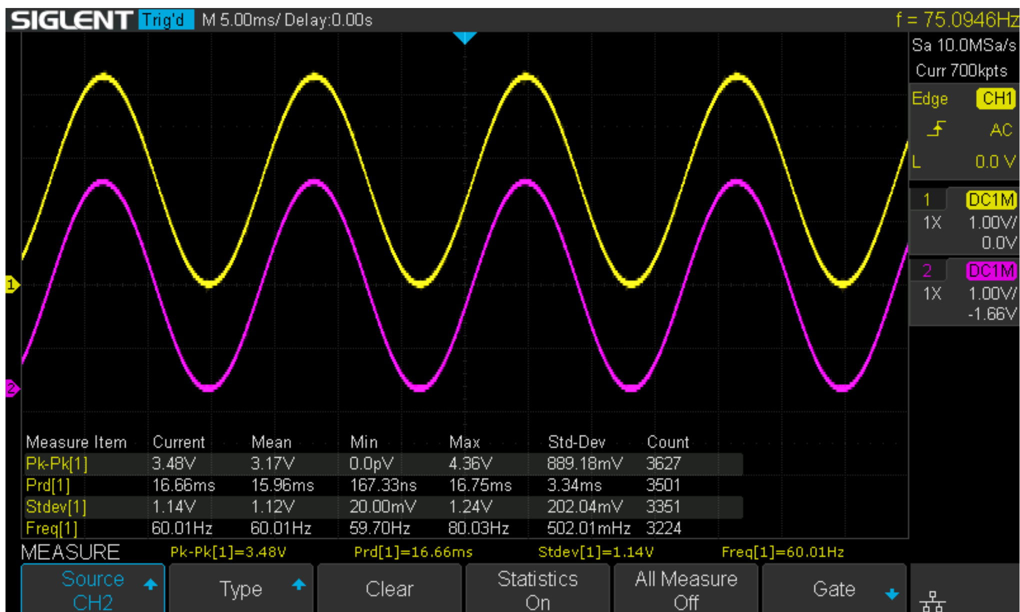

Figure 24: Op-amp input and output tracking, Oscilloscope Channel 2

The measurement values between the input and output channels match closely with each other just expected form simulations and design. Now, using the fully designed front-end of the ADC, Figure 25 will show the execution time of the analogRead function. The same input from testing the op-amp will be used.

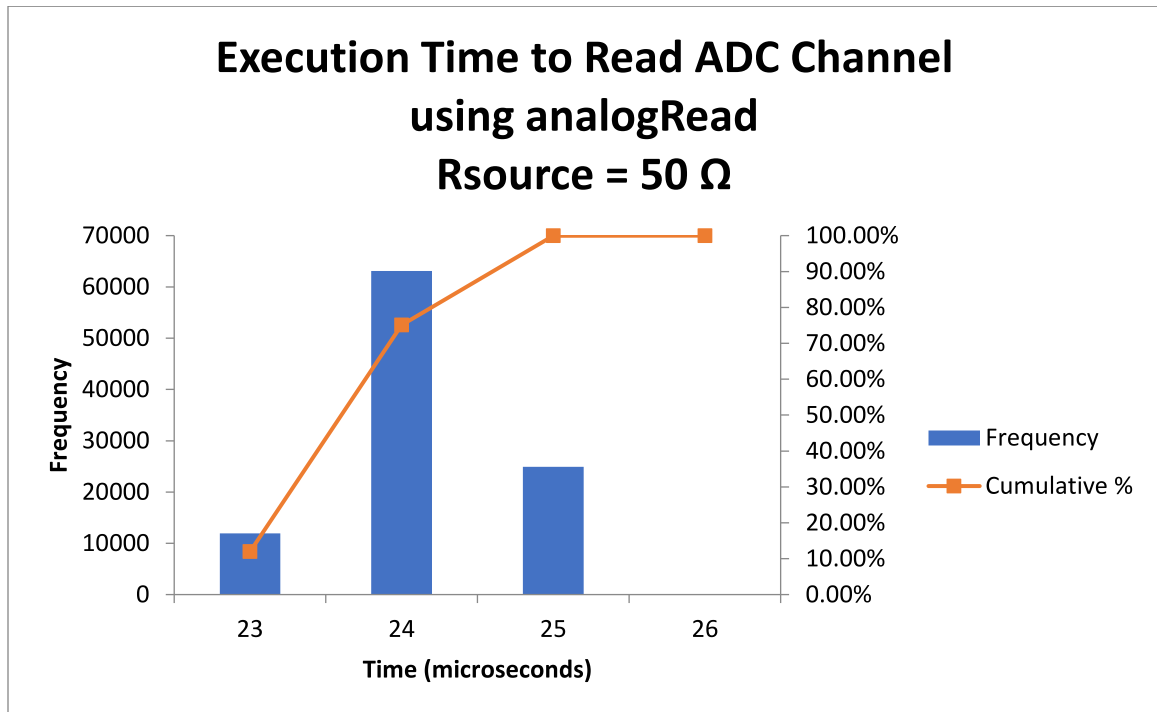

Figure 25: Execution Time of AnalogRead with RC Interface

Previously, when testing the execution time certain variables were taken for granted. One being the impedance seen by the ADC channel from the power output pin was assumed to be low. Second, the default implementation of analogRead has the sample length set to 5. Now, the Rsource is set deliberately to 50 Ω and the sample length is set to 1. The sample length change is reflected in the updated test code in Appendix D. Comparing the previous execution time now with the current times, there is a noticeable several microseconds decrease in the execution time with the adjustments. The previous data had

a mean around 27 microseconds, while now the mean is around 24 microseconds. Decreasing the sample time will help to free more time for DFT calculation.

Transmission

The WizNet W5500 is used to easily implement communications over ethernet with minimal testing and board preparations. In this implementation, it is provided on a breakout board. A photograph of the board is shown below in Figure 26.

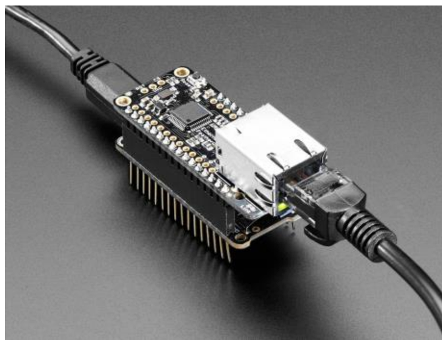

Figure 26: Adafruit Ethernet Feathering. Photographed by Adafruit [31].

The W5500 communicates with the microcontroller over Serial Peripheral Interface (SPI). Once the synchrophasor data is generated, the data is sent to the W5500 over SPI and then sent over the specified protocol (UDP) to the desired endpoint. An open-source library developed by Arduino is used to set chip and communication settings. Figure 27 shows the flowchart for the transmission functional block.

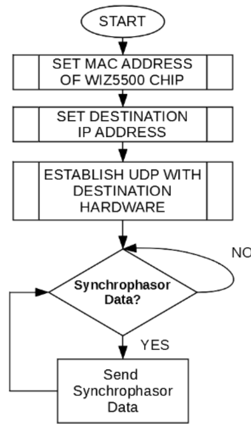

Figure 27: Flowchart for Transmission Block [32]

The Cortex-M4 can be configured with Ethernet 10/100 capabilities. However, this would require additional time for software and hardware testing. Instead, a separate Ethernet controller is used in this implementation for ease of use.

Tests were conducted on how long it would take to send a full PMU output message which consists of 336 Bytes of data per channel, not including any overhead associated with UDP. In Figure 28 is a histogram of times it took to send a full PMU message over UDP.

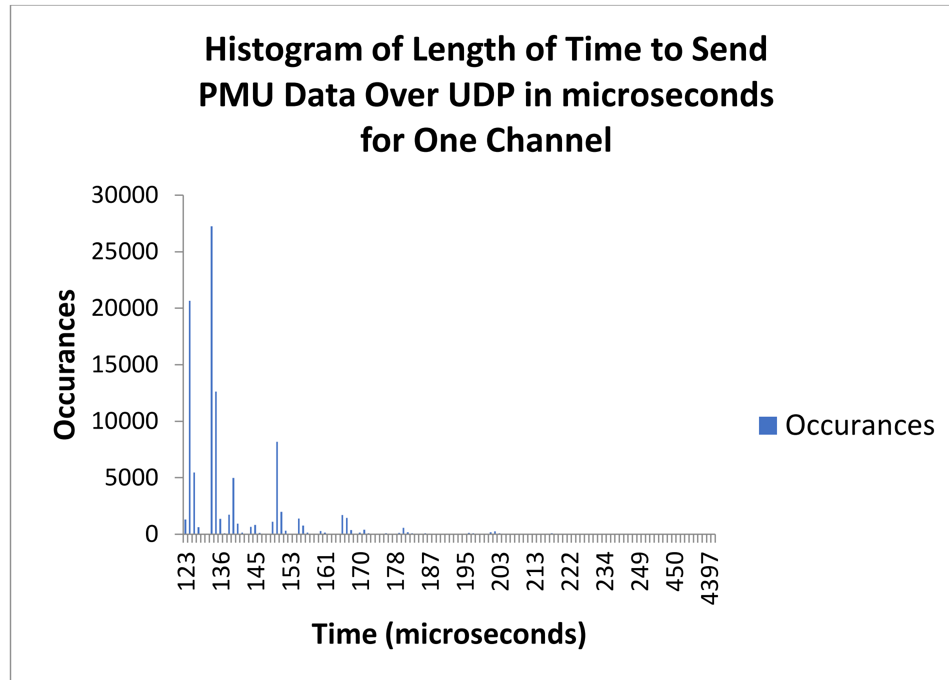

Figure 28: Histogram of Times to Send Full PMU Message

Looking at the spread of times seen, generally the data can be sent before the next FFT needs to be calculated and sent out. In a later section known as Timings will show during the period between sampling discipline signals, there is idle time that can be used for transmitting data. In the ideal case, the full message can be sent within a couple periods of the sampling disciplining signal. However, there are instances that are shown in the data where it takes greater than 3000 microseconds to send the message. In those cases, there is the possibility that there will be losses in the PMU output. A previous message in transmission may overlap with the calculation and transmission of the next message. This will only be worse if such a spike in transmission occurs for multiple or all channels and consecutively. However, looking directly at the data shows that instances of times to send in excess of 3000 microseconds do not occur often. Figure 29 shows another histogram of times to send a full PMU message with occurrences of less than one removed.

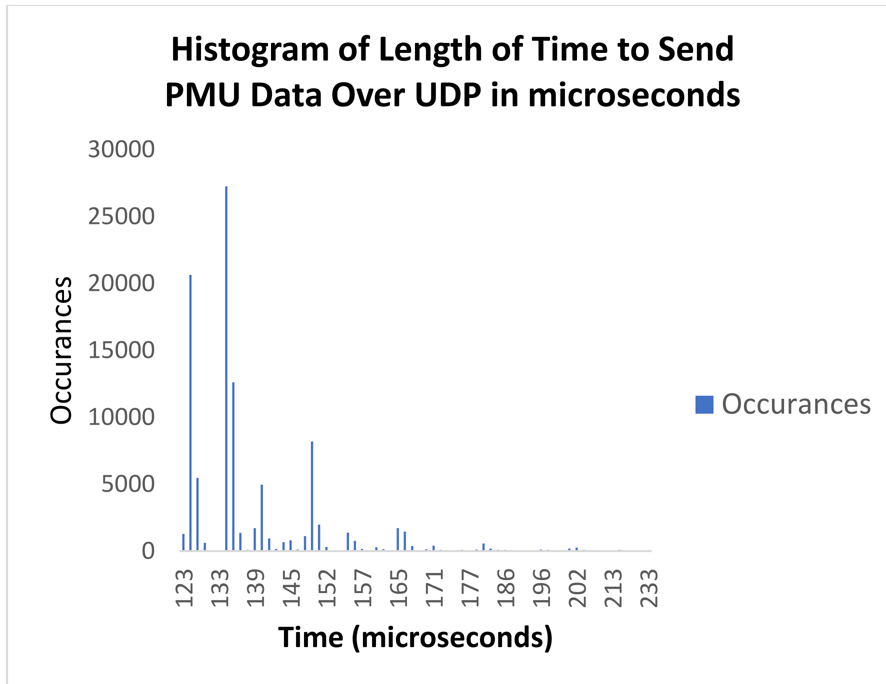

Figure 29: Reduced Histogram of Length of Time to Send Full PMU Message

The range of times now looks a lot better with the upper and lower limit being able to execute within several periods of the discipling signal and before the next PMU message. Using these timings, it would also be possible send a full PMU message for all three channels.

While these times are good, UDP is known to have questionable reliability and correctness during transmission. There are no methods implemented unlike in TCP which uses checksums to ensure correctness or ensure transmission of dropped packets. Even during testing of the Ethernet Featherwing, occasional UDP messages would be incorrect.

Timings

As discussed in previous sections, the time to execute a read on all the channels of the ADC and the period between sampling pulses are two important factors that limit the choice of FFT. Figure 30 graphically shows per period usage of the microcontroller for a

65 and 130 microsecond period (i.e., 15.36 kHz and 7.680 kHz sampling signal respectively).

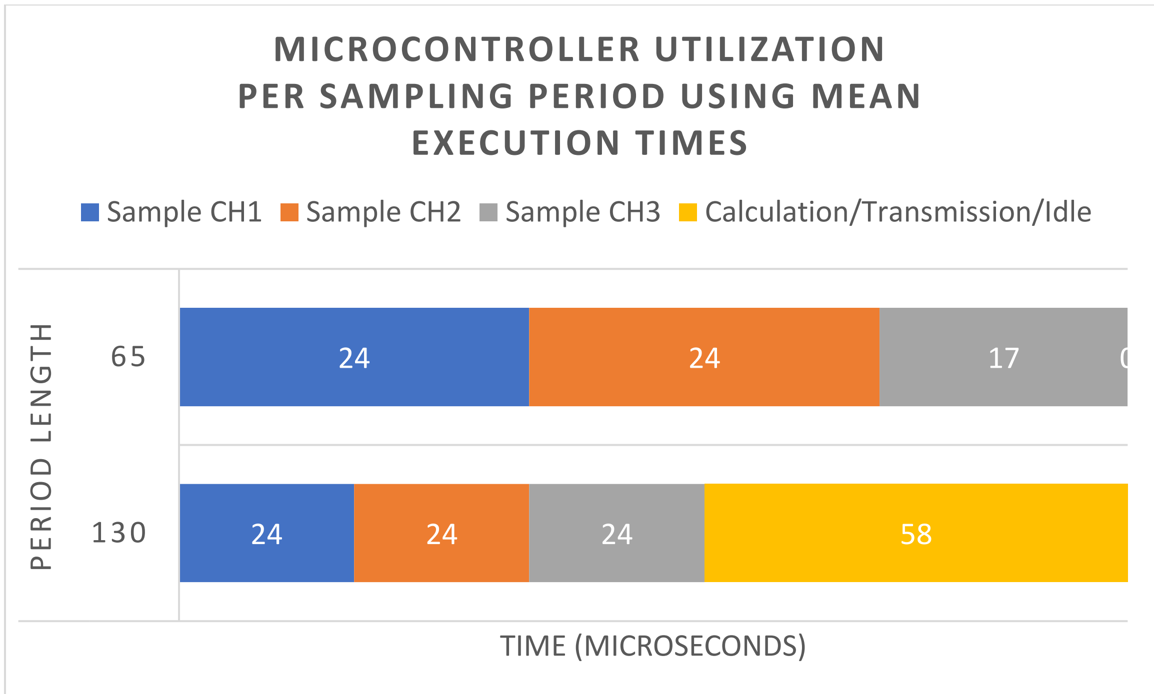

Figure 30: Microcontroller Utilization Per Sampling Period Using Mean Execution Times

In the case of the 65 microsecond period of the 15.36 kHz sampling signal, given the mean time to read per channel, it is not possible to execute a complete read before the next pulse which initiates an interrupt. That estimate also does include any additional overhead that may be present with the code implementation. Figure 30 shows that a complete read of channel 3 would never occur and that there is no excess time for any other calculations or transmission of data. A sampling frequency 15.36 kHz also correlates to a 256-point FFT, making it unfeasible and limiting the 128-point FFT as the next best option if trying to compute an FFT per period. The sampling frequency could be lowered even further if it's desired to maintain the FFT. Decreasing the sampling frequency and maintaining the FFT will also improve frequency resolution.

CHAPTER 4: EVALUATION METRICS

The following summary of evaluation metrics come from the standards set forth by IEEE/IEC 60255-118-1 [20], it highlights some of the main metrics to test.

To evaluate performance, the measured values generated by the PMU will be compared against reference values using several key formulas. The formulas used are: Total Vector Error (TVE), Frequency Error (FE), and ROCOF Error (RFE). Several other metrics to consider are measurement response and delay time, overshoot/undershoot, reporting latency, operational errors, and etc.

Based on the evaluation of the above-mentioned metrics and those in [20] help to determine how the PMU complies.

Total Vector Error (TVE)

The TVE is defined as:

$$TVE(n) = \sqrt{\frac{\left[\hat{X}_r(n) - X_r(n)\right]^2 + \left[\hat{X}_i(n) - X_i(n)\right]^2}{[X_r(n) + X_i(n)]^2}}$$

where

n	is the discrete report number representing the report time
$\hat{X}_r(n)$ and $\hat{X}_i(n)$	are the real and imaginary PMU estimates at report time n
$X_r(n)$ and $X_i(n)$	are the real and imaginary reference values at report time n

this the difference or error between the measured and reference vectors. This calculated value considers the amplitude and phase difference together and is normalized with respect to the reference signal.

Frequency and Rate of Change of Frequency (ROCOF) Error

The FE and RFE are both evaluated the same way as the difference of the measured values from the PMU and the reference values. They are in terms of Hz and Hz/s respectively:

$$FE(n) = \left(f_{measured}(n) - f_{reference}(n)\right)\ [Hz]$$

$$RFE(n) = \left(\left(\frac{df}{dt}\right)_{measured}(n) - \left(\frac{df}{dt}\right)_{reference}(n)\right)\left[\frac{Hz}{s}\right]$$

The measured and reference values are for the same time, which is given by the time tag from the corresponding measured and reference value.

Measurement Response and Delay Time

Measurement Response Time

Defined as the measurement response time to transition between two steady-state measurements before and after a step change is applied to the input. This can be determined as the difference between the time that the measurement leaves a specified accuracy limit and the time it reenters and stays within the limits when a step change is applied to the PMU input. Time is based on the UTC time scale.

In order to test, the input will be held at steady-state before and after a positive or negative step change in amplitude or phase. This step change and the measurement response time will be characterized by the TVE, FE, or RFE.

Results of this test help to categorize the PMU between Performance P-class and Measurement M-class.

Measurement Delay Time

Defined as the time interval between the instant that a step change is applied to the PMU input and measurement time that the stepped parameter achieves a value that is halfway between the initial and steady-state values. UTC is always used.

Along with the above condition the test for measurement delay time also applies a positive or negative step in amplitude or phase between two steady-state periods.

Results of this delay time evaluation are used to verify that the time tagging of synchrophasors measurement time is properly compensated for the filtering system group delay. This is a basic performance method associated with signals and measures the amount time for the different signal components of a signal to go from the measuring device input to output.

Overshoot and Undershoot

Aberrations before and after a transition such as a step change in phase or magnitude. The overshoot and undershoot magnitudes relative to the amplitude of the step are limited.

Measuring this performance metric determines if a PMU can accurately recognize and report dynamic changes in power signals. Abrupt changes in magnitude or phase are common and can help diagnose problems along the transmission lines.

However, if reporting rates of synchrophasors (F_s) is lower than 10/s, they do not have to meet dynamic performance requirements [18].

Reporting Latency

Defined as the maximum time interval between the data report time as indicated by the data time stamp, and the time when the data becomes available at the PMU output (this is when the first bit of the output message is available to the communication interface).

Operational Errors

The PMU must communicate all internal problems experienced during runtime such as, ADC errors, memory overflow, calculation overflow and any other condition that could cause an error in the measurement.

CHAPTER 5: METHODOLOGY

This project is based on the design of a low-cost and open-source PMU system based around a 32-bit RISC microcontroller. Specifically, the ATSAMD51 running an ARM Cortex-M4. The process begins by designing and assembling the different functional blocks of the PMU system. The system-block diagram in Figure 31 shows the different functional blocks

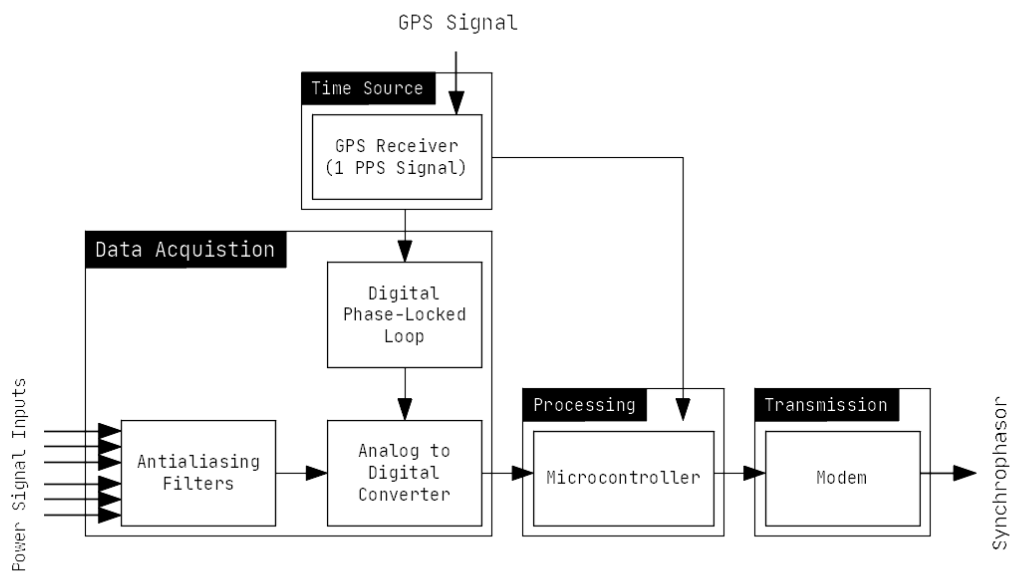

Figure 31: System-block Diagram of PMU system

Once each block is assembled, it will be tested to ensure it behaves properly. After each block is assembled and tested, they will be slowly integrated together and tested again as a new group. This integration will eventually encompass all blocks until the system is fully assembled.

The next step will be to follow the evaluation and compliance testing specified in [18].

Ideally, the PMU would be tested used real power signals but currently there is no safe access to such power signals for testing. Instead, an arbitrary function waveform generator will be used to mimic the signals that the PMU may see in actual operation. Then the PMU output will be evaluated based on the metrics set in [18].

Below in Figure 32 is a diagram of the system that will be under test. Figure 33 shows a photo of the actual setup used. Using the function waveform generator, it's limited to producing two outputs at two different phase shifts or one signal composed of multiple signals of different frequency components.

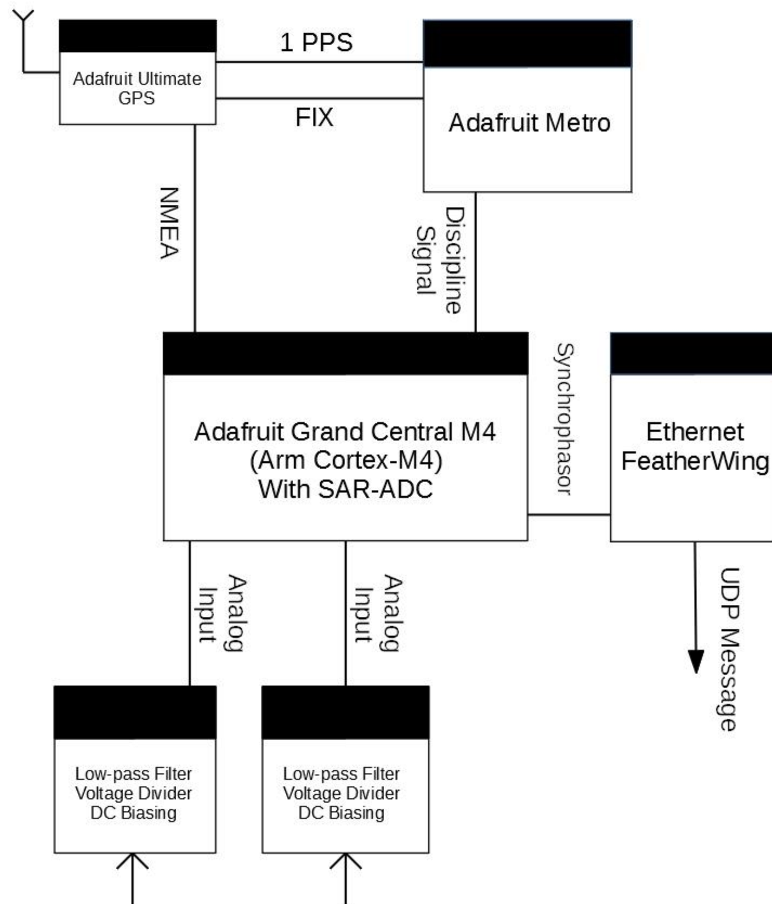

Figure 32: Actual System Under Test

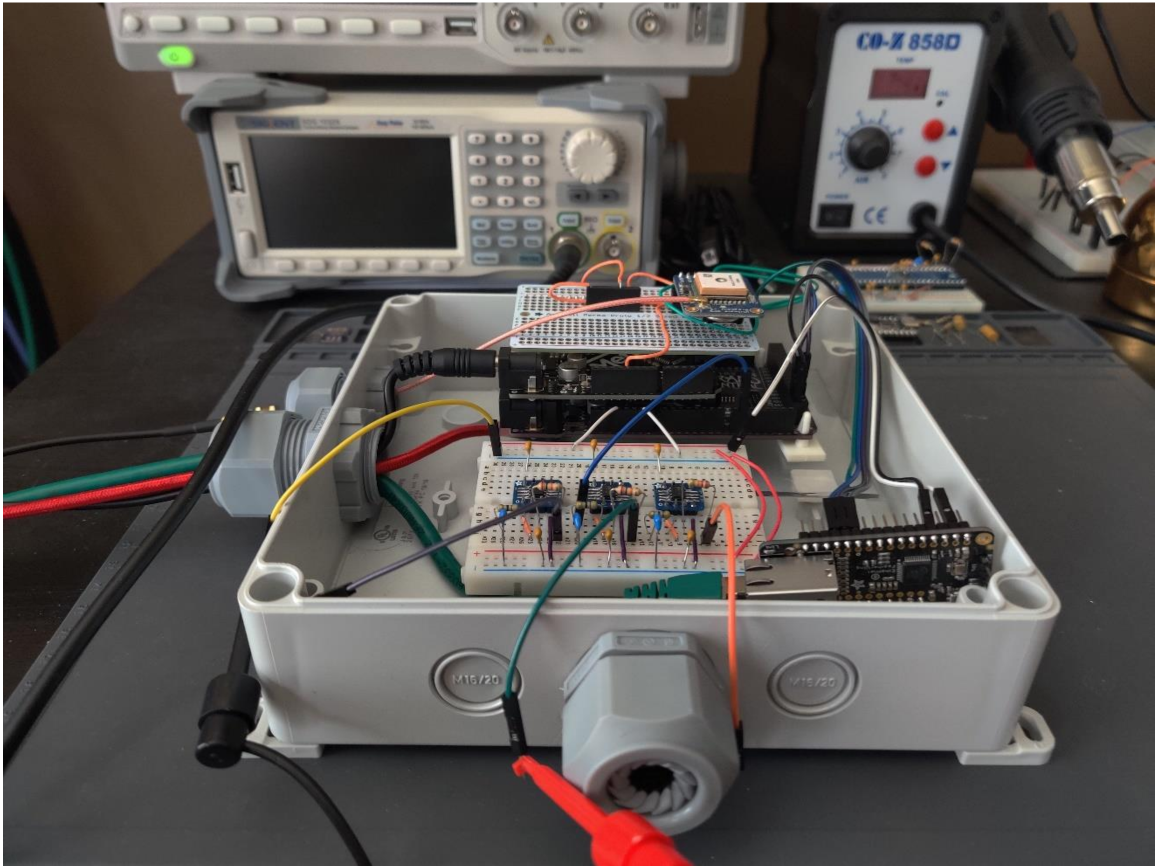

Figure 33: Photo of Actual Setup

CHAPTER 6: RESULTS & CONCLUSION

This project attempted to design a low-cost, microcontroller-based PMU which could accurately measure the magnitude, phase angle, frequency, and ROCOF. However, the project did not accomplish this goal, but it is on the right track.

When testing began, the UDP messages that were received seemed incorrect at a glance, mainly the phase angles seemed wrong. In order to check if it were the UDP messages that were incorrect, or if the data coming from the microcontroller was incorrect the, the phase angles for two channels were printed to the serial monitor of the Arduino IDE. The following phase angle graph was generated from the values copied in the serial monitor in Figure 34.

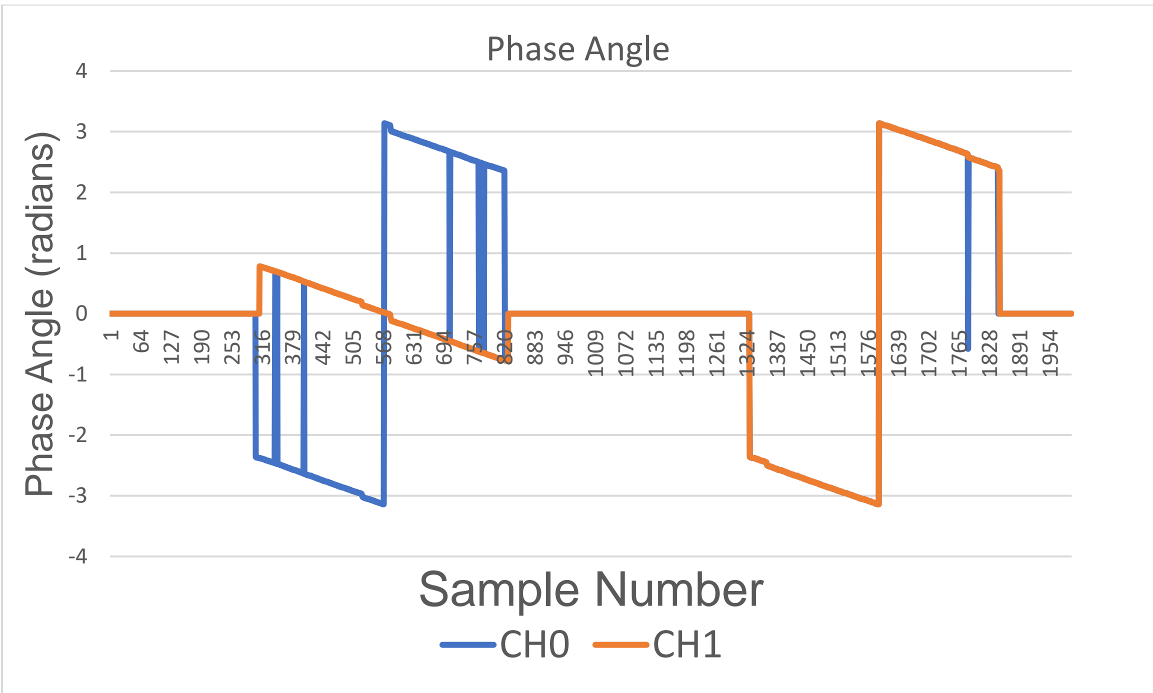

Figure 34: Phase angles Graph

For CH0 the phase angle was expected to be 0 but in Figure 33 the phase angles it sometimes zero but can spike to +/-1 or nearly +/-3 radians out of phase. Then, in an

attempt to recreate the problem, the whole system and function generator were restarted. The phase angle for just CH0 were then outputted to the serial plotter. The phase angles are still incorrect as seen in Figure 35.

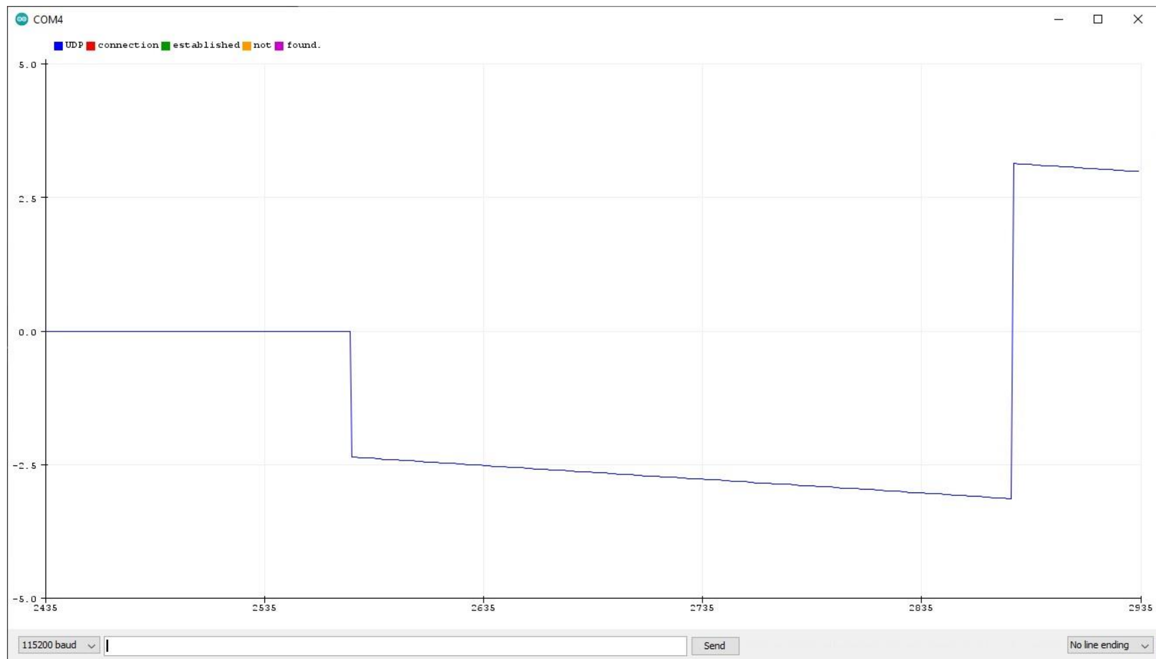

Figure 35: Phase Angle Channel 0 in Serial Monitor

To diagnosis the problem, the ADC values captured by CH0 are plotted on the serial plotter. Then each time a full capture of 128 ADC values is completed, a spike in value of 5 appears on the same graph of the CH0 captured values to indicate the end of the capture window, this is shown in Figure 36.

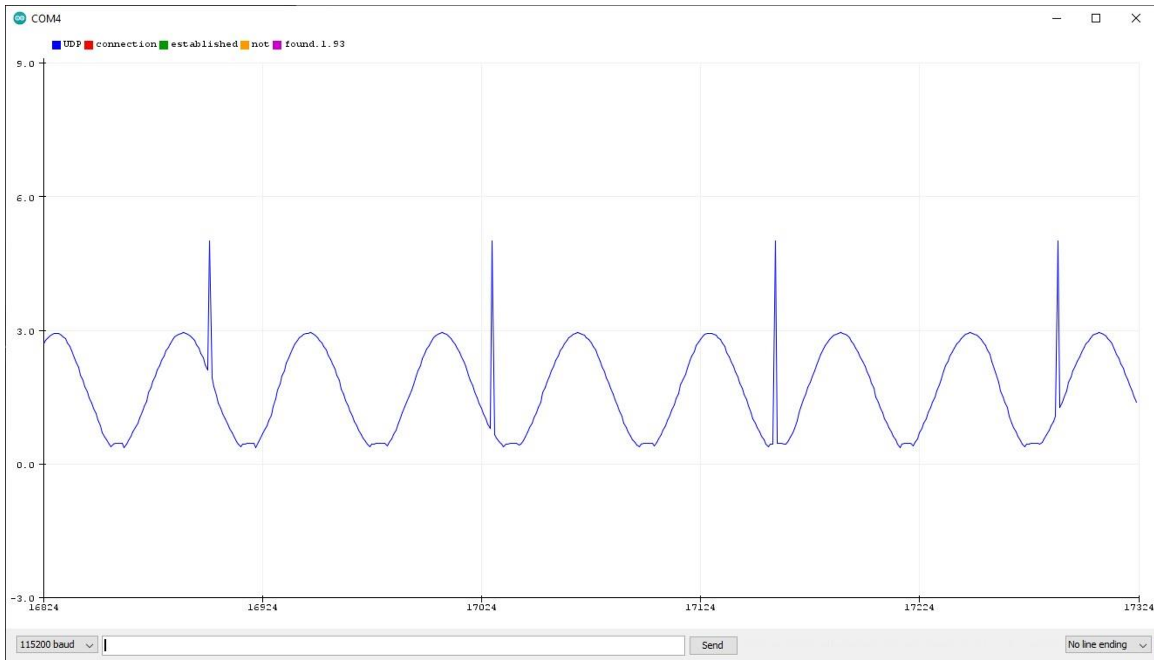

Figure 36: Values captured on ADC CH0

As seen in Figure 36, the sinusoid that is captured is changing between each full capture of 128 points or one period. One period could appear to be perfectly aligned with a cosine at the nominal frequency while another period could appear as a phase shifted version even if the signal from the function generator has no phase shift. This would explain the incorrect phase angle measurement Figures 34 and 35. In [8] it is mentioned that the phase estimation is sensitive to the sampling being consistent and accurate. This problem would also appear on the other channels since they are sampled sequentially. Figure 25 in the section Analog Interface and Antialiasing shows that on average there would be a 24 microsecond difference in time between captures with some variance in execution time. There could also be a time delay to the physical implementation which has the input signals going through breadboards. This could introduce extra capacitance or inductance, changing the input signal.

CHAPTER 7: IMPROVEMENTS AND FUTURE WORK

The end goal of the project to create a PMU and measure its accuracy was not achieved. Sampling of the input waveform is not accurate and the point on the waveform that is sampled changes every period. The waveform seen changing every period makes it look the signal is changing constantly to be a cosine that is phase shifted. This leads to inaccurate measurements of both magnitude and phase angle of the input signal. Accurate capture of a single period worth of data is important and has direct effects on the output. In order improve on this project, the ADC should be changed to one that allows for true simultaneous, parallel capture of multiple channels. Implementation would require more time and budget to properly implement and debug the software and hardware for it to be useable. DMA transfer of ADC data to memory may help but there would still be some delay on sampling of the other channels. Another improvement is on the generation of the sampling clock for the ADC. Currently a microcontroller that is triggered by the rising of a 1 PPS synchronized to match with UTC second is used. However, it would be more ideal to have a PLL, analog or digital, that is phase-locked with the 1PPS to provide a better sampling clock. Currently, if the PMU were to work it only measures voltages but it should be extended to include current measurements. The final improvement would be looking into a different form of communication aside from UDP, while UDP is fast it is also error prone. TCP would be a better form of communication since it ensures transmission of packets and ensures correctness through the use of checksums. This introduces more software overhead and would increase the amount of time used by the microcontroller to transmit. An idea would be to implement the microcontroller writing

the synchrophasor to a queue and having another microcontroller handle just communications.

REFERENCES

[1] “Grid Modernization and the Smart Grid,” *Energy.gov*. [Online]. Available: https://www.energy.gov/oe/activities/technology-development/grid-modernization-and-smart-grid. [Accessed: 11-Mar-2021].

[2] T. Bi, H. Liu, D. Zhang and Q. Yang, "The PMU dynamic performance evaluation and the comparison of PMU standards," 2012 IEEE Power and Energy Society General Meeting, San Diego, CA, USA, 2012, pp. 1-5, doi: 10.1109/PESGM.2012.6345328.

[3] M. Davis and S. Clemmer, “Power Failure: How Climate Change Puts Our Electricity at Risk and What We Can Do,” Union of Concerned Scientists, Cambridge, MA, Apr, 2014. Accessed on: Mar. 11, 2021. [Online]. Available: https://www.ucsusa.org/resources/power-failure

[4] “Smart Grid: The Smart Grid,” *Smart Grid: The Smart Grid | SmartGrid.gov*, 16-Dec-2019. [Online]. Available: https://www.smartgrid.gov/the_smart_grid/smart_grid.html. [Accessed: 11-Mar-2021].

[5] “About NASPI,” *North American SynchroPhasor Initiative*. [Online]. Available: https://www.naspi.org/. [Accessed: 11-Mar-2021].

[6] U.S. Department of Energy, “Synchrophasor Technologies and their Deployment in the Recovery Act Smart Grid Programs,” U.S. Department of Energy, Oak Ridge National Laboratory, Tech. Report, 31 July 2013 [Online]. Available: https://www.smartgrid.gov/document/synchrophasor_technologies_and_their_deployment_recovery_act_smart_grid_programs. [Accessed: 10-Mar-2021].

[7] D. M. Laverty, R. J. Best, P. Brogan, I. Al Khatib, L. Vanfretti and D. J. Morrow, "The OpenPMU Platform for Open-Source Phasor Measurements," in IEEE Transactions on Instrumentation and Measurement, vol. 62, no. 4, pp. 701-709, April 2013, doi: 10.1109/TIM.2013.2240920.

[8] X. Zhao, D. M. Laverty, A. McKernan, D. J. Morrow, K. McLaughlin and S. Sezer, "GPS-Disciplined Analog-to-Digital Converter for Phasor Measurement Applications," in IEEE Transactions on Instrumentation and Measurement, vol. 66, no. 9, pp. 2349-2357, Sept. 2017, doi: 10.1109/TIM.2017.2700158.

[9] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, 1st ed. San Diego, CA: California Technical Pub., 1997.

[10] M. Viswanathan, “Interpret FFT, complex DFT, frequency bins & FFTShift,” *GaussianWaves*, 16-Nov-2015. [Online]. Available: https://www.gaussianwaves.com/2015/11/interpreting-fft-results-complex-dft-frequency-bins-and-fftshift/. [Accessed: 10-Apr-2021].

[11] K. Moreland and E. Angel, “HWWS '03: Proceedings of the ACM SIGGRAPHEUROGRAPHICS conference on Graphics hardware,” in Proceedings of the ACM SIGGRAPHEUROGRAPHICS conference on Graphics hardware, 2003, pp. 112–119.

[12] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series", Math. Comput., vol. 19, no. 90, pp. 296-301, 1965.

[13] E. Analog Devices Inc., “Fast Fourier Transforms,” in *Mixed-Signal and DSP Design Techniques*, 1st ed., W. Kester, Ed. Newnes, 2002, pp. 5.a-5.26.

[14] J. Schloss, “FFT,” *FFT · Arcane Algorithm Archive*. [Online]. Available: https://www.algorithm-archive.org/contents/cooley_tukey/cooley_tukey.html. [Accessed: 12-Mar-2021].

[15] M. Viswanathan, “Interpret FFT results - obtaining magnitude and phase information,” *GaussianWaves*, 07-Nov-2020. [Online]. Available: https://www.gaussianwaves.com/2015/11/interpreting-fft-results-obtaining-magnitude-and-phase-information/. [Accessed: 10-Apr-2021].

[16] S. Rajan, Sichun Wang, R. Inkol and A. Joyal, "Efficient approximations for the arctangent function," in IEEE Signal Processing Magazine, vol. 23, no. 3, pp. 108-111, May 2006, doi: 10.1109/MSP.2006.1628884.

[17] N. Taylor, “How to Find a Fast Floating-Point atan2 Approximation,” *DSP Related*, 26-May-2017. [Online]. Available: https://www.dsprelated.com/showarticle/1052.php. [Accessed: 08-Apr-2021].

[18] *Measuring relays and protection equipment – Part 118-1: Synchrophasor for power systems – Measurements,* IEC/IEEE 60255-118-1, 2018

[19] GlobalTop Technology Inc., “GPS Standalone Module,” FGPMMOPA6H datasheet, Jan. 2013.

[20] Adafruit Ultimate GPS Breakout – 66 channel w/10 Hz updates – Version 3

[21] Chang Hong Technology Co., LTD., “GPS Active 28dB Magnetic Antenna+RG175(5M)+SMA Plug”, GPS Active 28dB Magnetic Antenna+RG175(5M)+SMA Plug Datasheet

[22] GPS Antenna – External Active Antenna – 3-5V 28dB 5 Meter SMA

[23] Adafruit Grand Central M4 Express featuring the SAMD51

[24] Microchip, "32-bit ARM Cortex-M4F MCUs with 1 Msps 12-bit ADC, QSPI, USB, Ethernet, and PTC," SAM D5x/E5x Family Data Sheet, July 2017 [Revised Jan. 2021].

[25] Analog Devices, "Quad input, 10-Output, Dual DPLL, 1 pps Synchronizer and Jitter Cleaner," AD9544 datasheet, Oct. 2017.

[26] AD9544 Functional Block Diagram

[27] K. Shirrif, "Secrets of Arduino PWM," *Ken Shirrif's Blog*, Jul-2009.

[28] Microchip, "8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash," ATmega328P Data Sheet, 2015 [Revised Sept. 2020].

[29] P. Horowitz and W. Hill, *The Art of Electronics*, 3rd ed. New York City, NY, USA: Cambridge University Press, 2015.

[30] B. Baker, "Optimize Your SAR ADC Design." Texas Instruments. [Online]. Available: https://e2e.ti.com/cfs-file/__key/communityserver-discussions-components-files/73/7384.4478.PA_2D00_001-Optimize_5F00_SAR_5F00_converter_5F00_design-REV-b.pdf [Accessed: 24-Mar-2021]

[31] Adafruit Ethernet FeatherWing

[32] WIZnet, "W5500," W5500 Datasheet, Aug. 2013 [Revised Apr. 2019].

APPENDIX A: TESTGPS.INO

```
#include <Adafruit_GPS.h>
#define GPSSerial Serial1
Adafruit_GPS GPS(&GPSSerial);

void setup() {
  while (!Serial); //Wait until Serial is ready
  Serial.begin(115200);

  GPS.begin(9600);
  GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCONLY);
  GPS.sendCommand(PMTK_SET_NMEA_UPDATE_1HZ);
  delay(1000);
  GPSSerial.println(PMTK_Q_RELEASE);
}

uint32_t timer = millis();
void loop() {
  char c = GPS.read();

  if (GPS.newNMEAreceived()) {
    Serial.println(GPS.lastNMEA());
    if (!GPS.parse(GPS.lastNMEA()))
      return;
  }

  // Approximately every 1 seconds or so, print out the current stats
  if (millis() - timer > 1000) {
    timer = millis();
    Serial.print("Time: ");
    if (GPS.hour < 10) { Serial.print('0'); }
    Serial.print(GPS.hour, DEC); Serial.print(':');
    if (GPS.minute < 10) { Serial.print('0'); }
    Serial.print(GPS.minute, DEC); Serial.print(':');
    if (GPS.seconds < 10) { Serial.print('0'); }
    Serial.print(GPS.seconds, DEC); Serial.print('.');
    if (GPS.milliseconds < 10) {
      Serial.print("00");
    } else if (GPS.milliseconds > 9 && GPS.milliseconds < 100) {
      Serial.print("0");
    }
    Serial.println(GPS.milliseconds);
    Serial.print("Date: ");
    Serial.print(GPS.day, DEC); Serial.print('/');
    Serial.print(GPS.month, DEC); Serial.print("/20");
```

```
    Serial.println(GPS.year, DEC);
    Serial.print("Fix: "); Serial.print((int)GPS.fix);
    Serial.print(" quality: "); Serial.println((int)GPS.fixquality);
    if (GPS.fix) {
      Serial.print("Location: ");
      Serial.print(GPS.latitude, 4); Serial.print(GPS.lat);
      Serial.print(", ");
      Serial.print(GPS.longitude, 4); Serial.println(GPS.lon);

      Serial.print("Speed (knots): "); Serial.println(GPS.speed);
      Serial.print("Angle: "); Serial.println(GPS.angle);
      Serial.print("Altitude: "); Serial.println(GPS.altitude);
      Serial.print("Satellites: "); Serial.println((int)GPS.satellites);
    }
  }
}
```

APPENDIX B: TESTADC.INO

```
#define TRUE 1
#define FALSE 0
bool OVERSAMPLE_FLAG = TRUE;

int nSamples = 100000;  //Number of test samples to collect

void analogReadOversample16(bool isOversampleRequested) {
  if(isOversampleRequested) {
    ADC0->AVGCTRL.reg = ADC_AVGCTRL_SAMPLENUM_16|
    ADC_AVGCTRL_ADJRES(4);
    ADC1->AVGCTRL.reg = ADC_AVGCTRL_SAMPLENUM_16 |
    ADC_AVGCTRL_ADJRES(4);
    while(ADC0->SYNCBUSY.reg & ADC_SYNCBUSY_AVGCTRL);
    while(ADC1->SYNCBUSY.reg & ADC_SYNCBUSY_AVGCTRL);
  }
  else {
    ADC0->AVGCTRL.reg = ADC_AVGCTRL_SAMPLENUM_1 |
    ADC_AVGCTRL_ADJRES(0);
    ADC1->AVGCTRL.reg = ADC_AVGCTRL_SAMPLENUM_1 |
    ADC_AVGCTRL_ADJRES(0);
    while(ADC0->SYNCBUSY.reg & ADC_SYNCBUSY_AVGCTRL);
    while(ADC1->SYNCBUSY.reg & ADC_SYNCBUSY_AVGCTRL);
  }
}

void setup() {
  Serial.begin(9600);
  analogReference(AR_DEFAULT);  //Set ADC AREF to 3.3V
  analogReadResolution(12);     //Set ADC bit resolution to 12
}

void loop() {
  unsigned long starttime, endtime, elapsedtime = 0;
  if(OVERSAMPLE_FLAG) {
    analogReadOversample16(OVERSAMPLE_FLAG);
    Serial.println("Execution time of analogRead with Oversampling (microseconds)");
  }
  else if(!OVERSAMPLE_FLAG) {
    analogReadOversample16(OVERSAMPLE_FLAG);
    Serial.println("Execution time of analogRead without Oversampling (microseconds)");
  }
  else {
    return;
  }
```

```
  for(int i = 0; i < nSamples; i++) {
    starttime = micros();
    analogRead(A0);
    endtime = micros();
    elapsedtime = endtime - starttime;
    Serial.println(elapsedtime);
  }

  Serial.println("========================== TEST COMPLETE
==========================");
}
```

APPENDIX C: TESTDFT.INO

```
#include <arm_math.h>
#include <arm_const_structs.h>
#include <arm_common_tables.h>
#include <math.h>

const uint16_t fftSize = 128;  //FFT length
uint8_t ifftFlag = 0;          //Inverse-transform flag
uint32_t doBitReverse = 1;     //Bit Reversal Flag
int num_samples = 100000;      //Number of desired sample points
const int buttonPin = 13;
int buttonState = 0;

static float32_t FFTOutput[fftSize];  //Output array holding FFT results

//There's 128 points from 1 period of 60 Hz, 5 Vp-p sine wave
//Data from Noisy_Vref5_7.68kHz_30dBW
float32_t testInput_f32_7680Hz[128] = {0};     //Values omitted for brevity

//There's 256 points from 1 period of 60 Hz, 5 Vp-p sine wave
//Data from Noisy_Vref5_15.36kHz_30dBW
float32_t testInput_f32_153600Hz[256] = {0};  //Values omitted for brevity

void setup() {
  pinMode(buttonPin, INPUT);
}

void loop() {
  buttonState = digitalRead(buttonPin);

  if(buttonState == HIGH) {
    for(int i = 0; i < num_samples; i++) {
      arm_rfft_fast_instance_f32 arm_rfft;
      arm_rfft_fast_init_f32(&arm_rfft, fftSize);

      unsigned long start = micros();
      arm_rfft_fast_f32(&arm_rfft, testInput_f32_7680Hz, FFTOutput, ifftFlag);
      unsigned long endtime = micros() - start;

      Serial.print(endtime);
      Serial.print('\n');
    }
  } else {
  }
}
```

APPENDIX D: TESTADC_V2.INO

```
#define TRUE 1
#define FALSE 0
bool OVERSAMPLE_FLAG = TRUE;

int nSamples = 100000;  //Number of test samples to collect

void adjustTSAMP() {
  ADC0->SAMPCTRL.reg = 1;   //Adjust sampling length
  ADC1->SAMPCTRL.reg = 1;

  while(ADC0->SYNCBUSY.reg);
  while(ADC1->SYNCBUSY.reg);
}

void analogReadOversample16(bool isOversampleRequested) {
  if(isOversampleRequested) {
    ADC0->AVGCTRL.reg = ADC_AVGCTRL_SAMPLENUM_16|
    ADC_AVGCTRL_ADJRES(4);
    ADC1->AVGCTRL.reg = ADC_AVGCTRL_SAMPLENUM_16 |
    ADC_AVGCTRL_ADJRES(4);
    while(ADC0->SYNCBUSY.reg & ADC_SYNCBUSY_AVGCTRL);
    while(ADC1->SYNCBUSY.reg & ADC_SYNCBUSY_AVGCTRL);
  }
  else {
    ADC0->AVGCTRL.reg = ADC_AVGCTRL_SAMPLENUM_1 |
    ADC_AVGCTRL_ADJRES(0);
    ADC1->AVGCTRL.reg = ADC_AVGCTRL_SAMPLENUM_1 |
    ADC_AVGCTRL_ADJRES(0);
    while(ADC0->SYNCBUSY.reg & ADC_SYNCBUSY_AVGCTRL);
    while(ADC1->SYNCBUSY.reg & ADC_SYNCBUSY_AVGCTRL);
  }
}

void setup() {
  Serial.begin(9600);
  analogReference(AR_DEFAULT);  //Set ADC AREF to 3.3V
  analogReadResolution(12);     //Set ADC bit resolution to 12
}

void loop() {
  adjustTSAMP();
  unsigned long starttime, endtime, elapsedtime = 0;
  if(OVERSAMPLE_FLAG) {
    analogReadOversample16(OVERSAMPLE_FLAG);
```

```
    Serial.println("Execution time of analogRead with Oversampling (microseconds)");
  }
  else if(!OVERSAMPLE_FLAG) {
    analogReadOversample16(OVERSAMPLE_FLAG);
    Serial.println("Execution time of analogRead without Oversampling (microseconds)");
  }
  else {
    return;
  }

  for(int i = 0; i < nSamples; i++) {
    starttime = micros();
    analogRead(A0);
    endtime = micros();
    elapsedtime = endtime - starttime;
    Serial.println(elapsedtime);
  }

  Serial.println("========================== TEST COMPLETE
==========================");
}
```

APPENDIX E: TESTPWM.INO

```
int out_pin = 10;   // OC1A/OC1B are on Metro Pins 9 & 10

void setup() {
  // put your setup code here, to run once:
  pinMode(out_pin, OUTPUT);
  TCCR1A = B10100011;   // Timer/Counter1 Control Reg A
  TCCR1B = B00011001;   // Timer/Counter1 Control Reg B
  TCCR1C = B00000000;   // Timer/Counter1 Control Reg C
  OCR1A = 2082;    // Output Compare Register 1A; FastPWM holds output high plus
one cycle
}

void loop() {}
```

APPENDIX F: PMU.INO

```
#include <Adafruit_GPS.h>
#include <Ethernet.h>
#include <SPI.h>
#include <EthernetUdp.h>
#include <arm_const_structs.h>
#include <arm_common_tables.h>
#include <arm_math.h>
#include <math.h>

#define GPSSerial Serial1
Adafruit_GPS GPS(&GPSSerial);

#define NUMCHANNELS 3              // Number of ADC channels in use
#define INT2VOLT 0.00080586        // Conversion factor from ADC 12-bit value to
floating point
#define TIMEBTWNPULSES 0.000130208  // Time between ADC sampling pulses
#define FREQRES 60                 // Frequency resolution of FFT
#define NOMFREQ 60                 // Nominal frequency of the input signals
#define MIDPOINT 128/2             // Midpoint of a period for a given signal
#define pi 3.14159265
#define PIOVER4 pi/4

inline float32_t derivative(float32_t y1, float32_t y2, uint32_t x1, uint32_t x2 ) {
  return (y2-y1)/(x2-x1);
}

// ========== Ethernet-related Variables and Object(s) ==========
byte MAC[] = {
  0x98, 0x76, 0xB6, 0x11, 0xB0, 0x97
};
IPAddress IP(192, 168, 1, 218);
IPAddress DestIP(192, 168, 1, 217);
unsigned int localPort = 8888;
char ethernetCSpin = 21;
char OutBuffer[256];
EthernetUDP UDP;

// ========== Interrupt-related Variables ==========
const byte interruptPin = 7;      // Digital Pin used to check for external interrupt signal
volatile uint8_t n = 0;           // Keeps track of number of samples sets collected
volatile uint8_t m = 0;           // Keeps track of which set of datapoints currently
sampling (Max = 60)
volatile bool hasSampledPeriod = false;
volatile byte sampSet = 1;        // Keeps track of which sampData_CHx_x to write to
```

```
// ========== FFT-related Variables ==========
const uint16_t fftSize = 128;     // FFT length
const uint8_t ifftFlag = 0;       // Inverse-transform flag

// Sampled data for each CHx
// There are two sets; one is used to keep track of sampled data for a period
// The other set is used in actual FFT calculation
static float32_t sampData_CH0_1[fftSize] = {0};
static float32_t sampData_CH1_1[fftSize] = {0};
static float32_t sampData_CH2_1[fftSize] = {0};

static float32_t sampData_CH0_2[fftSize] = {0};
static float32_t sampData_CH1_2[fftSize] = {0};
static float32_t sampData_CH2_2[fftSize] = {0};

// FFT Output Buffer for each channel
static float32_t fftOutput_CH0[fftSize] = {0};
static float32_t fftOutput_CH1[fftSize] = {0};
static float32_t fftOutput_CH2[fftSize] = {0};

// Complex Magnitude for each channel
// When FFT is calculated and stored in fftOutput_CHx
// the data is formatted as: fftOutput_CHx = {real[0], imag[0], ... real[(N/2)-1],
imag[(N/2)-1]}
// when the complex magnitude is calculated one pair of real and imag portions are used
// leaving one data point, this is why the cmplxMag_CHx is of size fftSize/2
static float32_t cmplxMag_CH0[fftSize/2] = {0};
static float32_t cmplxMag_CH1[fftSize/2] = {0};
static float32_t cmplxMag_CH2[fftSize/2] = {0};

// Pointer declaration and initilizations for sampled data arrays
// Initialized to the first set of arrays
float32_t *pSampData_CH0 = &sampData_CH0_1[0];
float32_t *pSampData_CH1 = &sampData_CH1_1[0];
float32_t *pSampData_CH2 = &sampData_CH2_1[0];

// ========== PMU Struct ==========
struct PMU_output {
  float32_t magnitude;      // Amplitude of signal
  float32_t phase[2] = {0}; // Phase angle of signal; phase[0] is the previous angle,
phase[1] is the newest angle.
  float32_t ROCOF;          // Rate of Change of Frequency
  float32_t Real;           // Real component
  float32_t Imag;           // Imaginary component
  uint32_t UTC[2] = {0};    // Timestamp
  float32_t microseconds;   // Microsecond timestamp, GPS doesn't give this accuracy.
```

```
  uint32_t date;            // Current date (Format: DD,MM,YY)
  uint32_t gps_latitude;    // Current latitude
  byte gps_lat;             // East (E) or West (W)
  uint32_t gps_longitude;   // Current longitude
  byte gps_lon;             // North (N) or South (S)
} synch_CH0, synch_CH1, synch_CH2;

static PMU_output synch_CH[NUMCHANNELS] = {synch_CH0, synch_CH1,
synch_CH2}; // Array of PMU_output structs

// ========== ADC Adjustment Function ==========
// adjustTSAMP()
// Used to adjust sampling period of ADC
// Adafruit/Arduino hardcode sampling period to be 4 cycles of clock
void adjustTSAMP() {
  // Adjusting sampling period for each ADC
  ADC0->SAMPCTRL.reg = 1;
  while(ADC0->SYNCBUSY.reg);
  ADC1->SAMPCTRL.reg = 1;
  while(ADC1->SYNCBUSY.reg);
}

// ========== Atan/Atan2 Calculation Function(s) ==========
float32_t ApproxAtan(float32_t phi) {
  float32_t absphi = 0;
  if( phi < 0 ) {
    absphi = -phi;
  }
  absphi = phi;
  return (PIOVER4*phi)+(0.273*phi)-absphi;
}

float32_t ApproxAtan2(float32_t y, float32_t x) {
  if( x != 0.0 ) {
    if( abs(x) > abs(y) ) {
      const float32_t z = y / x;
      if( x > 0.0 ) {
        return ApproxAtan(z);
      }
      else if( y >= 0.0 ) {
        return ApproxAtan(z) + pi;
      }
      else {
        return ApproxAtan(z) - pi;
      }
    }
```

```
  }
  else {
    if( y > 0.0 ) {
      return pi/2;
    }
    else if( y < 0.0 ) {
      return -pi/2;
    }
  }
  return 0.0;
}

void setup() {
  while(!Serial);
  Serial.begin(115200);
  GPS.begin(9600);
  GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCONLY);
  GPS.sendCommand(PMTK_SET_NMEA_UPDATE_1HZ);
  delay(1000);

  // Sets digital pin 7 as input
  pinMode(interruptPin, INPUT);
  // Attachs interrupt to digital pin 7, calls SAMPLE function as rising edge.
  attachInterrupt(digitalPinToInterrupt(interruptPin), SAMPLE, RISING);

  Ethernet.init(ethernetCSpin);
  Ethernet.begin(MAC, IP);
  UDP.begin(localPort);
}

//=======================Main loop =======================
void loop() {
  char i = 0;   // Iterator variable
  // Reads a new character from the GPS
  char NMEAmsg = GPS.read();
  // Keep checking if a new NMEA messaged as been received
  if( GPS.newNMEAreceived() ) {
    // If a NMEA message is successfully parsed, update synchrophasor variables.
    if( GPS.parse(GPS.lastNMEA()) ) {
      // Update to the new UTC timestamp
      synch_CH0.UTC[1] = GPS.fullUTC;
      for(i = 1; i < NUMCHANNELS; i++) {
        synch_CH[i].UTC[1] = synch_CH[0].UTC[1];
      }

      // Update the date if it changes
```

```
    if(synch_CH[0].date != GPS.fulldate) {
      synch_CH[0].date = GPS.fulldate;
      for(i = 1; i < NUMCHANNELS; i++) {
        synch_CH[i].date = synch_CH[0].date;
      }
    }

    // Update the latitude if it changes
    if(synch_CH[0].gps_latitude != GPS.latitude) {
      synch_CH[0].gps_latitude = GPS.latitude;
      for(i = 1; i < NUMCHANNELS; i++) {
        synch_CH[i].gps_latitude = synch_CH[0].gps_latitude;
      }
    }

    // Update the latitude cardinal direction if it changes
    if(synch_CH[0].gps_lat != GPS.lat) {
      synch_CH[0].gps_lat = GPS.lat;
      for(i = 1; i < NUMCHANNELS; i++) {
        synch_CH[i].gps_lat = synch_CH[0].gps_lat;
      }
    }

    // Update the longitude if it changes
    if(synch_CH[0].gps_longitude != GPS.longitude) {
      synch_CH[0].gps_longitude = GPS.longitude;
      for(i = 1; i < NUMCHANNELS; i++) {
        synch_CH[i].gps_longitude = synch_CH[0].gps_longitude;
      }
    }

    // update the longitude cardinal direction if it changes
    if(synch_CH[0].gps_lon != GPS.lon) {
      synch_CH[0].gps_lon = GPS.lon;
      for(i = 1; i < NUMCHANNELS; i++) {
        synch_CH[i].gps_lon = synch_CH[0].gps_lon;
      }
    }
  }
}

if (hasSampledPeriod) {
  hasSampledPeriod = false;

  float32_t maxValue_CH0, maxValue_CH1, maxValue_CH2 = 0;
  uint32_t maxIdx_CH0, maxIdx_CH1, maxIdx_CH2 = 0;
```

```
   arm_rfft_fast_instance_f32 arm_rfft;
   arm_rfft_fast_init_f32(&arm_rfft, fftSize);   // Initializes rfft for length of fftSize

   if( sampSet == 1 ){
     arm_rfft_fast_f32(&arm_rfft, sampData_CH0_2, fftOutput_CH0, ifftFlag);
     arm_rfft_fast_f32(&arm_rfft, sampData_CH1_2, fftOutput_CH1, ifftFlag);
     arm_rfft_fast_f32(&arm_rfft, sampData_CH2_2, fftOutput_CH2, ifftFlag);
   }
   else if( sampSet == 2 ){
     arm_rfft_fast_f32(&arm_rfft, sampData_CH0_1, fftOutput_CH0, ifftFlag);
     arm_rfft_fast_f32(&arm_rfft, sampData_CH1_1, fftOutput_CH1, ifftFlag);
     arm_rfft_fast_f32(&arm_rfft, sampData_CH2_1, fftOutput_CH2, ifftFlag);
   }

   // Calculation of complex magnitude
   arm_cmplx_mag_f32(fftOutput_CH0, cmplxMag_CH0, fftSize/2);
   arm_cmplx_mag_f32(fftOutput_CH1, cmplxMag_CH1, fftSize/2);
   arm_cmplx_mag_f32(fftOutput_CH2, cmplxMag_CH2, fftSize/2);

   // Calculation of max value and its corresponding freq. index
   arm_max_f32(cmplxMag_CH0, fftSize/2, &maxValue_CH0, &maxIdx_CH0);
   arm_max_f32(cmplxMag_CH1, fftSize/2, &maxValue_CH1, &maxIdx_CH1);
   arm_max_f32(cmplxMag_CH2, fftSize/2, &maxValue_CH2, &maxIdx_CH2);

   synch_CH0.magnitude = maxValue_CH0/fftSize;
   synch_CH0.phase[1] = ApproxAtan2(fftOutput_CH0[2*maxIdx_CH0+1],
fftOutput_CH0[2*maxIdx_CH0])*180/pi;
   synch_CH1.magnitude = maxValue_CH1/fftSize;
   synch_CH1.phase[1] = ApproxAtan2(fftOutput_CH1[2*maxIdx_CH1+1],
fftOutput_CH1[2*maxIdx_CH1])*180/pi;
   synch_CH2.magnitude = maxValue_CH2/fftSize;
   synch_CH2.phase[1] = ApproxAtan2(fftOutput_CH2[2*maxIdx_CH2+1],
fftOutput_CH2[2*maxIdx_CH1])*180/pi;

   for(i = 0; i < NUMCHANNELS; i++) {
     synch_CH[i].ROCOF = derivative(synch_CH[i].phase[1], synch_CH[i].phase[0],
synch_CH[i].UTC[1], synch_CH[i].UTC[0]);    // Calculation of ROCOF
     synch_CH[i].phase[0] = synch_CH[i].phase[1];
// The "current" phase becomes the "previous" phase for the next calculation
     synch_CH[i].microseconds = (MIDPOINT*TIMEBTWNPULSES)*(m+1);
// Microseconds timestamp for synchrophasor
   }

   Serial.println(synch_CH0.phase[1])
```

```
//==================Interrupt Service Routine =====================
// Function that samples each channel in use on the ADC sequentially
// Called everytime their is a rising signal on the interrupt pin
void SAMPLE() {
  // Saves analogRead values per channel to appropriate array in memory using pointers
  // Raw ADC value is converted to floating point before writing to array
  *(pSampData_CH0 + n) = analogRead(A0)*INT2VOLT;
  *(pSampData_CH1 + n) = analogRead(A1)*INT2VOLT;
  *(pSampData_CH2 + n) = analogRead(A2)*INT2VOLT;

  // Sets flag to TRUE when a full period sample has been acquired
  // This initiates FFT calculation after interrupt function is complete
  if( (n+1) % fftSize == 0 ) {
    m = m + 1;
    hasSampledPeriod = true;
  }
  else {
    hasSampledPeriod = false;
  }

  m = (m == (NOMFREQ-1)) ? 0 : m;    // Resets tracking of sets

  // Writes data sequentially until an array limit is reached, then arrays are switched.
  if( n < (fftSize)-1 ) {
    n = n + 1;
  }
  else {
    n = 0;  // Resets counter
    if(sampSet == 1) {
      pSampData_CH0 = &sampData_CH0_2[0];
      pSampData_CH1 = &sampData_CH1_2[0];
      pSampData_CH2 = &sampData_CH2_2[0];
      sampSet = 2;  // Designates set 2 as the arrays used in sampling data
    }
    else if(sampSet == 2) {
      pSampData_CH0 = &sampData_CH0_1[0];
      pSampData_CH1 = &sampData_CH1_1[0];
      pSampData_CH2 = &sampData_CH2_1[0];
      sampSet = 1;  // Designates set 1 as the arrays used in sampling data
    }
  }
}
```

APPENDIX G: GENSAMPLECLK.INO

```
// OC1A/OC1B are on Metro Pins 9 & 10

#define HZ7680 2082
#define HZ3840 4163

static int interruptPin_1PPS = 2;

void setup() {
  Serial.begin(9600);
  pinMode(interruptPin_1PPS, INPUT);
  pinMode(9, OUTPUT);
  pinMode(10, OUTPUT);
  attachInterrupt(digitalPinToInterrupt(interruptPin_1PPS), genSamplingSignal,
RISING);
}

void loop() {}

// Generate ADC discplining signal
void genSamplingSignal() {
  //Serial.println("Beginning Generation of ADC Disciplining Signal. . .");
  TCCR1A = B10100011;   // Timer/Counter1 Control Reg A
  TCCR1B = B00011001;   // Timer/Counter1 Control Reg B
  OCR1A = HZ7680;    // Output Compare Register 1A; FastPWM holds output high plus
one cycle
}
```