

Passive Aircraft Radar

Team 5 - Rushik Desai, Benjamin Du, Kyle Rogers
ECE 445 Final Report - Spring 2019
Professor: Michael Oelze, TA: Kyle Michal

Abstract

This report contains the design and implementation of a passive radar receiver which, when connected in a network of four receivers, uses time delay of arrival (TDOA) to calculate the geographical position of an aircraft. It also contains an analysis of the parts which were successfully implemented, and the issues encountered during the verification stage. The receiver captures, amplifies, and decodes the aircraft message through an RF front end. This message is verified, and a timestamp is created. The receiver was designed to be low-cost, powered by power over ethernet (POE), and to have an accuracy of 100m.

**Thank you to Professor Kirill Levchenko
for sponsoring this project!**

Table of Contents

1 Introduction	3
1.1 Purpose	3
1.2 Background	3
1.3 Functionality	3
1.4 Subsystem Overview	4
1.4.1 RF Front End	4
1.4.2 Control Unit	4
1.4.3 GPS Module	4
1.4.4 Power Module	5
1.4.4 Host and Server	5
2 Design	6
2.1 RF Front End	6
2.1.1 Band Pass Filter	7
2.1.2 Low Noise Amplifier	7
2.1.3 Logarithmic Amplifier	8
2.1.4 Voltage Comparator	8
2.2 Control Unit	9
2.2.1 GPS Module	10
2.2.2 Timing Module	10
2.2.3 Microcontroller	10
2.2.4 Raspberry Pi	11
2.2.5 Software	12
2.3 Power Module	12
3 Design Verification	13
3.1 RF Front End Verification	13
3.2 GPS Verification	16
3.3 Control Unit Verification	17
4 Costs	18
5 Conclusion	19
5.1 Accomplishments	19
5.2 Problems	19
5.3 Uncertainties	19
5.4 Ethical Considerations	20
5.5 Future Work	20
References	21
Appendix A - Requirements and Verification	23
Appendix B - Project Schematics and Layout	28
Appendix C - Software Flowchart	31
Appendix D - Software Code	33
D.1 Timestamp Code	33
D.2 GPS Code	36
D.3 Raspberry Pi Read Code	40
Appendix E - TDOA Calculation and Implementation in MATLAB	41
Appendix F - Omitted Design Implementations	44
F.1 Low-profile Antenna	44
F.2 PIC32 Microcontroller	44

1 Introduction

The major markets for passive radar technologies are military, commercial airports, and flight tracking services. Military and commercial applications are sophisticated, but expensive. Commercial applications do exist, and range from personal projects to subscription based services. Most of these applications rely on decoding ADS-B/Mode-S transponder data to identify aircraft and determine altitude. This method, while effective, is vulnerable to spoofing for air-to-ground communication [1]. Passive radar technology is also relatively new. Locating an aircraft is just one of its applications. In general, time delay of arrival (TDOA) can be used to locate any transmitter.

1.1 Purpose

We propose to create an affordable, accessible solution that does not depend on transponder data to get accurate aircraft positioning. The solution shall consist of four receivers that together form a radar network. Instead of using ADS-B signals like other solutions [2], the network will strictly use multilateration (MLat) to determine an aircraft's position. The receivers shall be 'plug-and-play', requiring only a Power over Ethernet (PoE) connection to be used in our network. Our solution will be modular so that it can be used to locate the position of any signal by modifying its front end.

1.2 Background

Radar technology has existed since the beginning of World War II. Traditional radar systems use a transmitter to send an electromagnetic (EM) wave, and read the received reflected wave to compute the position of an object. In recent years, research of passive radar technology has increased. A passive radar does not transmit, and only uses existing EM waves of the target to locate an object [3]. The military is researching passive radar solutions to detect stealth aircraft, but these solutions require a heavy amount of signal processing, and can be costly to implement [4]. There are many hobbyist solutions which use a passive receiver to detect aircraft, but these implementations often use external websites to interpret the received data, and designs which are inexpensive are not very accurate [5].

1.3 Functionality

We centered our design around five objectives which are listed below.

1. Capture and demodulate transponder signal
2. Estimate aircraft position using TDOA
3. Generate a timestamp per capture event and synchronize receivers with common time reference
4. Power design using Power over Ethernet (PoE)
5. Create a low-cost design (under \$100)

From these objectives, we created three high level requirements which would act as criteria for a successful project. These requirements are shown below.

1. This project shall have a cost below \$100 per receiver, excluding the costs of the server and PoE injector.
2. This project shall be able to detect an aircraft position with an accuracy of at least 100 m in 3D space.
3. Each receiver shall consume less than 15.4 W, in accordance with IEEE 802.3af standards [6].

1.4 Subsystem Overview

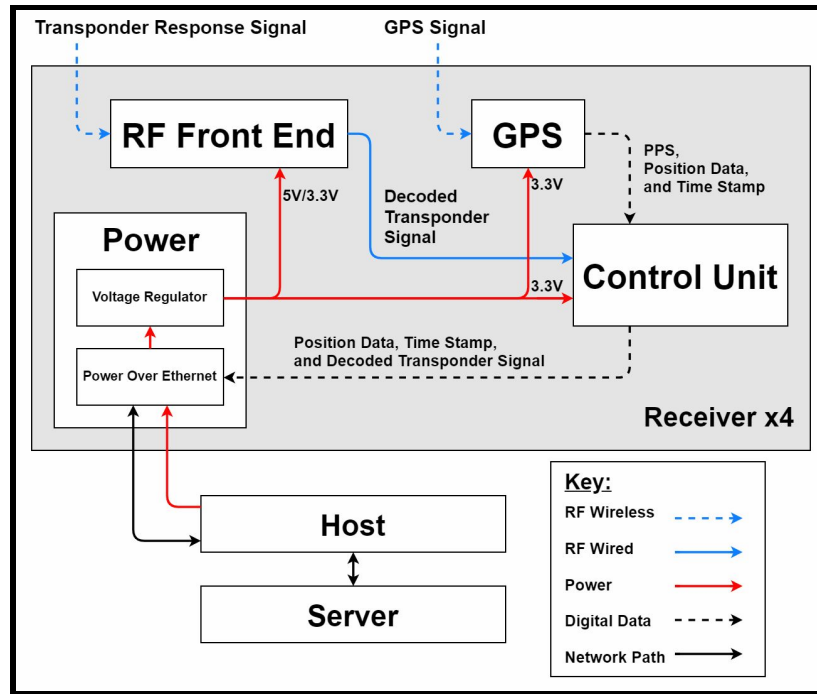


Figure 1.1: High Level Diagram of a Single Receiver

Shown in Fig 1.1 is the high level diagram for a single receiver. Each block in this diagram is described below.

1.4.1 RF Front End

The RF front end first captures and amplifies the aircraft transponder signal. Next, it filters out undesired frequencies. Finally, the signal is demodulated, turned into a digital signal, and is sent to the control unit.

1.4.2 Control Unit

The control unit supports the overall functionality of the receiver's ability to react to the detection of a transponder signal from the RF Front End and deliver time data to the server for processing the airplane location using TDOA. Minimal latency and high reliability of detecting the transponder signals are key concerns for maintaining accuracy. Furthermore, the unit needs to handle configuration settings for ethernet communication and update the server with its location.

1.4.3 GPS Module

The GPS module will provide positional data for the receiver and a synchronized time reference. This time reference is crucial for creating the timestamp for the TDOA.

1.4.4 Power Module

The power module uses PoE along with voltage regulators in order to power the active components of our receiver.

1.4.4 Host and Server

The host receives the timestamp created by the control unit, and the position data of the receiver received by the GPS module. This data is then sent to the central server, which performs the TDOA calculation, and computes the position of the aircraft. Although we would have liked to implement a server and host, we instead used a Raspberry Pi in order to extract data from the control unit due to time constraints.

2 Design

2.1 RF Front End

Aircraft transponder signals are transmitted using a 1090 MHz carrier signal, and use on/off keying for their modulation scheme [7]. To design the RF front end, we first needed to calculate the receive power of the transponder signal. This was done using the Friis transmission formula (shown in Eq 1.1 below).

$$P_R = \frac{G_T G_R \lambda^2}{(4\pi R)^2} P_T \quad (1.1)$$

The transponder transmitter power P_T was estimated as 250 W as declared by the FAA [8], the distance between the receiver and the transponder R was estimated as 12 km, and the gains of the antenna and receiver were set to 1 (in reality the gains are not 1, but both antennas will have relatively low gain [9] so for the purposes of this calculation we will assume that our antennas are isotropic). Since the transponder signal transmits at 1090 MHz, we can calculate the wavelength $\lambda = c/1090 \text{ MHz} = 0.275 \text{ m}$. With these parameters, we calculated the received power as $8.33 \times 10^{-7} \text{ mW} = -60.8 \text{ dBm}$ for an aircraft 12 km directly above a receiver.

Since the modulation scheme of the transponder signal was on/off keying, we decided to use a logarithmic amplifier to demodulate the received signal. We then used a comparator to convert the demodulated signal into a digital signal.

We chose a power of -10 dBm to be sent into the logarithmic amplifier. This resulted in required gain of 50 dB, which we chose to implement in two stages using low noise amplifiers (LNAs). We included band pass filters (BPFs) in order to filter out non-linearities created by the LNAs and other unwanted frequencies. The resulting block diagram is shown in Fig 2.1 below.

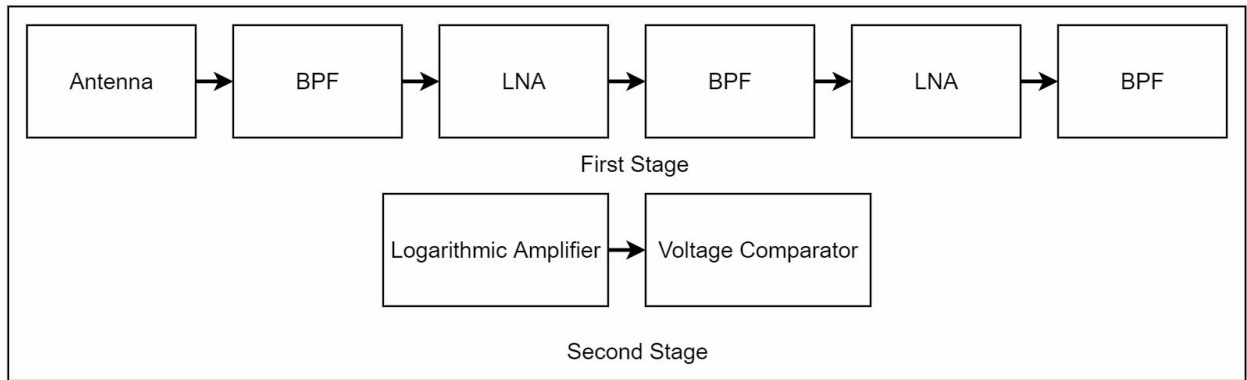


Figure 2.1: Block Diagram of the RF Front End

Each component of the RF front end is described in further detail below. We chose components which had a low propagation delays, low rise times, and which were power efficient in order to meet our design objectives.

2.1.1 Band Pass Filter

We chose to use the TA1090EC as our BPF, as its center frequency was 1090 MHz, and had a low insertion loss (less than 2.5 dB). Figure 2.2 contains the S21 of the filter as given by its datasheet. The TA1090EC was chosen over other BPFs because similar filters were either much more expensive or unavailable.

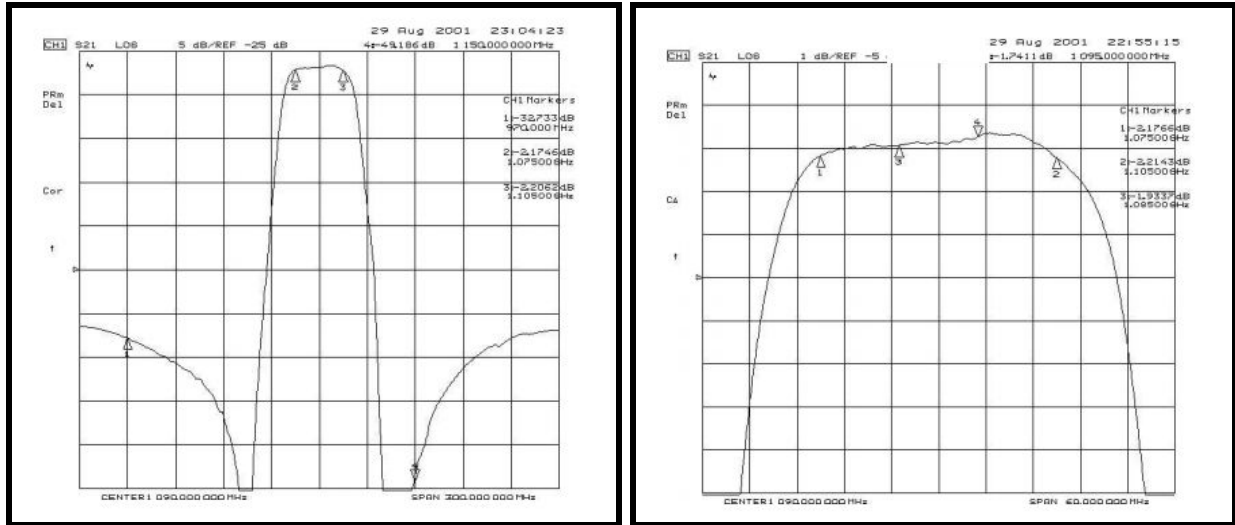


Figure 2.2: S21 Measurements for TA1090EC BPF (200 and 50 MHz span) [10]

2.1.2 Low Noise Amplifier

The MAAL-011078 was chosen as the LNA as it had a very low noise figure (less than 0.5 dB), consumed a low amount of power (165 mW when driven at 3.3 V and 50 mA), and a high gain (27 dB) [11]. With two gain stages and three filter stages, the total gain of the BPFs and LNAs would theoretically equal about 50 dB. Other LNAs did not have as low of a noise figure, and consumed more power. We wanted to have as low of a noise figure as possible in order to ensure that the captured transponder signal would not be distorted. The LNAs were used as shown in Fig 2.3 below (specified by the datasheet). R_{BIAS} was set to 680 Ω in order to ensure that the LNA would draw about 50 mA of current when supplied 3.3 V.

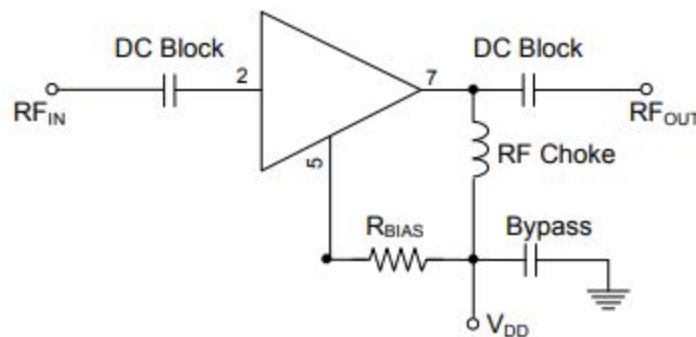


Figure 2.3: Circuit diagram for MAAL-011078 LNA to be used in a 50 Ω system [11]

2.1.3 Logarithmic Amplifier

The AD8138 was chosen as the logarithmic amplifier because it had a very low propagation delay (less than 40 ns) when compared to similar products. Its dynamic range was larger than 70 dB, which made it capable of detecting low powered signals [12]. It provided envelope detection and amplification crucial to our design.

2.1.4 Voltage Comparator

The logarithmic amplifier gave an output which was inversely related to the input power. In order to correct this, we passed the output of the logarithmic amplifier through a comparator set up in the inverting configuration. The TLV3201 voltage comparator was chosen because it had a very low rise time (less than 10 ns) and because it was very inexpensive (\$0.60) [13].

The TLV3201 is shown in Fig 2.4 below in the inverting configuration with hysteresis. Hysteresis is the idea that the output of the comparator changes the behavior of the circuit. This is beneficial for our design. Without hysteresis, the comparator might change states due to small changes in the input, which is undesirable.

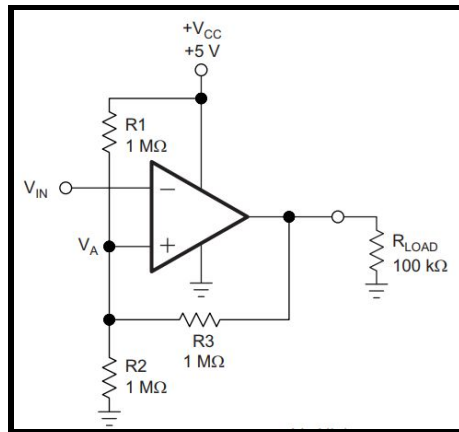


Figure 2.4: TLV3201 in Inverting Configuration with Hysteresis [13]

The design equations to find R1, R2, and R3 with parameters of two threshold voltages V_{A1} and V_{A2} were given in the datasheet for the TLV3201, and are shown below. V_{A1} and V_{A2} were chosen after verifying the performance of the logarithmic amplifier. V_{A1} corresponds to the threshold voltage required for the output of the comparator to switch from a 'low' to 'high' state, while converse is true for V_{A2} .

$$V_{A1} = V_{CC} \times \frac{R2}{(R1 || R3) + R2} \quad (1.2)$$

$$V_{A2} = V_{CC} \times \frac{(R2 || R3)}{(R2 || R3) + R1} \quad (1.3)$$

2.2 Control Unit

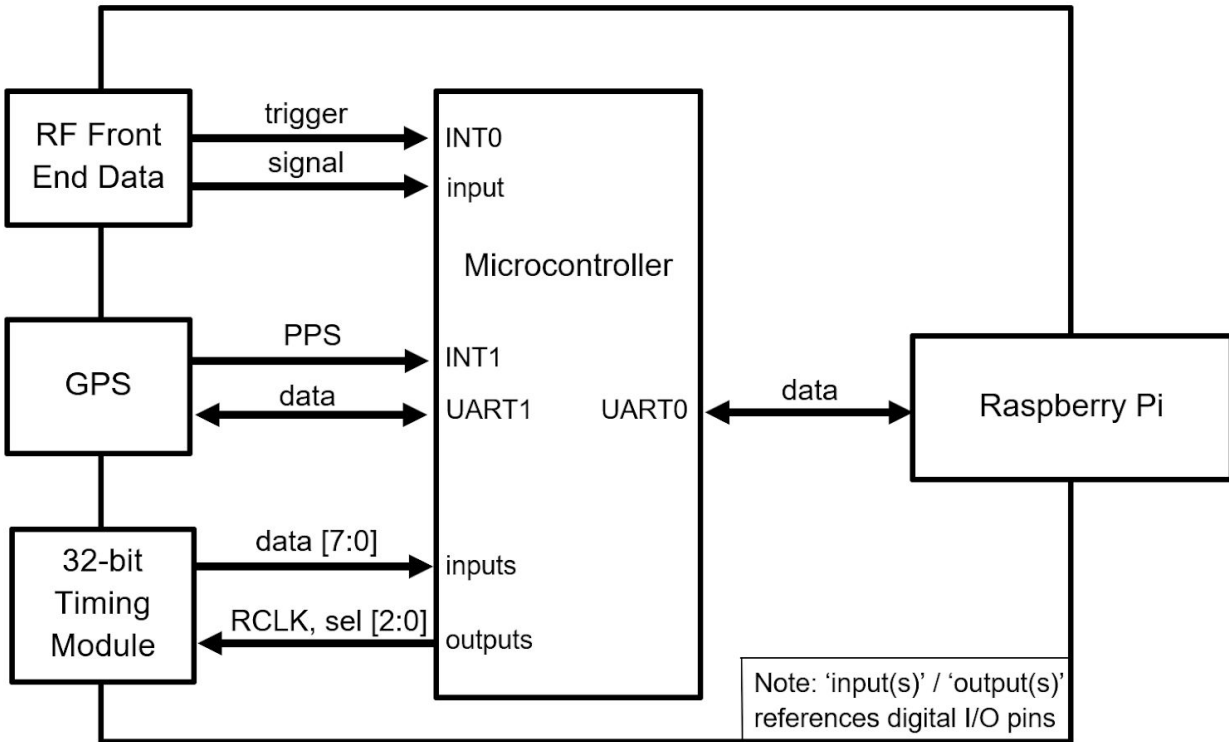


Figure 2.5: Block Diagram of the Control Unit

The primary objective for the Control Unit involves providing the exact time a transponder signal is discovered by the RF front end. This timestamp of the signal's arrival is required to be both precise and consistent for application in TDOA calculations. Each receiver in the network performs the same timestamping task; however, using the differences in time only becomes relevant if all receivers use a common time reference. Synchronized receivers also have the benefit that processing delay is abstracted away when using the difference of two timestamps.

At its core, the Control Unit processes data from external sensors, acts on the information, and makes the results available in a valuable manner for the end-user. Reviewing the general block diagram of the control unit presented in Fig. 2.5, the digital hardware block consists of three data-producers: the output digital feed from the RF front end, the GPS board, and a timing module. In addition to these producers, a microcontroller handles communication and performs data analysis. A Raspberry Pi functions as the pseudo-server for the purposes of demonstration and focusing the project on achieving one functional receiver.

The following discusses the design used for final demonstrations. Timestamp generation occurs in two steps: the first involves an external counter working at a target frequency to continuously count the number of clock cycles before a RF signal is received. The second requires resetting the counter value upon the synchronization trigger, which is provided by a Pulse-Per Second (PPS) signal. A GPS module provides the PPS from the GPS interface protocol with the satellite update ping [14]. The number of clock cycles when the RF signal trips an interrupt divided by the number of cycles in a single second (between PPS interrupts) provides the fraction-of-a-second time accuracy.

Apart from the timestamp, the TDOA computation requires the known location of the receivers to provide physical reference to estimate the transponder location. GPS performs a second purpose of delivering the positional data over an UART connection to the microcontroller where the raw data is decoded, and the required latitude and longitude data may be extracted [15] and relayed to the server.

Alternatives for the control unit include internally processing the counter data on the microcontroller and connecting directly to an ethernet control module that provides direct access to the internet. A benefit of this design is a removal of external hardware to support the counter and the increased integration of the design, saving on power usage and minimizing form-factor.

A concern with an internal design is the limitations set by the operating frequency of the timer block imbedded in the chip. Using clock cycles to represent timestamps quantizes the capacity of the system to resolve the distance light travels to the outcome of Eq. (1.4). Further target frequency analysis is discussed in Section 2.2.2 Timing Module.

$$\text{Distance Light Travels per Cycle} = F^{-1} \times c \quad (1.4)$$

Where c is the speed of light, and F is the operating frequency of the timer block.

2.2.1 GPS Module

The GPS module consists of a u-blox NEO-M8N receiver and a GAACZ-A active antenna. Together this system can receive a GPS signal with a sensitivity of -170 dBm. This is a result of adding the gain of the active antenna, which is 3 dB [16], to the receiver's sensitivity of -167 dBm [17]. The NEO-M8N was specifically chosen since it can supply a bias voltage to the active antenna while still providing both a time-synchronized PPS signal and positional data for the receiver. Positional data can be fetched through a UART serial interface as latitude and longitude.

2.2.2 Timing Module

The external timing module provides access to a dedicated 32-bit counter with control logic to save the current value to registers upon request, reset the internal counter logic, and read out the contents in 8-bit segments. The SN74LV8154 Dual 16-Bit Binary Counter is configured to interface both counters together by connecting pins RCOA and CLKBEN, thus functioning as the required 32-bit counter[18]. Its 8 pins of output are connected to the microcontroller's digital input pins. The RCLK pin used to save the current clock value to the registers [19] is connected directly to an output pin on the microcontroller

Active-low control logic for the counter is handled by a SN74LVC138A 3-line to 8-line decoder that provides a logic-low signal for only the selected address and logic-high for all other output lines [20]. As documented in Fig. B.4, the decoder's first 5 output lines are connected to the input control pins of the counter. The decoder reduces the required control bits to interface between the microcontroller and counter ICs from 5 to 3 bits, thereby reducing control overhead when using the counter.

The module's crystal oscillator runs at 40 MHz to provide the input signal for the counter chip. Applying Eq. (1.4) with the 40 MHz operating frequency, light travels about 7.49 m per clock cycle. 40 MHz was selected as the operating frequency because both counter and decoder datasheets function at this frequency range and the availability of oscillators.

2.2.3 Microcontroller

The microcontroller selection changed from a PIC32MX795F512L to an ATMEGA328 during the progression of the project: further information behind this substitute is found in Section 5.2.

An ATMEGA328 was selected to perform the communication and control requirements of the design. This microcontroller has the minimal amount of I/O capacity to connect all three data-producing devices and the Raspberry Pi. The counter IC outputs are connected to the microcontroller's digital pins 0-7 which is also called Port D. Pins 8 and 9 are configured as a software supported UART communicating with the Raspberry Pi, and pins 10 and 11 is another software supported UART port interfacing with the GPS module. Both PPS and RF comparator signals are mapped to digital pins 13 and 12 respectively. Rounding out the remainder of connections is Port C (analog pins 0-3) are connected to the input pins of the timing module.

The selection of this IC in particular is the same chip used in the Arduino Uno series, and provides enough I/O pins to implement the base-level design used for demonstration. Port manipulation allows access to multiple I/O data pins at once, provided these pins are connected to the same port. Digital pins 0 to 7 support reading a Byte of data at a time and interface directly with the timing module's output [20]. Software supported UART communication channels allow data to flow between the GPS and microcontroller, and the microcontroller and Raspberry Pi despite the hardware UART (digital pins 0 and 1) already being consumed by the timing module outputs. This software UART approach is limited by microcontroller limitations and only may receive from one port at a time [21]. In order to accomodate this issue, the Pi is only sent data and not expected to transmit.

In order for the microcontroller to utilize interrupt functionality when a PPS and RF comparator signal is identified, a special Pin Change Interrupt library is used to identify when a pin changes from a logic-low to logic-high and triggers the appropriate interrupt [22].

2.2.4 Raspberry Pi

A Raspberry Pi functions to display information that is created on the microcontroller and sent over a UART connection to GPIO pins 8 and 10 [23]. The purpose of this device is to aid demonstration purposes in order to visualize the timestamps collected at each PPS and RF interrupt. This is considered our pseudo-server as the data acquired on this device would be sent through an ethernet controller to a central computer and process the TDOA provided the Host and Server were accomplished in the scope of the project.

2.2.5 Software

The software for the project was developed using the Arduino IDE and targets the ATMEGA328 directly. The microcontroller needs the capability to synchronize and generate timestamps, as well as decode GPS data strings into position information. Due to a library compatibility issue between the Pin Change Interrupt [22] and NMEAGPS [15] libraries, separate codes demonstrating the functionality of interpreting the GPS data and accessing the timestamp data from externally triggered signals were required.

Appendix D.1 covers the software used to generate and transfer timestamps from both PPS and RF external interrupts to a Raspberry Pi for display and confirmation of functionality. The interrupt handlers for PPS and RF signals vary by a small difference: PPS needs to record the current number of cycles that occurred since the last PPS trigger and then reset the counter value, RF only records the current cycle count to achieve a timestamp.

The GPS code provided in Appendix D.2 functions to read data from the software supported UART port off of the GPS module and uses the provided Neo libraries to decode the message. The message contains, in part, Coordinated Universal Time (UTC) time, latitude, and longitude data [15]. Information is then delivered to the Raspberry Pi via a separate software UART port.

The Raspberry Pi's functionality is to read from the UART receiver port using the code under Appendix D.3 and filters the context to only display strings containing content for the user's visibility [23]. The device functions as a placeholder for communicating the information to a server over ethernet. By demonstrating the successful data movement from off the receiver unit and into the Raspberry Pi, the functionality of accessing the created data is provided.

2.3 Power Module

A 48V PoE wall adapter was used to power our receiver. We then stepped the voltage down to 12 V using a PoE splitter, and stepped the voltage down even further to 5 V and 3.3 V using the MC7805 and LD1117A voltage regulators. These regulators were capable of supplying 1 A of current, which was more than sufficient for our design. Originally, our plan was to use a stripped ethernet cable combined with a wall supply in order to create a PoE cable. However, we felt this implementation would be unsafe, so we instead decided to use a PoE splitter.

3 Design Verification

Appendix A contains the requirements and verification tables for each major component used in our design. The results obtained through the verification procedures are shown and described below.

3.1 RF Front End Verification

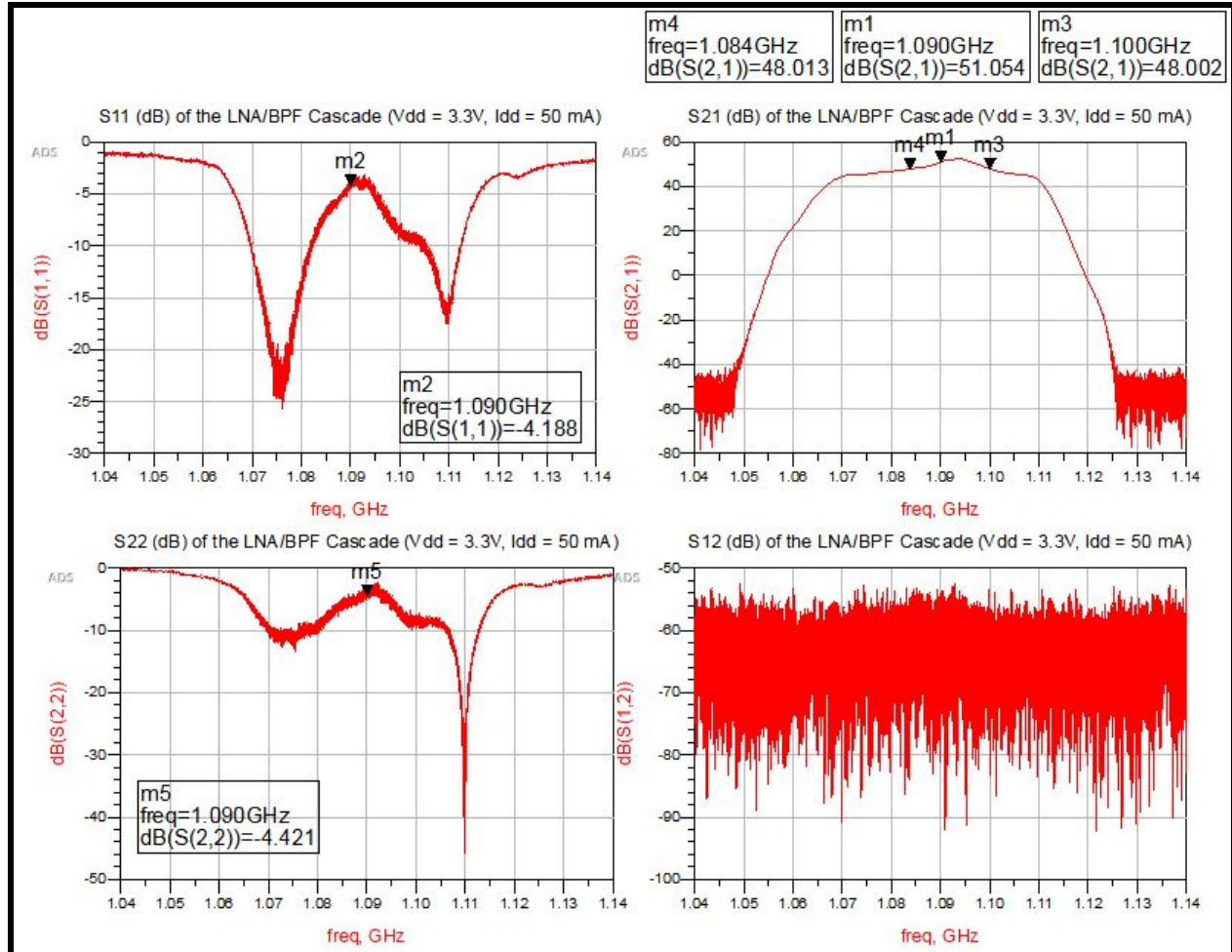


Figure 3.1: S-Parameter Measurements of the BPF/LNA Cascade

Shown in Fig 3.1 are the measured S-Parameters of the BPF/LNA cascade. The measured S21 of the cascade was about 51 dB at 1090 MHz, and the 3 dB bandwidth was about 20 MHz, which met the requirements for both the LNA and the BPF (see full requirements in Tables A.1 and A.2 in Appendix A below).

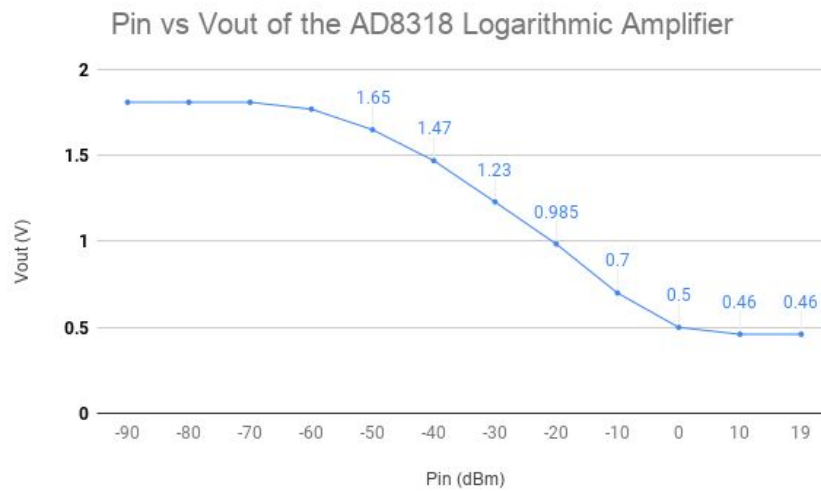


Figure 3.2: Measured Output Voltage of the AD8318 for Different Input Powers

Figure 3.2 contains the output voltage of the logarithmic amplifier in response to different input powers. The input consisted of a pulsed RF signal with a carrier frequency of 1090 MHz. We can see that there is an inverse relationship between the input power and the output voltage. The rise time of logarithmic amplifier was measured to be 20 ns, and as shown in Fig 3.2, we can see for an input power level of -20 dBm, the output voltage is greater than 0.7 V, which meets the requirements specified in Table A.3 in Appendix A below.

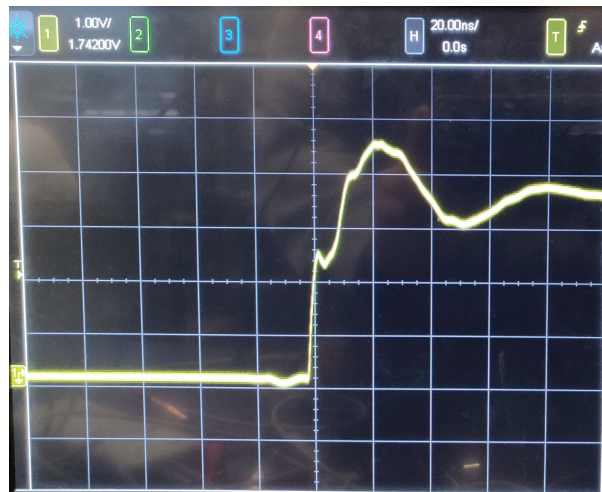


Figure 3.3: TLV3201 Comparator Response to a Square Wave Input

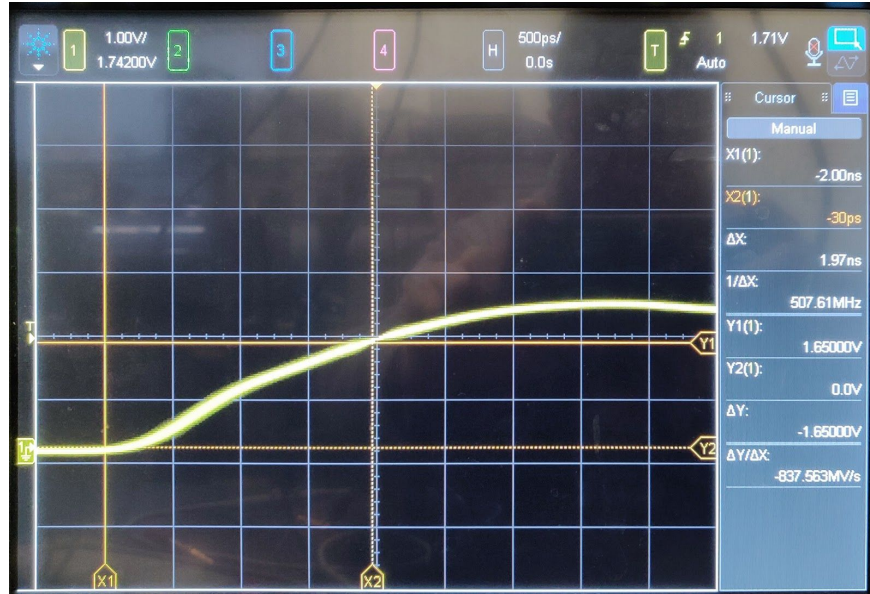


Figure 3.4: TLV3201 Comparator Rise Time Measurement

Figures 3.3 and 3.4 contain the measured response of the comparator to a square wave input. Figure 3.3 shows a wide view of the rising edge at the comparator output. We can see that the undershoot value is 3 V, which means that the comparator does not fall below 3 V when it switches from a 'low' to 'high' state. In Fig 3.4, we can see a narrow view of the rising edge of the comparator. The measured rise time is 1.97 ns, which meets the requirements for the comparator specified in Table A.4 in Appendix A below. Based on Fig 3.2, V_{A1} was chosen to be 1.2 V and V_{A2} was chosen to be 1.1 V, resulting in $R1 = 36 \text{ k}\Omega$, $R2 = 18 \text{ k}\Omega$, and $R3 = 390 \text{ k}\Omega$

3.2 GPS Verification

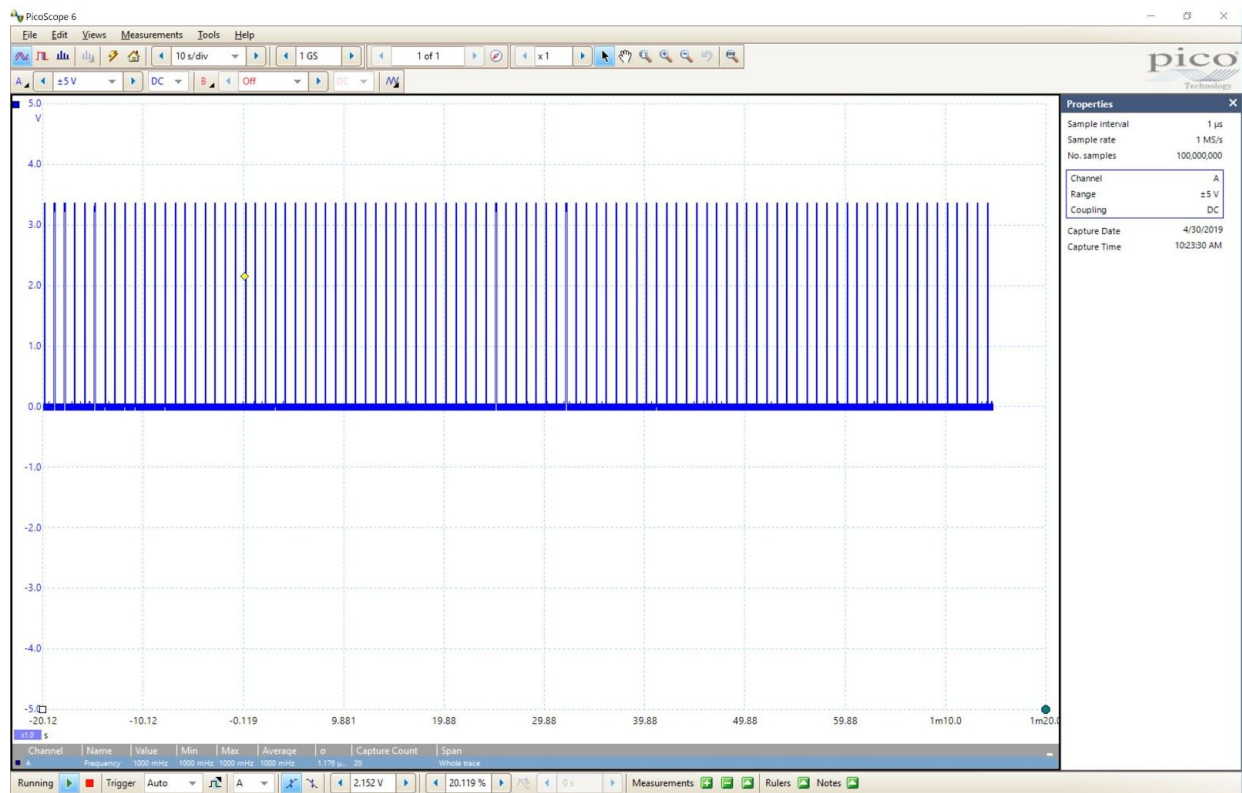


Figure 3.5: PPS Trigger Waveform

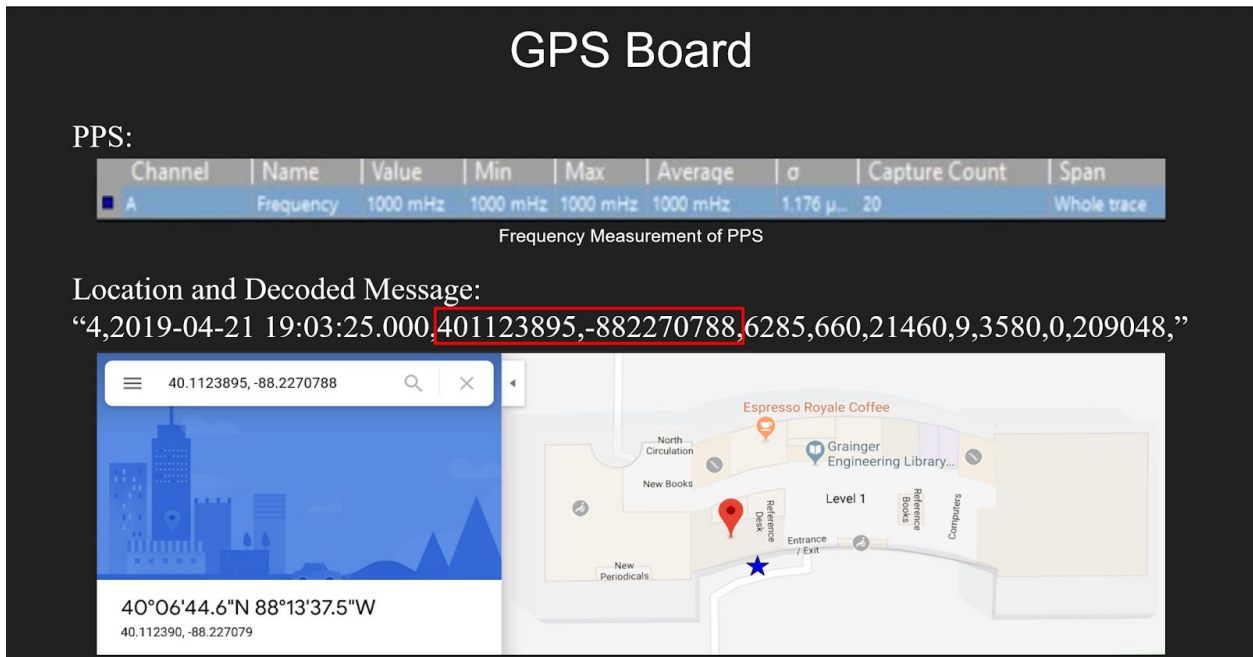
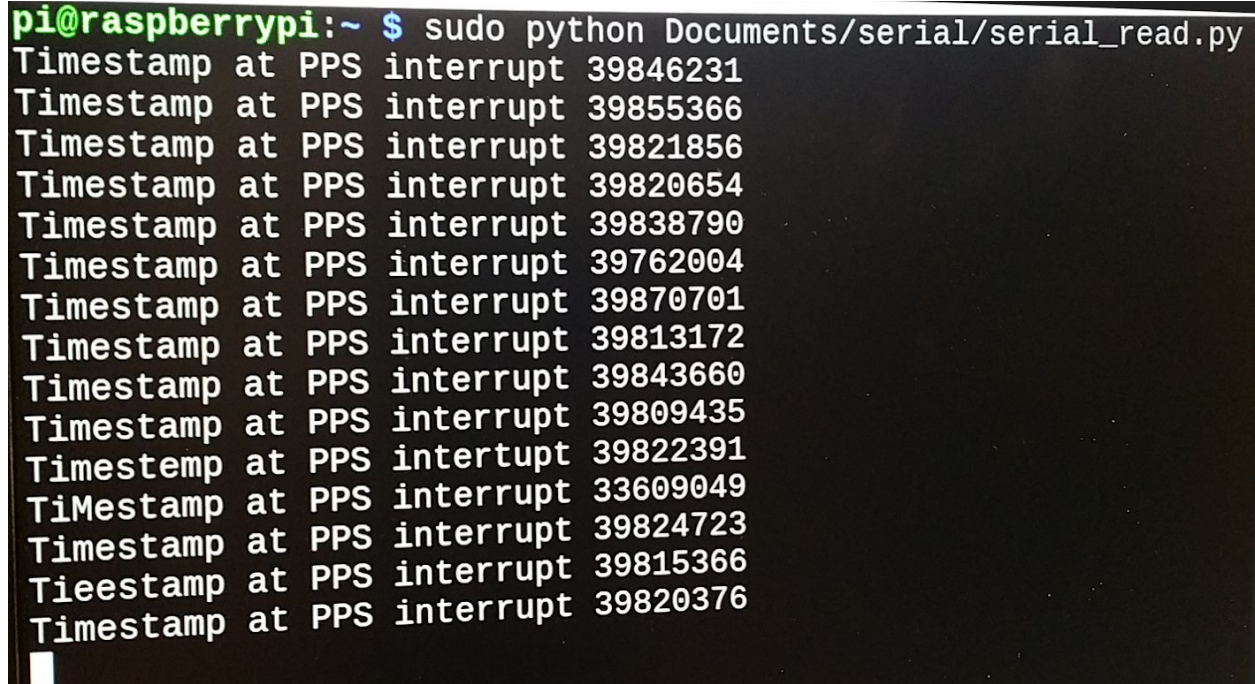


Figure 3.6: Example of GPS Output and Google Map Position [23]

The GPS verification covers the modules requirements from Table A.6 to create a PPS (Figure 3.5) and have the locational output be interpreted (Figure 3.6) where the error was about 20 feet.

3.3 Control Unit Verification



```
pi@raspberrypi:~ $ sudo python Documents/serial/serial_read.py
Timestamp at PPS interrupt 39846231
Timestamp at PPS interrupt 39855366
Timestamp at PPS interrupt 39821856
Timestamp at PPS interrupt 39820654
Timestamp at PPS interrupt 39838790
Timestamp at PPS interrupt 39762004
Timestamp at PPS interrupt 39870701
Timestamp at PPS interrupt 39813172
Timestamp at PPS interrupt 39843660
Timestamp at PPS interrupt 39809435
Timestamp at PPS interrupt 39822391
Timestamp at PPS interrupt 33609049
Timestamp at PPS interrupt 39824723
Timestamp at PPS interrupt 39815366
Timestamp at PPS interrupt 39820376
```

Figure 3.7: Example of PPS functionality

In Fig. 3.6, the PPS interrupt is being triggered by the GPS module's output and the control unit is generating a timestamp for the encounter and sending that data to the Raspberry Pi to be displayed. The fourth to last message actually occurred when the PPS signal was temporarily dropped and caused an unexpected output which explains the vastly different timestamp. The average clock cycle at the time of PPS reset was 39,825,980, or approximately represents 995.65 ms, with a standard deviation of 25,587.72 clock cycles. The output as displayed on the Pi covers the functionality of the timing module's third RV element (Table A.7).

The first and second requirements are both covered by each component supporting a 40 MHz operation frequency in their respective data sheets [18, 19] and the counter having the capacity to hold 32 bits of data.

The microcontroller's requirements, as listed in Table A.8, could not be fully validated due to the change from PIC32 to ATMEGA328 ICs. From this change, the operating frequency peaked at 16 MHz, only two software-based interrupts could be implemented, and the Ethernet and User input functionalities were abandoned due to time and hardware constraints. Figure 3.7 demonstrates that although the capability of the microcontroller was limited, the project was still capable of supporting the generation of timestamps and interface with the timing unit.

4 Costs

Table 4.1: Cost of Parts for a Single Receiver

Part number	Description	Quantity	Unit Price	Subtotal	Module
MAAL-011078	LNA	2	\$5.88	\$11.76	RF
TA1090EC	BPF	3	\$1.50	\$4.50	RF
AD8318	RF Detector	1	\$10.60	\$10.60	RF
M7805	5V Regulator	2	\$0.57	\$1.14	Power
LD1117A	3.3V Regulator	1	\$0.55	\$0.55	Power
TLV3201DCK	Comparator	1	\$0.60	\$0.60	Digital
NEO-M8N	GPS Module	1	\$15.05	\$15.05	GPS
GAACZ-A	GPS Antenna	1	\$5.00	\$5.00	GPS
SN74LVC138AQPWRQ1	3-to-8 Bit Decoder	1	\$0.61	\$0.61	Digital
SN74LV8154N	Dual 16 Bit Counter	1	\$1.09	\$1.09	Digital
ATMEGA328-PU	Arduino Uno MCU	1	\$1.95	\$1.95	Digital
TOYOCOM 121-049	40 MHz xtal	1	\$1.12	\$1.12	Digital
A160L8F	16 MHz xtal	1	\$0.75	\$0.75	Digital
OshPark pcb#1	PCB	1	\$10.00	\$10.00	PCB
	RLC elements and wires	1	\$8.00	\$8.00	Misc.
Total Cost for a Single Receiver:				\$72.72	

Table 4.2: Labor Cost

Name	Hourly Rate	Hours Invested	Subtotal = Hourly Rate x Hours Invested x 2.5
Rushik Desai	\$36.22	180	\$16,299
Benjamin Du	\$36.22	180	\$16,299
Kyle Rogers	\$36.22	180	\$16,299
Total	\$36.22	540	\$48,897

Table 4.3: Total Cost

Labor	\$48,897
Parts Cost (Single Receiver Cost x 4)	\$290.88
Total Cost	\$49,187.88

5 Conclusion

5.1 Accomplishments

Our project succeeded in two of the three high level requirements: each receiver costs less than \$100 and consumes less than 15.4 W using PoE. The project has a fully integrated RF front end that can receive a pulsed 1090 MHz signal and demodulate down to a digital signal. Additionally, we have a fully integrated control unit that can produce a time-synchronized timestamp for a received digital signal via an interrupt. From there the timestamp is sent to a Raspberry Pi to be displayed, which acts as a pseudo-server.

5.2 Problems

Even though both the RF front end and control unit were fully integrated individually, we failed to integrate them together due to time constraints. Theoretically our two systems could've been fully integrated. Unfortunately, there were setbacks caused by an accidental short-circuit that destroyed our control unit before an integration test could be made.

The project failed to meet the third high level requirement of detecting an aircraft. However, if we were able to detect an aircraft signal, a MATLAB script was prepared to calculate its position as shown in Appendix D. Issues with detecting an aircraft signal extended to tests that were attempted using existing 1090 MHz receivers provided by Professor Levchenko. Tests included using a receiver with the roof antenna in ECEB 5020, as well as a field test near the Willard CMI Airport.

5.3 Uncertainties

There were a few changes that had to be made to the original design. First of all, we changed our focus from having four receivers to having one functioning receiver. Unfortunately that meant that detecting an aircraft's position wouldn't be possible. However, creating one receiver meant that four could theoretically be implemented, which would have to suffice. As a consequence, implementation of the ethernet connection to a central server was replaced with a Raspberry Pi that acted as a pseudo-server for our single receiver.

Many RF front end issues were a result of our PCB layout. The first issue we encountered was with some ground pads coupling to adjacent DC bias pads. What was observed was floating voltages on adjacent ground pads, which we suspect lead to incorrect biasing of our LNA and logarithmic amplifier. This was fixed by removing certain ground pads adjacent to the DC bias pads. Coupling is the main suspect considering that the thickness of our board, which is 62 mil, is significantly larger than the spacing of our IC pads of 10 mil. This means that the pads couple more strongly to adjacent pads than the bottom ground plane. Another issues appeared with the impedance matching of our circuit. We expected a match with return loss of at least -10 dB for our cascaded system. However, the inclusion of board parasitics and discontinuities from large components, such as 1206 capacitors, made matching much more difficult.

One critical error that we made was forgetting to add thermal vias for our IC's. This, along with board parasitics, might explain the unusually high current biasing we saw with our amplifiers. Nominal current for our amplifiers was around 50 mA [11,12] while we were actually drawing around 90 mA at 5 V. The LNA's current draw was fixed by biasing them with 3.3 V instead of 5 V, and adjusting the bias resistor to

compensate. They now draw around 50 mA, an acceptable value, and maintained the same desired gain. We weren't able to fix the logarithmic amplifier this way since it only operates at 5 V [12]. Unfortunately, biasing at a lower voltage for the LNA's decreases the P1dB, which decreases our dynamic range by 10 dB [11]. This was a sacrifice that had to be taken for a stabler front end.

See Appendix F for further details on omitted design implementations.

5.4 Ethical Considerations

The project incurred several possible ethical and safety concerns that required attention during the design, implementation, and deployment phases. These ethical concerns will reference both IEEE and Association of Computing Machinery (ACM) Code of Ethics.

From the Association of Computing Machinery's Code of Ethics, "An essential aim of computing professionals is to minimize negative consequences of computing, including threats to health, safety, personal security, and privacy. [25]". This is similar to the #1 IEEE Code of Ethics [26]. A particular concern arises as the data acquired by the proposed receiver solutions includes GPS positional data and this information is needed on the server, sent over the ethernet. Had we implemented communication with our central server through ethernet, we would have encrypted the positional data.

Our receiver was passive and did not interfere with any FCC regulations. It also met the power requirements set by the IEEE 802.3af standard.

5.5 Future Work

In the future, we would first fully integrate our design onto a single PCB. This would allow each receiver to have a smaller form factor. We would also enclose our receivers so they they would be resistant to harsh weather conditions. Next, we would build all four receivers to and create a network in order to connect them to a central server.

The first design change we would make would be to make the RF front end dynamic, so that its gain could respond to different input power levels. With a dynamic front end, the risk of our components malfunctioning or being overdriven would decrease significantly.

On the digital end, more could be done to improve receiver's capability to determine the signals that it is receiving. While the modulation scheme is the same, there are a few different message schemas that aircraft can use to communicate [7].

Finally, we would add user-friendly support to our receivers, including push buttons which could perform system tests or reset the receivers, status LEDs, and web-applications which could extract data from each receiver.

References

- [1] M. Thurber, "ADS-B Is Insecure and Easily Spoofed, Say Hackers," Aviation International News, 03-Sep-2012. [Online]. Available: <https://www.ainonline.com/aviation-news/aviation-international-news/2012-09-03/ads-b-insecure-and-easily-spoofed-say-hackers>. [Accessed: 08-Feb-2019].
- [2] "How It Works," Flight Radar 24. [Online]. Available: <https://www.flightradar24.com/how-it-works>. [Accessed: 07-Feb-2019].
- [3] "PASSIVE RADAR ACTIVISTS," Thales Group, 19-Nov-2014. [Online]. Available: <https://www.thalesgroup.com/en/worldwide/aerospace/case-study/passive-radar-activists>. [Accessed: 07-Feb-2019].
- [4] J. R. Wilson, "New frontiers in passive radar and sonar," Military & Aerospace Electronics - Military technology, weapons, equipment & systems for the military industrial complex., 08-Feb-2016. [Online]. Available: <https://www.militaryaerospace.com/articles/print/volume-27/issue-2/special-report/new-frontiers-in-passive-radar-and-sonar.html>. [Accessed: 07-Feb-2019].
- [5] J. Vierinen, "Building your own SDR-based Passive Radar on a Shoestring," Hackaday, 01-Jul-2015. [Online]. Available: <https://hackaday.com/2015/06/05/building-your-own-sdr-based-passive-radar-on-a-shoestring/>. [Accessed: 07-Feb-2019].
- [6] "What is Power Over Ethernet (PoE) and what is it used for?" versatek.com [Online]. Available: <https://www.versatek.com/what-is-power-over-ethernet/> [Accessed: 19-Feb-2019]
- [7] C. Wolff, "SSR - The Reply Message", Radartutorial.eu. [Online]. Available: <http://www.radartutorial.eu/13.ssr/sr07.en.html>. [Accessed: 28-April-2019]
- [8] Mott, "Estimation of aircraft distances using transponder signal strength information", *Taylor & Francis*, 2018. [Online]. Available: <https://www.tandfonline.com/doi/full/10.1080/23311916.2018.1466619> [Accessed: 20-Feb-2019].
- [9] Sensor Systems Inc, "ADS-B/UAT S65-5366-895L", S65-5366-895L Datasheet [Accessed 21-Feb-2019].
- [10] TAI-SAW Technology, "Saw Filter 1090 MHz," TA1090EC Datasheet, [Accessed: 20-Feb-2019]
- [11] MACOM, "Low Noise Amplifier, 700 MHz - 6 GHz," MAAL-011078 Datasheet [Accessed: 19-Feb-2019]
- [12] Analog.com. *1 MHz to 8 GHz, 70 dB Logarithmic Detector/Controller*. [Online] Available at: <https://www.analog.com/media/en/technical-documentation/data-sheets/ad8318.pdf>

- [13] Texas Instruments, "TLV7031 and TLV7041 Small Size, nanoPower, Low-Voltage Comparators," TLV7031 Datasheet, [Accessed: 28-April-2019]
- [14] u-blox, "Versatile GNSS modules," NEO-M8 Series Datasheet [Accessed: 19-Mar-2019]
- [15] SlashDevin/NeoGPS. [online] GitHub. Available at: <https://github.com/SlashDevin/NeoGPS>[Accessed 13 Apr. 2019].
- [16] GENERIC BRAND, "GPS Active Antenna," GAACZ-A Datasheet [Accessed: 19-Mar-2019]
- [17] u-blox, "Versatile GNSS modules," NEO-M8 Series Datasheet [Accessed: 19-Mar-2019]
- [18] Texas Instruments, "Dual 16-Bit Binary Counters With 3-State Output Registers," SN74LV8154 Datasheet, Aug. 2004 [Revised Oct. 2015].
- [19] Texas Instruments, "3-LINE TO 8-LINE DECODER/DEMULTIPLEXER," SN74LVC138A-Q1 Datasheet, Sept. 2003 [Revised Feb. 2008].
- [20] Arduino - PortManipulation. [online] Available at: <https://www.arduino.cc/en/Reference/PortManipulation>.
- [21] Arduino - SoftwareSerial. [online] Available at: <https://www.arduino.cc/en/Reference/SoftwareSerial>.
- [22] NicoHood/PinChangeInterrupt. [online] GitHub. Available at: <https://github.com/NicoHood/PinChangeInterrupt>.
- [23] Read and Write From Serial Port With Raspberry Pi. [online] Instructables. Available at: <https://www.instructables.com/id/Read-and-write-from-serial-port-with-Raspberry-Pi/>.
- [24] Google Maps. (2019). 40°06'44.6"N 88°13'37.5"W. [online] Available at: <https://www.google.com/maps/place/40%C2%B006'44.6%22N+88%C2%B013'37.5%22W/@40.1120095,-88.2268208,19z/data=!4m5!3m4!1s0x0:0x0!8m2!3d40.1123895!4d-88.2270788> [Accessed 30 Apr. 2019].
- [25] "ACM Code of Ethics and Professional Conduct," *Association for Computing Machinery*, 2018. [Online]. Available: <https://www.acm.org/code-of-ethics>. [Accessed: 08-Feb-2019].
- [26] "IEEE Code of Ethics," *IEEE - Advancing Technology for Humanity*. [Online]. Available: <https://www.ieee.org/about/corporate/governance/p7-8.html>. [Accessed: 08-Feb-2019].
- [27] K. Nishikawa and K. Sato, "LOW PROFILE ANTENNA FOR LAND MOBILE COMMUNICATIONS", 5146232, 1992.

Appendix A - Requirements and Verification

Table A.1: R&V for the LNA (MAAL-011078)

Requirements	Verification
<p>1. The LNA must have a supply voltage of 3.3 V +/- 100 mV, and must draw a load current of 50 +/- 5 mA</p> <p>2. The LNA must have a gain (S21) of 27 +/- 2 dB at 1.09 GHz</p>	<p>1. Set bias resistance to 600 Ω so that the LNA will draw 50 +/- 5 mA [MAACOM]</p> <p>2. Use a network analyzer to measure the S-Parameters (particularly S21 and S11)</p> <ol style="list-style-type: none"> Define the parameters of the network analyzer. Set the frequency range to 1040 to 1140 MHz and the number of points to 8001. Set the attenuation to 30 dB (very important!) Calibrate the network analyzer using a TRL or SOLT calibration kit. Connect ports 1 and 2 of the analyzer to the input and output of the LNA, respectively, using pigtailed. Measure the gain of the LNA (S21) using markers.

Table A.2: R&V for the BPF (TA1090EC)

Requirements	Verification
<p>1. The BPF must have an insertion loss (S21) less than 2.5 dB</p> <p>2. The BPF must have a 0.5 dB bandwidth of less than 50 MHz</p>	<p>1 & 2. Use a network analyzer to measure the S-Parameters (particularly S21 and S11) of the BPF.</p> <ol style="list-style-type: none"> First, define the parameters of the network analyzer. Set the frequency range to 1040 to 1140 MHz and the number of points to 8001. Calibrate the network analyzer using a TRL or SOLT calibration kit. Connect ports 1 and 2 of the analyzer to the input and output of the BPF, respectively, using pigtailed. Use markers in order to measure the insertion loss (S21) of the filter at various points around 1090 MHz. These markers will also be used to verify the 0.5 dB bandwidth.

Table A.3: R&V for the Logarithmic Amplifier (AD8318)

Requirements	Verification
1. Given a 1090 MHz carrier (with a power greater than -20 dBm) modulated with rectangular pulses with a width of 0.45 μ s and spacing of 1 μ s, the AD8318 logarithmic amplifier must be able to demodulate the carrier, and output the pulses with a rise time no longer than 100 ns and amplitude greater than 700 mV.	1. The modulated carrier can be generated using an SDR, or by mixing a pulsed signal created by a waveform generator with a 1090 MHz carrier created using a signal generator. Using an oscilloscope, the response at the output of the logarithmic amplifier can be measured, and the rise time and output voltage can be measured.

Table A.4: R&V for the Voltage Comparator (TLV3201AQDCKRQ1)

Requirements	Verification
1. When given a square pulse with an amplitude greater than 700 mV, the comparator must be able to propagate the pulse with a propagation time no longer than 100 ns. The resulting pulse's maximum amplitude should be V_{cc} (which in our case is 3.3 +/- 25 mV).	1. <ul style="list-style-type: none"> A. Use a signal generator/pulse generator in order to generate a square wave with a period of 2 μs, and an amplitude of 1.5V B. Use a pigtail in order to connect the input of the comparator to the signal generator, and connect the output of the comparator to an oscilloscope. C. Connect the signal generator to a separate channel on the oscilloscope D. Use the oscilloscope in order to verify that the square wave has a propagation time through the comparator of less than 5 μs and has an amplitude of 3.3 +/- 25 mV

Table A.5: R&V for the 5 V and 3.3 V Voltage Regulators (MC7805 and LD1117A)

Requirements	Verification
1. Given an input voltage of 12V, the MC7805 must be able to supply a constant voltage of 5 +/- 50 mV, and the LD1117A must be able to supply a constant voltage of 3.3 +/- 50 mV for a period of 20 minutes while drawing 1A of current.	1. An electronic load can be used to draw 1A of current, and a digital multimeter can be used to measure the voltage across the terminals of the voltage regulators.

Table A.6: R&V for the GPS Module (NEO-M8N)

Requirements	Verification
<p>1. The GPS module must be able to generate a 1 Hz PPS signal.</p> <p>2. The GPS module must be able to output position data that can be interpreted by a microcontroller.</p>	<p>1. The module will be powered with a voltage supply and measured using a oscilloscope. The frequency and rise/fall time can be recorded over time to measure PPS time stability (preferably in a ppm format).</p> <p>2. For convenience, another microcontroller such as a Arduino can be used to receive the GPS data and unpack the data. Ideally, the microcontroller can support I2C or UART data transfer (which an Arduino can).</p>

Table A.7: R&V for the Timing Module

Requirements	Verification
1. Unit shall operate at a frequency of at least 40 MHz.	<p>1. Testing the frequency of the unit requires providing a known input frequency and monitoring the output for correctness of data and the switching delay on the output pins. This is also confirmed via the device's datasheet.</p> <ul style="list-style-type: none"> A. Place the counter unit on a breadboard with 0.1uF +/- 5 % capacitors between Vcc pin and voltage supply of 3.3V +/- 9%, drive pins 4, 5, & 6 to 3.3V +/- 9%, ground pins 3 & 10, connect pins 8 & 9 together. B. Configure the wave generator to output a 40 MHz square wave to pins 1, 2, & 7. C. Probe the LSB of counter A (pin 19) and input wave signal for reference using an oscilloscope with reference to the common ground. D. Evaluate characteristics of the output signal for stability and identify the frequency of the output. E. Repeat for each output pin on counter to ensure expected output waveforms for data bus without control glitches when overflowing.
2. Supports continues counting for at least 2 seconds (data representation of 27+ bits).	2. Data representation is device specific. Device's datasheet correctly demonstrates ability to hold the required amount of bits.
3. Control logic to access timestamps and supports GPS synchronization using the PPS signal.	<p>3.</p> <ul style="list-style-type: none"> A. Following pin connections in component's datasheet and figure 13, connect device to a microcontroller to provide the control signals and intake the counter data. B. Activate a clear by sending logic LOW to pin 11 and begin reading the counter's data with debug statements to show the incrementation of values. C. Address the four bytes of data to receive the full content from the counter and verify results via datasheet time diagram.

Table A.8: R&V for the Microcontroller

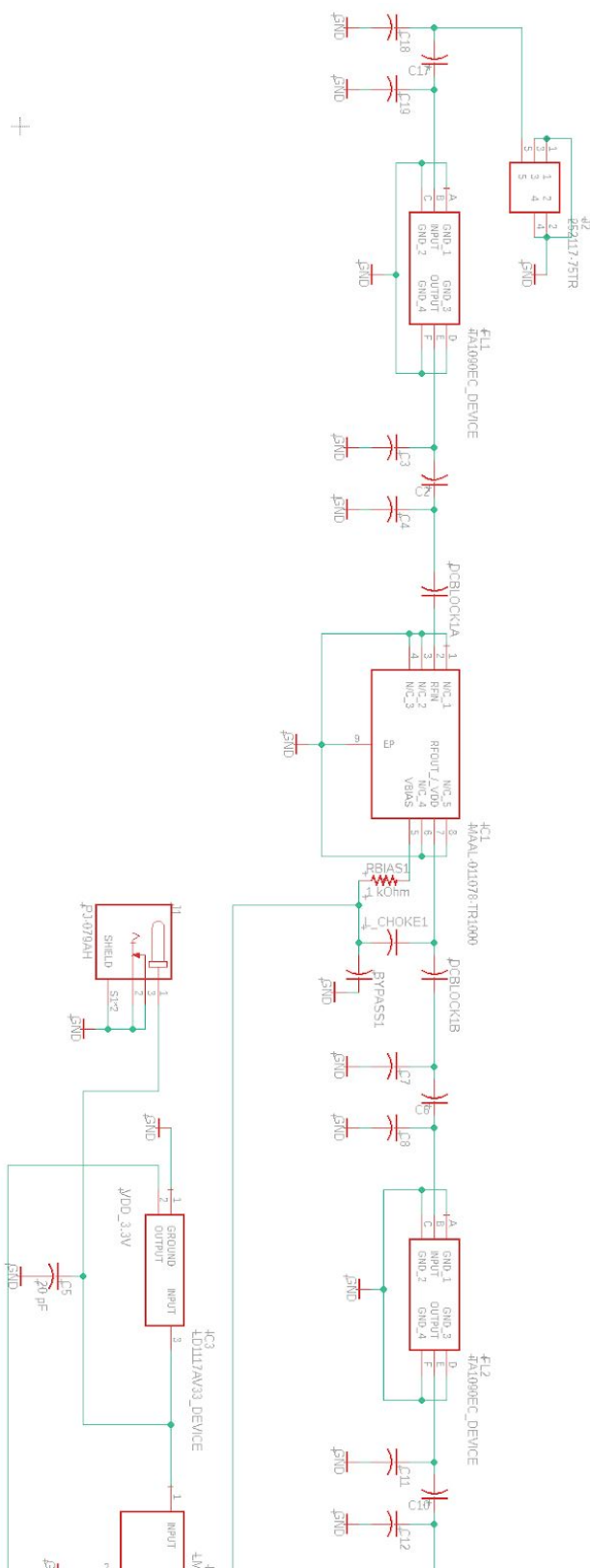
Requirements	Verification
<p>1. Sufficient operating frequency: at least 40 MHz.</p> <p>2. Controller requires at least 4 interrupt pins (two buttons, RF Front End output data, and PPS signal from GPS module).</p> <p>3. Capable of communication with ethernet module, timing unit, user inputs, and GPS.</p>	<p>1. Confirm operational conditions via datasheet, includes reviewing I/Vdd and Frequency curves.</p> <p>2. Pins are based on packaging and hardware support of the interrupt pins, consult datasheet specifications for sourced component.</p> <p>3. Communication testing requires test benching particular component interactions.</p> <ul style="list-style-type: none"> A. Using SPI, connect device to the ethernet controller. Run test program to verify upload and download capability. B. Connect microcontroller's digital I/O pins to the timing unit. Then program microcontroller with test bench code to clear, increment, and read data from timing unit. C. Verify push buttons trigger interrupts that are handled by the microcontroller by connecting the buttons to an input voltage and ground. The output goes to the interrupt pins. The interrupt code simply needs to be a debug statement or LED illumination to confirm connection. D. Finally, connect GPS module to microcontroller and configure input data. Use supported libraries to read from the module and decode the positional data. E. Combine benchmarks to send timestamp and GPS data to the server using the ethernet interface.

Power over Ethernet

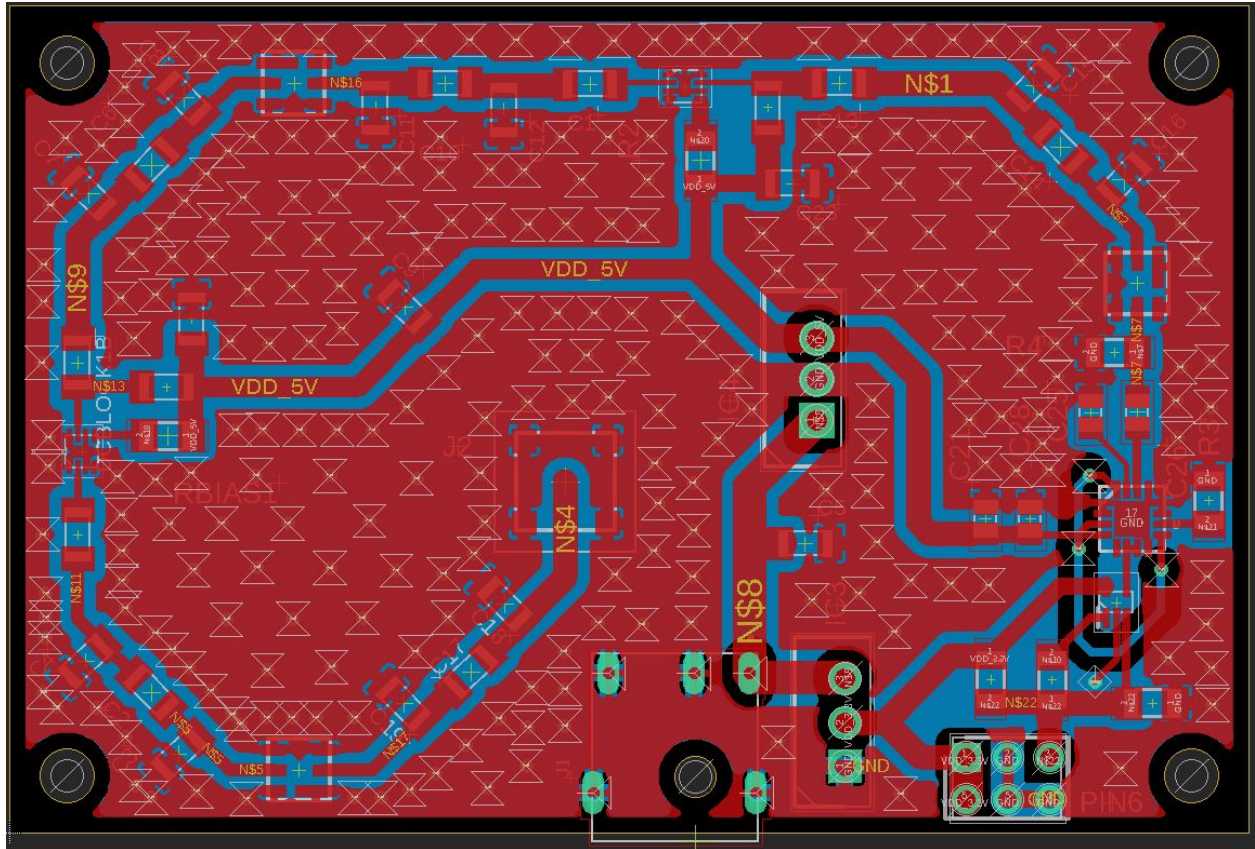
Table A.9: R&V for PoE

Requirements	Verification
<p>1. The POE cable must be able to provide up to 12V and up to 1A to our circuit (to remain below the IEEE 802.3af-2003 standard of 15.4W)</p> <p>2. The POE cable must allow communication between the host and ethernet microcontrollers.</p>	<p>1. An electronic load can be used to measure the voltage from the POE cable. The current draw can be set to 1A, and the voltage can be measured using a digital multimeter.</p> <p>2. The ethernet controller will ping the host using the ethernet port, and the host must recognize it.</p>

Appendix B - Project Schematics and Layout







Figures B.1, B.2, and B.3: Schematic and Layout for the RF Front End

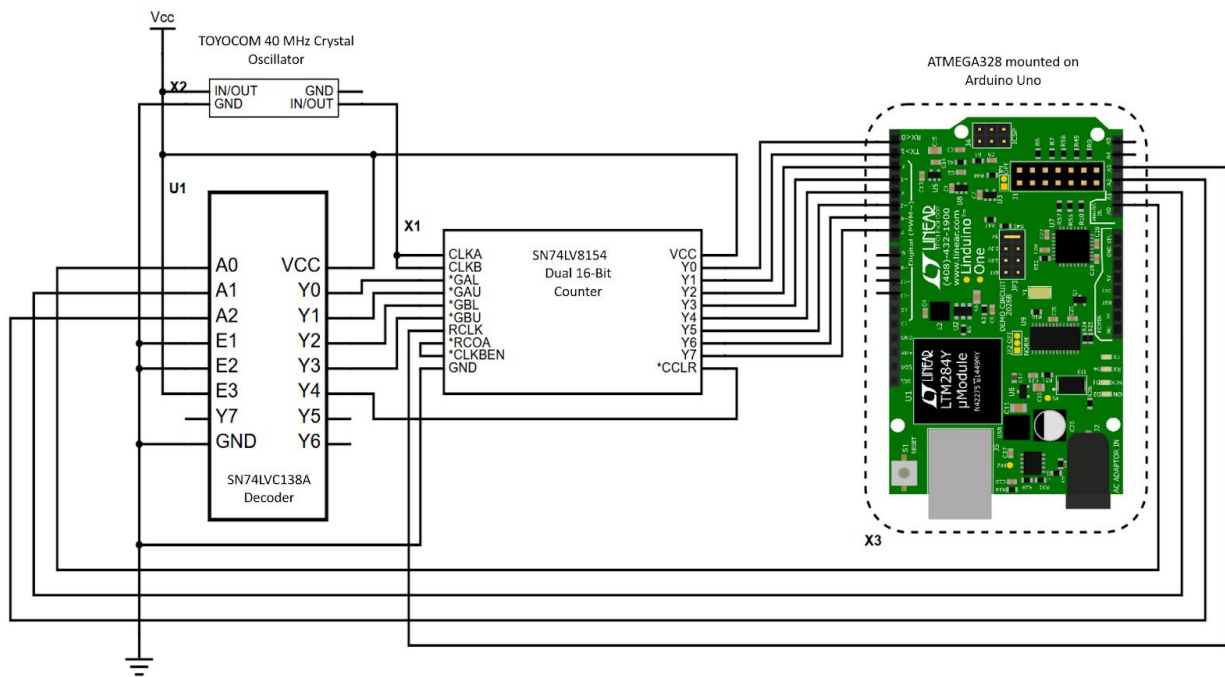


Figure B.4: Schematic of Timing Module

Appendix C - Software Flowchart

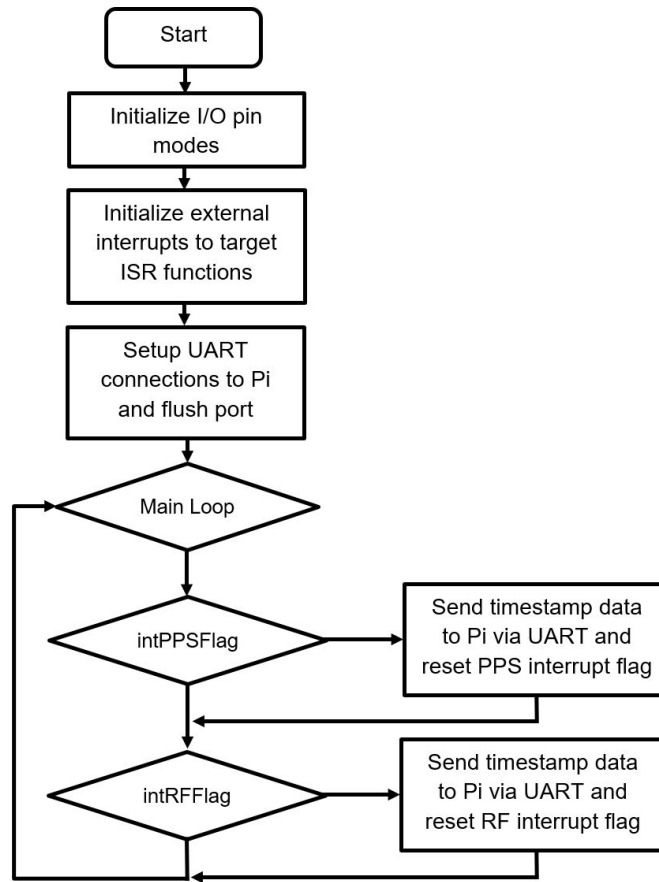


Figure C.1: Flowchart of Main Timestamp Function

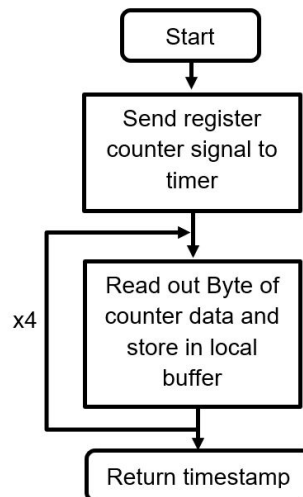


Figure C.2: Acquire Timestamp Flowchart

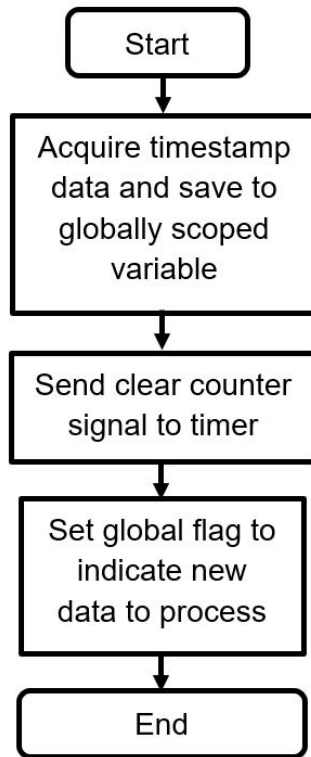


Figure C.3: PPS Interrupt Handler Flowchart

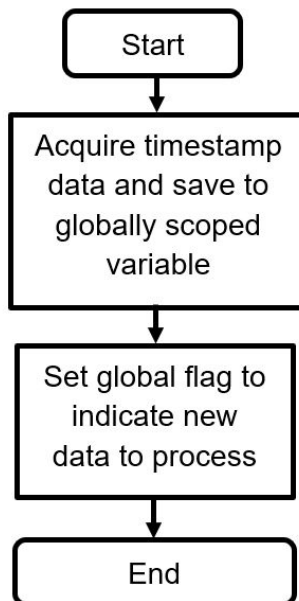


Figure C.4: RF Interrupt Handler Flowchart

Appendix D - Software Code

D.1 Timestamp Code

```
// allow interrupts to trigger from any pin: only detects changes
#include <PinChangeInterrupt.h>

// Software Serial Modules
#include <AltSoftSerial.h>
// #include <NeoSWSerial.h> // incompatible with PinChangeInterrupt library

#define PI_PORT_NAME "AltSoftSerial"
// #define GPS_PORT_NAME "NeoSWSerial(10,11)"

#define DELAY_MS 1 // to ensure signal stability

// macro definition to save the current counter value to registers
#define saveTimestamp() ({PORTC = B001000; delay(DELAY_MS);})

// macro definition to reset counter's value to 0
#define resetTimestamp() ({PORTC = B000100; delay(DELAY_MS);})

#define RF_COMPARE_PIN 12 // software supported external interrupt LOW-to-HIGH trigger for RF
input
#define PPS_PIN 13 // software supported external interrupt LOW-to-HIGH trigger for GPS's PPS

AltSoftSerial piPort; // software supported UART
// NeoSWSerial gpsPort(10, 11);

// globally scoped memory
unsigned long intTimeStamp=0;
bool intPPSFlag = false;
bool intRFFlag = false;

void setup() {
    Serial.end(); // disable UART from being used, so we can use as IO

    pinMode(RF_COMPARE_PIN, INPUT_PULLUP);
    pinMode(PPS_PIN, INPUT_PULLUP);

    attachPCINT(digitalPinToPCINT(RF_COMPARE_PIN), ISR_handle_RFC, RISING);
    attachPCINT(digitalPinToPCINT(PPS_PIN), ISR_handle_PPS, RISING);

    // using port manipulation for greater I/O efficiency
    DDRD = B00000000; // input
    DDRC = B111111; // output
}
```

```

    resetTimestamp();

    piPort.begin(9600);
    piPort.flush();
    piPort.print("Setup complete for Timestamp code\n");
    piPort.flush();

    Serial.end(); // disable UART from being used, so we can use as IO
}

// save timestamp to global memory and signal main loop that interrupt has occurred from RF
void ISR_handle_RFC() {
    unsigned long timestamp = getTimestamp();
    intTimeStamp = timestamp;
    intRFFlag = true;
}

// save timestamp to global memory, reset counter, and signal main loop that interrupt has occurred
// from PPS
void ISR_handle_PPS() {
    unsigned long timestamp = getTimestamp();
    resetTimestamp();
    intTimeStamp = timestamp;
    intPPSFlag = true;
}

// saves timestamp and reads out all four Bytes of data: requires four reads from counter chip
unsigned long getTimestamp() {
    saveTimestamp();
    unsigned long AL=0, AH=0, BL=0, BH=0;

    PORTC = B000000;
    delay(DELAY_MS);
    AL = PIND;
    PORTC = B000001;
    delay(DELAY_MS);
    AH = PIND;
    PORTC = B000010;
    delay(DELAY_MS);
    BL = PIND;
    PORTC = B000011;
    delay(DELAY_MS);
    BH = PIND;

    // recombine data Bytes into unsigned long format using bit shifts
    return (BH << 24) | (BL << 16) | (AH << 8) | (AL);
}

```

```

// main loop function, pools global variables until flag is set, then sends timestamp data to Pi via UART
for display
void loop() {
    unsigned long timestamp = getTimestamp();

    if (intPPSFlag) {
        piPort.print("Timestamp at PPS interrupt ");
        piPort.print(intTimeStamp);
        piPort.print("\n");
        intPPSFlag = false;
    }
    if (intRFFlag) {
        piPort.print("Timestamp at RF interrupt ");
        piPort.print(intTimeStamp);
        piPort.print("\n");
        intRFFlag = false;
    }
}
}

```

D.2 GPS Code

// Primary source of code from <https://github.com/SlashDevin/NeoGPS> [**NEOGPS**].

```
#include <NMEAGPS.h>
//=====
// Program: NMEA.ino
//
// Description: This program uses the fix-oriented methods available() and
//             read() to handle complete fix structures.
//
//             When the last character of the LAST_SENTENCE_IN_INTERVAL (see NMEAGPS_cfg.h)
//             is decoded, a completed fix structure becomes available and is returned
//             from read(). The new fix is saved the 'fix' structure, and can be used
//             anywhere, at any time.
//
//             If no messages are enabled in NMEAGPS_cfg.h, or
//             no 'gps_fix' members are enabled in GPSfix_cfg.h, no information will be
//             parsed, copied or printed.
//
// Prerequisites:
// 1) Your GPS device has been correctly powered.
//    Be careful when connecting 3.3V devices.
// 2) Your GPS device is correctly connected to an Arduino serial port.
//    See GPSport.h for the default connections.
// 3) You know the default baud rate of your GPS device.
//    If 9600 does not work, use NMEAdiagnostic.ino to
//    scan for the correct baud rate.
// 4) LAST_SENTENCE_IN_INTERVAL is defined to be the sentence that is
//    sent *last* in each update interval (usually once per second).
//    The default is NMEAGPS::NMEA_RMC (see NMEAGPS_cfg.h). Other
//    programs may need to use the sentence identified by NMEAorder.ino.
// 5) NMEAGPS_RECOGNIZE_ALL is defined in NMEAGPS_cfg.h
//
// 'Serial' is for debug output to the Serial Monitor window.
//
// License:
// Copyright (C) 2014-2017, SlashDevin
//
// This file is part of NeoGPS
//
// NeoGPS is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// NeoGPS is distributed in the hope that it will be useful,
```

```

//      but WITHOUT ANY WARRANTY; without even the implied warranty of
//      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
//      GNU General Public License for more details.
//
//      You should have received a copy of the GNU General Public License
//      along with NeoGPS. If not, see <http://www.gnu.org/licenses/>.
//
//=====

//-----
// The GPSPort.h include file tries to choose a default serial port
// for the GPS device. If you know which serial port you want to use,
// edit the GPSPort.h file.

// software implementations of UART
#include <AltSoftSerial.h>
#include <NeoSWSerial.h>

AltSoftSerial piPort; // 8, 9
NeoSWSerial gpsPort(10, 11);

#define PI_PORT_NAME "AltSoftSerial"
#define GPS_PORT_NAME "NeoSWSerial(10,11)"
#define DEBUG_PORT Serial

//-----
// For the NeoGPS example programs, "Streamers" is common set
// of printing and formatting routines for GPS data, in a
// Comma-Separated Values text format (aka CSV). The CSV
// data will be printed to the "debug output device".
// If you don't need these formatters, simply delete this section.

#include <Streamers.h>

//-----
// This object parses received characters
// into the gps.fix() data structure

static NMEAGPS gps;

//-----
// Define a set of GPS fix information. It will
// hold on to the various pieces as they are received from
// an RMC sentence. It can be used anywhere in your sketch.

static gps_fix fix;

//-----

```

```

// This function gets called about once per second, during the GPS
// quiet time. It's the best place to do anything that might take
// a while: print a bunch of things, write to SD, send an SMS, etc.
//
// By doing the "hard" work during the quiet time, the CPU can get back to
// reading the GPS chars as they come in, so that no chars are lost.

static void doSomeWork()
{
    // Print all the things!

    trace_all( DEBUG_PORT, gps, fix );
    trace_all( piPort, gps, fix );

} // doSomeWork

//-----
// This is the main GPS parsing loop.

static void GPSloop()
{
    while (gps.available( gpsPort )) {
        fix = gps.read();
        doSomeWork();
    }
} // GPSloop

void setup()
{
    DEBUG_PORT.begin(9600);
    while (!DEBUG_PORT)
        ;

    piPort.begin(9600);
    gpsPort.begin(9600);

    DEBUG_PORT.print( F("NMEA.INO: started\n") );
    DEBUG_PORT.print( F(" fix object size = ") );
    DEBUG_PORT.println( sizeof(gps.fix()) );
    DEBUG_PORT.print( F(" gps object size = ") );
    DEBUG_PORT.println( sizeof(gps) );
    DEBUG_PORT.println( F("Looking for GPS device on " GPS_PORT_NAME) );

    piPort.print( F("NMEA.INO: started\n") );
    piPort.print( F(" fix object size = ") );
    piPort.println( sizeof(gps.fix()) );
    piPort.print( F(" gps object size = ") );

```

```

    piPort.println( sizeof(gps) );
    piPort.println( F("Looking for GPS device on " GPS_PORT_NAME) );

#ifndef NMEAGPS_RECOGNIZE_ALL
#error You must define NMEAGPS_RECOGNIZE_ALL in NMEAGPS_cfg.h!
#endif

#ifdef NMEAGPS_INTERRUPT_PROCESSING
#error You must *NOT* define NMEAGPS_INTERRUPT_PROCESSING in NMEAGPS_cfg.h!
#endif

#if !defined( NMEAGPS_PARSE_GGA ) & !defined( NMEAGPS_PARSE_GLL ) & \
    !defined( NMEAGPS_PARSE_GSA ) & !defined( NMEAGPS_PARSE_GSV ) & \
    !defined( NMEAGPS_PARSE_RMC ) & !defined( NMEAGPS_PARSE_VTG ) & \
    !defined( NMEAGPS_PARSE_ZDA ) & !defined( NMEAGPS_PARSE_GST )

    DEBUG_PORT.println( F("\nWARNING: No NMEA sentences are enabled: no fix data will be displayed.") );

#else
    if (gps.merging == NMEAGPS::NO_MERGING) {
        DEBUG_PORT.print ( F("\nWARNING: displaying data from ") );
        DEBUG_PORT.print ( gps.string_for( LAST_SENTENCE_IN_INTERVAL ) );
        DEBUG_PORT.print ( F(" sentences ONLY, and only if ") );
        DEBUG_PORT.print ( gps.string_for( LAST_SENTENCE_IN_INTERVAL ) );
        DEBUG_PORT.println( F(" is enabled.\n"
            " Other sentences may be parsed, but their data will not be displayed.") );
    }
#endif

    DEBUG_PORT.print ( F("\nGPS quiet time is assumed to begin after a ") );
    DEBUG_PORT.print ( gps.string_for( LAST_SENTENCE_IN_INTERVAL ) );
    DEBUG_PORT.println( F(" sentence is received.\n"
        " You should confirm this with NMEAorder.ino\n") );

    trace_header( DEBUG_PORT );
    trace_header( piPort );
    DEBUG_PORT.flush();
    piPort.flush();
    gpsPort.flush();
}

//-----

void loop()
{
    GPSloop();
}

```


D.3 Raspberry Pi Read Code

// Source code used from
// <https://www.instructables.com/id/Read-and-write-from-serial-port-with-Raspberry-Pi/> **[PI UART]**.

```
#!/usr/bin/env python
```

```
import time
import serial
```

```
ser = serial.Serial(
    port='/dev/ttyUSB0',
    baudrate = 9600,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
    timeout=1
)
counter=0
```

```
while 1:
    x=ser.readline()
    print x
```

Appendix E - TDOA Calculation and Implementation in MATLAB

Figures E.1 and E.2 show a program written in Matlab which calculates the position of an aircraft using TDOA. Given the positions of the receivers, the time differences of the received signals between receivers, and an initial guess as to where the plane is, the program calculates the position of the aircraft by solving a nonlinear system of equations shown in Fig E.1.

```
function F = tdoa(planexyz,rx,times)
% t12, t13, t14, t23, t24, t34
F(1) = sqrt((planexyz(1)-rx(1,1))^2+(planexyz(2)-rx(2,1))^2+(planexyz(3)-rx(3,1))^2)...
-sqrt((planexyz(1)-rx(1,2))^2+(planexyz(2)-rx(2,2))^2+(planexyz(3)-rx(3,2))^2)-times(1)*3e5
F(2) = sqrt((planexyz(1)-rx(1,1))^2+(planexyz(2)-rx(2,1))^2+(planexyz(3)-rx(3,1))^2)...
-sqrt((planexyz(1)-rx(1,3))^2+(planexyz(2)-rx(2,3))^2+(planexyz(3)-rx(3,3))^2)-times(2)*3e5
F(3) = sqrt((planexyz(1)-rx(1,1))^2+(planexyz(2)-rx(2,1))^2+(planexyz(3)-rx(3,1))^2)...
-sqrt((planexyz(1)-rx(1,4))^2+(planexyz(2)-rx(2,4))^2+(planexyz(3)-rx(3,4))^2)-times(3)*3e5
F(4) = sqrt((planexyz(1)-rx(1,2))^2+(planexyz(2)-rx(2,2))^2+(planexyz(3)-rx(3,2))^2)...
-sqrt((planexyz(1)-rx(1,3))^2+(planexyz(2)-rx(2,3))^2+(planexyz(3)-rx(3,3))^2)-times(4)*3e5
F(5) = sqrt((planexyz(1)-rx(1,2))^2+(planexyz(2)-rx(2,2))^2+(planexyz(3)-rx(3,2))^2)...
-sqrt((planexyz(1)-rx(1,4))^2+(planexyz(2)-rx(2,4))^2+(planexyz(3)-rx(3,4))^2)-times(5)*3e5
F(6) = sqrt((planexyz(1)-rx(1,3))^2+(planexyz(2)-rx(2,3))^2+(planexyz(3)-rx(3,3))^2)...
-sqrt((planexyz(1)-rx(1,4))^2+(planexyz(2)-rx(2,4))^2+(planexyz(3)-rx(3,4))^2)-times(6)*3e5
end
```

Figure E.1: System of Equations for the Difference of Distances as a Function in Matlab

```
clc, clear all, close all;
rx = [4 7 6 3;
      2 7 3 1;
      0.05 0.08 0.05 0.1];

planexyz = [44 2 12];

d1 = sqrt((rx(1,1)-planexyz(1))^2+(rx(2,1)-planexyz(2))^2+(rx(3,1)-planexyz(3))^2);
d2 = sqrt((rx(1,2)-planexyz(1))^2+(rx(2,2)-planexyz(2))^2+(rx(3,2)-planexyz(3))^2);
d3 = sqrt((rx(1,3)-planexyz(1))^2+(rx(2,3)-planexyz(2))^2+(rx(3,3)-planexyz(3))^2);
d4 = sqrt((rx(1,4)-planexyz(1))^2+(rx(2,4)-planexyz(2))^2+(rx(3,4)-planexyz(3))^2);

t12 = (d1-d2)/3e5;
t13 = (d1-d3)/3e5;
t14 = (d1-d4)/3e5;
t23 = (d2-d3)/3e5;
t24 = (d2-d4)/3e5;
t34 = (d3-d4)/3e5;

times = [t12 t13 t14 t23 t24 t34];

guess = [2 46 15];
positions = [];
position_next = fsolve(@(planexyz) tdoa(planexyz,rx,times),guess)
```

d1	41.7469
d2	39.1929
d3	39.8472
d4	42.7037
guess	[2,46,15]
planexyz	[44,2,12]
position_next	[44.0000,2.0000,12.00...
positions	[]
rx	3x4 double
t12	8.5132e-06
t13	6.3322e-06
t14	-3.1895e-06
t23	-2.1810e-06
t24	-1.1703e-05
t34	-9.5217e-06
times	[8.5132e-06,6.3322e-0...

Figure E.2: Position Calculation of the Aircraft Given an Initial Guess

The issue with this program is the initial guess could be incorrect such that multiple iterations of the program yield an incorrect result. A monte-carlo analysis was performed in order to see the tolerance of the program, and the results are shown in Fig E.3 and E.4 below:

```

tolerance = 25e-9;
numval = 3;

t12 = linspace(t12-tolerance,t12+tolerance,numval);
t13 = linspace(t13-tolerance,t13+tolerance,numval);
t14 = linspace(t14-tolerance,t14+tolerance,numval);
t23 = linspace(t23-tolerance,t23+tolerance,numval);
t24 = linspace(t24-tolerance,t24+tolerance,numval);
t34 = linspace(t34-tolerance,t34+tolerance,numval);

guess = [20.5 41.5 27.4];
positions = [];

for i = t12
    for j = t13
        for k = t14
            for l = t23
                for m = t24
                    for n = t34
                        times = [i j k l m n];
                        position_next = fsolve(@(planexyz) tdoa(planexyz,rx,times),guess);
                        positions = [positions ; position_next];
                    end
                end
            end
        end
    end
end

%%
scatter3(positions(:,1),positions(:,2),positions(:,3))

```

Figure E.3: Code Performing Monte-Carlo Analysis in Matlab

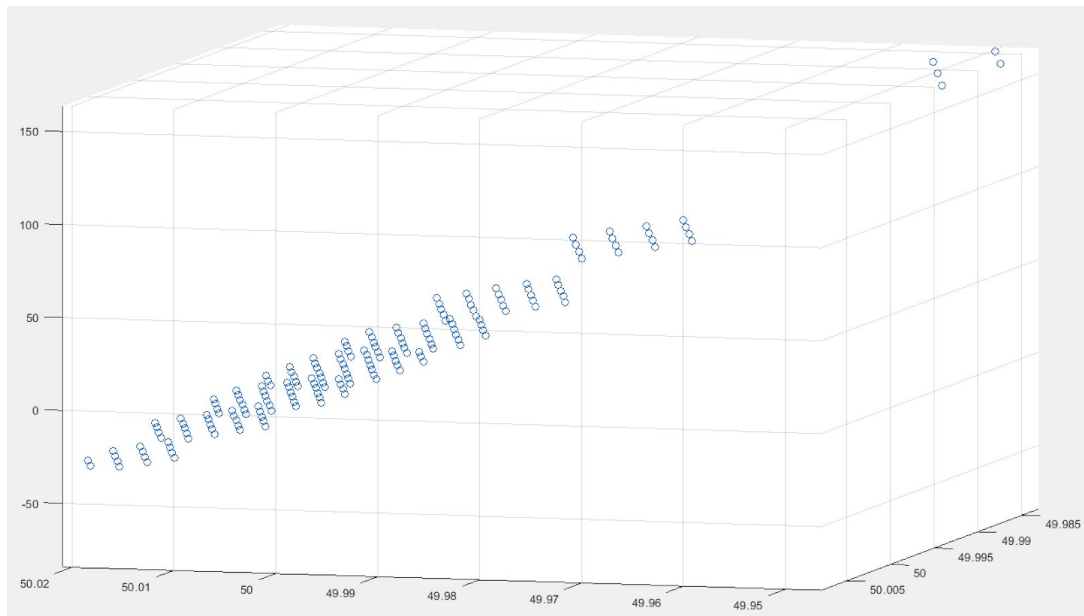


Figure E.4: Results of the Monte-Carlo Analysis

As seen in Fig E.4, a small variation in the time taken for the transponder signal to reach the receiver may drastically change the calculated position of the aircraft. Luckily, we can exclude some of the calculated results since we know the range in which commercial aircrafts fly (below 15 km). Once a

position is calculated, it can be used as the initial guess in the program, and the actual position can be verified (the initial guess will match the calculated position if the initial guess is the actual position of the plane).

Appendix F - Omitted Design Implementations

F.1 Low-profile Antenna

The low-profile antenna was omitted from the final design. The intention of using a low-profile antenna was to implement a compact all-in-one design for our receiver. We decided that the antenna wasn't critical to our design since any linearly polarized omnidirectional antenna would suffice for our design. Notably, the roof antenna in ECEB 5020 was very convenient since it meets all the requirements above while also having excellent line of sight.

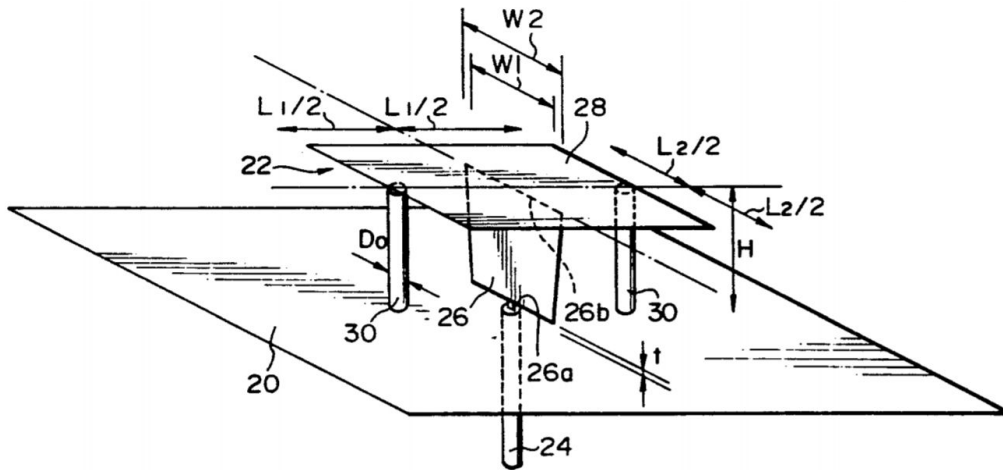


Figure F.1 Low-profile antenna design example by K. Nishikawa. [27]

F.2 PIC32 Microcontroller

As previously mentioned, a change of microcontrollers was a set-back to the design process and modified our expected data flow. The primary reason for this change was due to challenges interfacing and programming the PIC32 using three separate primary development tools: MPLAB X, Harmony, and the Arduino IDE. Although marketed as an Arduino compatible device, the proper bootloader installed, and initial unit tests working, the primary Arduino libraries did not support the PIC32 device. After several weeks of attempting to debug the issues and generate a function code, the decision was made to focus on a demonstration-viable implementation of the Control Unit. The result of the change was a controller that had far less capabilities compared to its predecessor, but had the necessary support to allow for overcoming hardware limits with software libraries that emulated UART and external interrupts.

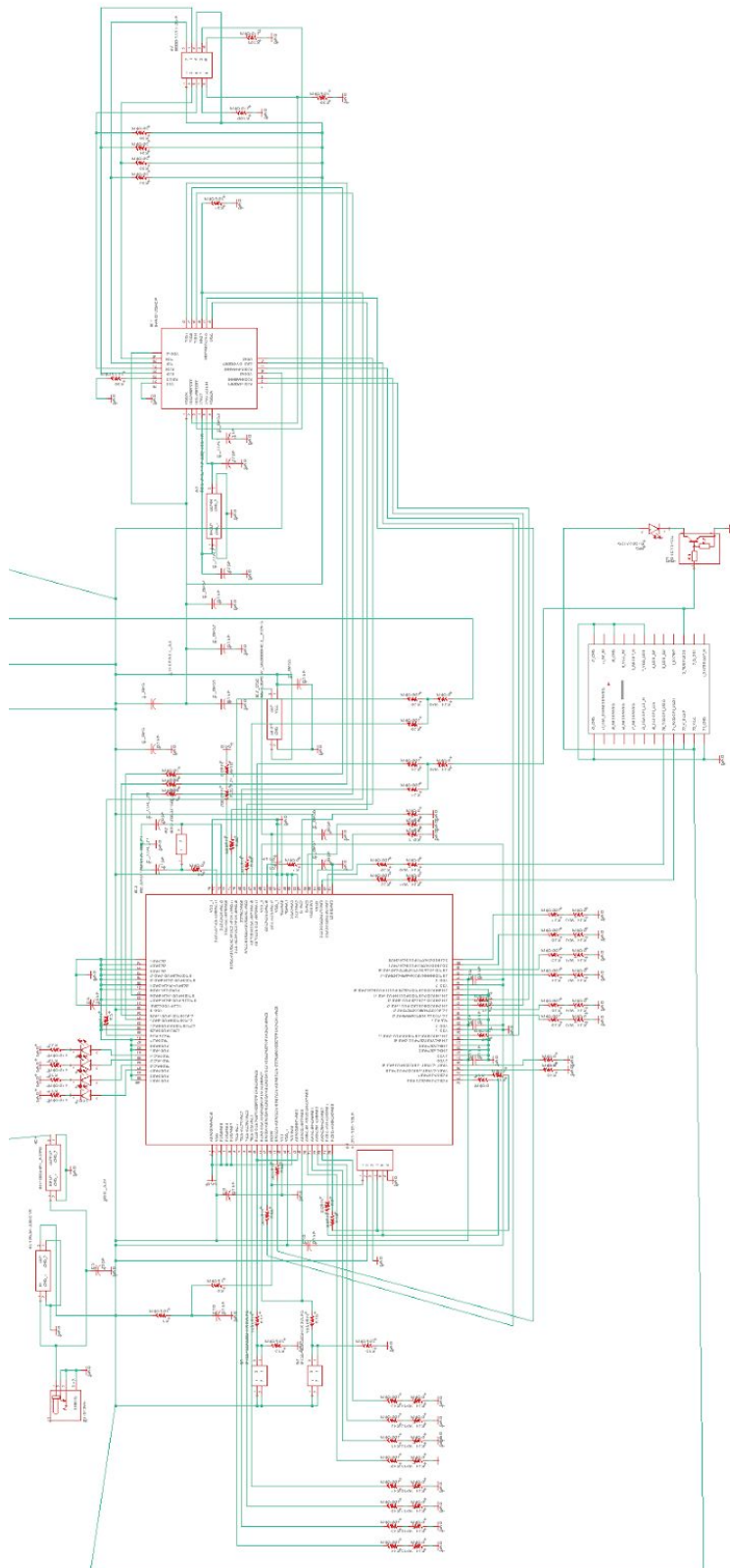


Figure F.1 Control Unit schematic using PIC32