Senior Design 1 Final Report

August 2, 2016

AirBud

Auto Fixed Base Operator

Department of Electrical Engineering and Computer Science

University of Central Florida

Dr. Lei Wei

Sponsored by: Michael F. Young and MacAvionics

Group 4

Jordan Ruhl (Project Manager)	j.ruhl@knights.ucf.edu	CE
Carmen Henriquez Rojas	carmenh@knights.ucf.edu	EE
Mason Maines	mmaines16@gmail.com	CE
Francisco Galue	frangalue@knights.ucf.edu	EE

Table of Contents

Table of Figures

1 Executive Summary

When a pilot chooses to fly there are a few things that the pilot must know for certain. Before a pilot takes off they should know that their microphone, radio, and headset are operational. This is so that they can communicate with other pilots in the area in order to avoid deadly collisions and for communicating with Air Traffic Control at another airport soon after takeoff. Also, as they come into land, a key piece of information to know is the wind direction, wind speed, and gusts at the airport they are landing at. This is because pilots always need to land into a headwind to shorten their landing distance. And if a crosswind exists (and they usually do) the pilot needs to know so they can choose the best runway to land on.

Usually Fixed Base Operators (FBO) are the ones to relay this information to the pilots over the radio, but some airports do not have FBOs. This creates a problem for the pilots who need that very information. One solution to try to mitigate this issue at such airports is a windsock. A windsock is a light and flexible cone of fabric mounted on a mast, usually somewhere along the airstrip of an airport. Windsocks let the pilots know some of the important weather readings, such as wind direction, but they are small and cannot be seen until the aircraft is very close to the airport. On the other hand, there are some automated systems currently on the market that perform task such as broadcasting weather conditions and transmit radio checks, but they are costly and not suited for smaller airports.

The proposed project is a low-cost system that satisfies these two basic needs. This system needs to broadcast important weather information when prompted by pilots in the area. For example, when the system is prompted, the system will broadcast a weather report that includes the latest recorded wind direction and wind speed as well as gusts. This system also needs to perform a transmit radio check for any pilot that consists of recording the transmission from the pilot and playing it back so the pilot knows exactly how operational their equipment is. Therefore, this can be classified as an “Auto Fixed Base Operator” for small airports. This “Auto Fixed Base Operator” would act as a hub of communication for these small airports that do not have a dedicated FBO or weather station. This system would provide a source from which any pilot can obtain crucial weather information or perform any radio communication checks they need prior to taking off and landing their aircrafts. We call it the AirBud.

Our goal is to use our technical experience to connect a weather station and VHF radio through an interface board to a microprocessor that can process all of the necessary information. Using these components, we will build a system that can assist pilots in taking off and landing safely, all while being configurable and cost-effective.

2 Project Description

2.1 Project Justification and Motivation

The idea for the AirBud came to fruition when our sponsor Michael Young, professor at UCF, started flying out of Orlando Apopka Airport where there is no control tower or FBO. He sought out products that would fit his needs to broadcast weather conditions and perform radio checks and found a few. But products he found would cost over $75,000 to install and maintain. He came to our senior design class to pitch the project to us and we accepted the challenge to build a system for his home airport.

Our motivation for this project is the safety of pilots and passengers at these smaller airports with no FBO. When pilots aren't sure of wind conditions they do not know which runway to land on. The airports that don't have a dedicated FBO usually don't have the financial means to fund the expensive automatic weather systems on the market. Our system would become the model for a low-cost effective solution.

2.2 Goals and Objectives

Our goal for this project is to create a low-cost, easy to maintain system for small airports that provides the basic communication services offered by an FBO. The system will be in a hangar at Apopka Airport and when prompted by pilots, through transmission radios, perform the tasks of broadcasting wind conditions and transmit radio checks. Once we had the broad goal of our project defined, we decided on some main objectives to determine the success of our project by.

2.2.1 Wind Conditions Report

The wind conditions report is one of our main functions of the system. When the user/pilot keys the mic a specified number of times, the system should broadcast wind conditions. This wind conditions report should be accurate within ±1 knot and ±5 degrees. It will need to check if the channel is occupied and only broadcast when the channel is unoccupied.

Another feature of this function is to broadcast an updated weather report if the conditions change more than a specified amount. For example, if the system broadcast that winds are 5 knots at 120 degrees, and they change to 10 knots or 150 degrees, the system will broadcast the new wind conditions so that the pilot is always up to date with the most current and accurate conditions.

This also touches on the Crosswind Alert the system will have. A crosswind is when winds blow near perpendicular to a runway, and this causes makes landing more difficult. Our system will detect when a crosswind exists and broadcast an alert. The system should also announce when a runway is "favorable" to land on.

A pilot wants to land into headwind so the length of their landing is shorter. If the system detects winds are more than, say, 5 knots and they are in the direction of a runway, the system should announce that that runway is favorable to land on. All specified parameters discussed in this and following sections are given by the user and discussed more in the “Web Interface” below.

2.2.2 Transmit Radio Check

The second main function of our system is a Transmit Radio Check. Before a pilot takes off, they want to ensure that their mic, radio, and headset work so they can communicate with Air Traffic Control (ATC) and other pilots. Normally, the pilot would contact the FBO and the FBO would respond with a radio check and wind conditions. Our system will be used at an airport without an FBO. When the user keys the mic a specified number of times, the system should prompt the user for a Transmit Radio Check. The system will listen to what the pilot says, and play it back exactly how it was heard and then give the power level of the transmission. This way the pilot can verify their mic and radio are operating normally. During this process, the system will verify that the channel is not occupied before transmitting.

2.2.3 Artificial Intelligence

Both main functions of the system will use the microcomputer as the driving force using software to determine what actions are performed and when, which can be viewed as a form of AI. The microcontroller of choice should have all code written on it and use that as the logic for the system. It should be a “headless” system that does not need any setup process, meaning when the microprocessor has power, the software is running. The microcontroller will take input from the GPIO pins on the board and output in similar fashion. The microcontroller should also be able to act as a database in order to store and log previous weather broadcasts and transmit checks. Thus, the microprocessor should have access to a good amount of flash storage making the system run quickly and effectively. All of this will be controlled by the AI to determine what actions to take and what to broadcast. Our options and selection for a microprocessor are further discussed in Section 3.2.1.

2.2.4 Printed Circuit Board (PCB) Interface

In order to interface the radio base station and the microcomputer we will need to design and build custom circuitry and ultimately fabricate a PCB. This PCB will have all necessary inputs from the radio base station and convert them into usable signals for the microcomputer. For example, the radio operates on 13.8V but a 5V and 3.3V power sources are needed. The PCB will have these power supplies. In turn, it will also create usable signals for the radio that come from the microcomputer. The wind sensors would also be connected to the PCB and the data processed by the microcomputer.

2.2.5 Web Interface

Our device will also be a local web server that serves the function of a graphical interface that the user can get information from the system. The user will be able to specify any parameter and adjust the system. For example, if the user wants to change the number of clicks for the weather report, they will be able to change that from the web interface. This interface will also show a graphic of the runway, a compass overlay, and the wind conditions so that the user can get a graphical representation of the current weather situation like what is shown in Figure 2.1. The user should be able to type in the IP address of the microcomputer and then access the web interface within a small range of the system.

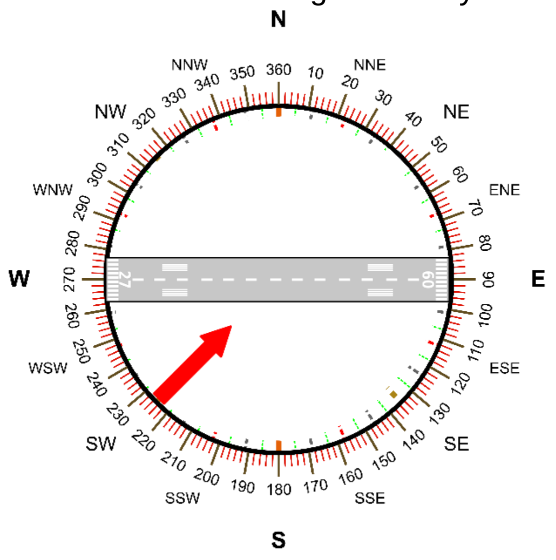


Figure 2.1 GUI Example

2.3 Product Specifications

- The system shall use a power supply, radio, anemometer, and microcomputer.
- The system shall broadcast the current weather conditions when the pilot keys the mic with the specified parameters.
- The system shall update the pilot and broadcast the current wind conditions if they change outside of the chosen parameters.
- The system shall perform a “Comm Check” when the pilot keys the mic with the specified parameters that does the following
  - Prompts pilot for Transmit radio check
  - Records the audio from the pilots comm radio
  - Plays back what is heard
  - Reports the power level of the received signal
- The system shall allow the user to change any parameter below.
  - Carrier Dwell Time (ms): min & max
  - Interval Period between Carriers (ms): min & max
  - No. of clicks for Weather report
  - No. of clicks for Comm radio check
  - When to report gusts over steady winds (kts)
  - When to report variable winds (degree variation)
  - History time for variation (secs)
  - When to announce UPDATE: deg change (deg.) & wind change (kts)
  - Wait time after last report to announce UPDATE (minutes)
  - Runway Headings (degrees)
  - When to announce CROSSWIND WARNING: wind speed (kts) and degrees min and max (degrees)
  - When to announce WINDS FAVOR runway choice #1: wind speed (kts) and degrees min and max (degrees)
  - When to announce WINDS FAVOR runway choice #2: wind speed (kts) and degrees min and max (degrees)
- The system shall have a web IP graphical interface from which the user can read the current winds and make parameter changes.
- The system shall announce crosswind warnings if they are present.
- The system shall announce a favorable runway if conditions fall within chosen parameters.
- The system shall not broadcast if the radio channel is occupied.
- The system shall operate on the airports UNICOM frequency.

2.4 System Description

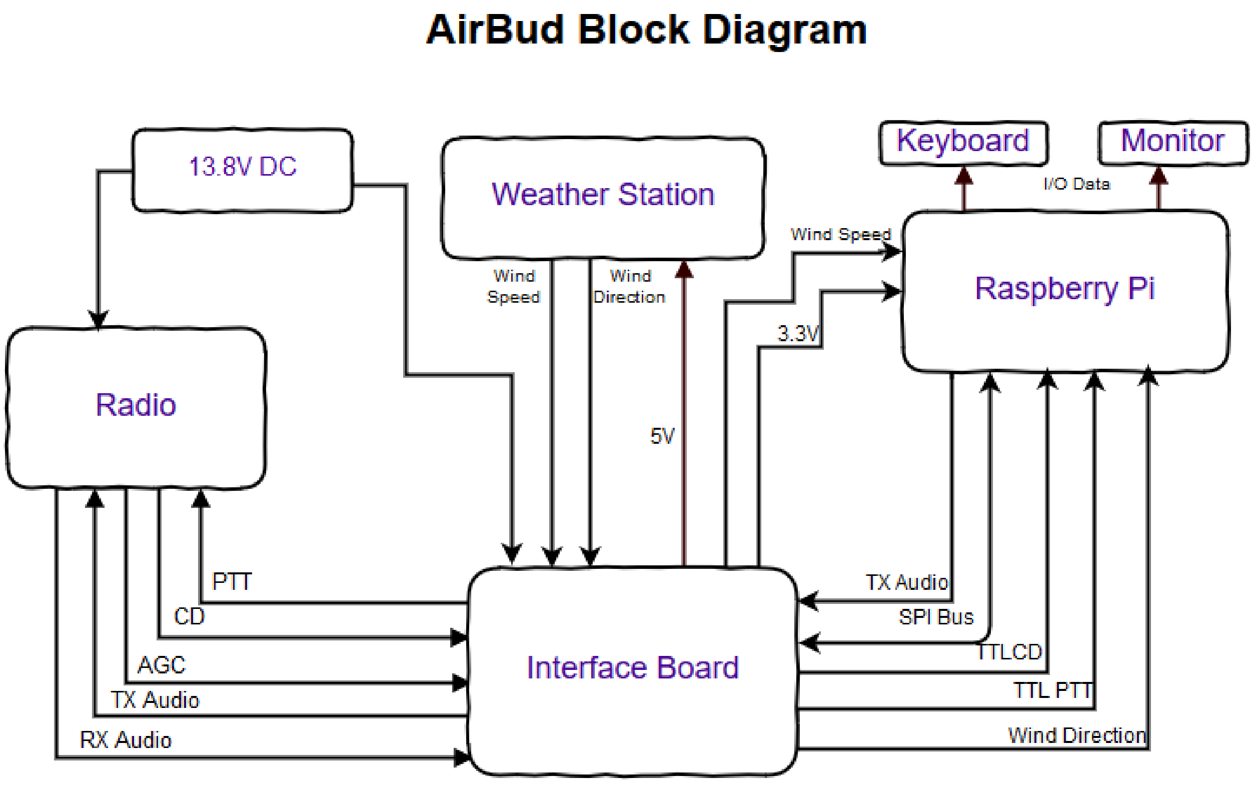


Figure 2.2 System Block Diagram

The block diagram above shows the overall system which is comprised of five main components; the power supply, the VHF aircraft radio, the interface board, the microprocessor, and the anemometer. Every component will be connected to the central interface board for everything from power distribution to data transfer. The items and their specifications are listed below.

2.4.1 Power Supply

The power supply will be a 13.8V 5 Amp power supply.

2.4.2 VHF Aircraft Radio

The VHF Aircraft Radio our system is using is a KX-170B, a solid-state NAV/COMM transceiver manufactured by King in the mid-1970s. It features 720 COMM frequencies and was an industry standard for many years due to its reliability and dependability.

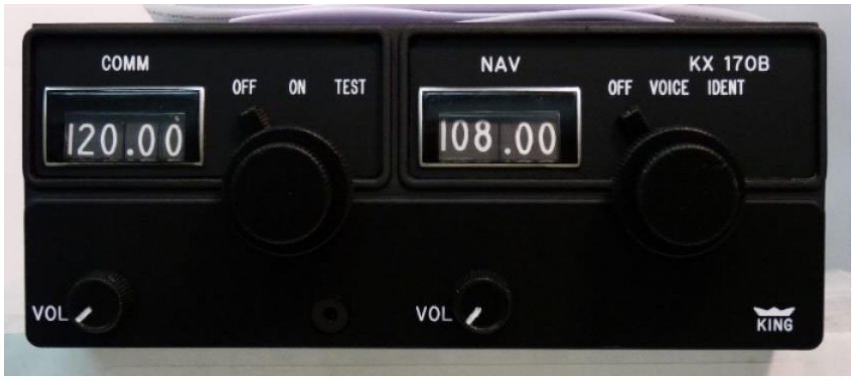


Figure 2.3 King KX-170B Radio

2.4.2.1 Avionics Communication Standards

Because AIRBUD is going to be a repurposed VHF Aircraft Radio there is a specific set of communication protocols, set by the Federal Aviation Administration, that need to be accounted for before any transmission is used. These standards dictate the hardware design, from which signals need to be taken into account prior to transmission to the output frequency of the radio. In the FAA's "REQUIREMENTS FOR 760 CHANNEL VHF RADIO FOR AERONATICAL OPERATIONS" there are many UNICOM broadcast frequencies that can be used by the KX-170B Radio.

Frequency Range (MHz)	System	Purpose
122.700-122.725	UNICOM	Uncontrolled Airport and Aeronautical Utility
122.800	UNICOM	Uncontrolled Airport
122.950	UNICOM	Airport with full time ATCT of full time FSS
122.975-123.000	UNICOM	Uncontrolled Airport
123.050-123.075	UNICOM	Uncontrolled Airport
136.100-Up	UNICOM or AWOS	For Future Use

2.4.3 Interface Board

The Interface PCB Board will be a custom made PCB board used to communicate between the KX-170B VHF Aircraft Radio, Raspberry Pi 3 Model B, and Davis 7911 Anemometer. This interface board will also be responsible for power distribution to all circuits and the Raspberry Pi 3. As well as be prepared for individual signal testing without the radio at hand.

2.4.4 Microprocessor

The microprocessor our system is using is a Raspberry Pi 3 Model B.

2.4.5 Anemometer and Wind Vane

The anemometer our system uses is a Davis Instruments 7911 Anemometer. It is a component of the Weather Monitor II and Weather Wizard III, both of which are complete weather stations also manufactured by Davis Instruments. The 7911 Anemometer features 3 polycarbonate wind cups to measure wind speed and a UV-resistant ABS plastic wind vane to measure wind direction. It comes with a 40-foot long, 26 AWG cable that ends with an RJ-11 connector. It can measure wind speeds up to 173 knots (200 mph) with a 1 knot resolution and a ±5% accuracy. It can also measure wind direction from 0 degrees to 360 degrees with a 1-degree resolution and a ±7% accuracy.

2.4.6 Monitor and Keyboard

These peripherals will be connected to and used by the Raspberry Pi. They can be of any variety as long as they are connected through via HDMI cable for the monitor and USB for the keyboard. They will be used to interact with the Raspberry Pi and view or modify code and graphical user interfaces.

2.5 Design Constraints and Standards

2.5.1 Time Constraints

This project will be a complete working product by the end of Senior Design II in Fall 2016. This creates a limited timeframe for the team to work with. The total time for this project is about 28 weeks, and to develop, design, build, and test a system of this nature will take diligence to complete in that amount of time. The plan is to have a working prototype at week 11, at the end of Senior Design I. This in and of itself is a lofty goal and requires teamwork and persistent hard work.

2.5.2 Budget Constraints

The team is comprised of four college students with limited incomes, which limits the solutions, but also provides motivation to make this as low-cost of a system as possible. Our primary sponsor has provided us with $250 towards our project and that has been set as the target cost for the entire system. Another sponsor has offered an additional $500 past the primary sponsor's initial budget. If the need arises, the team can use up to $750 before having to use personal funds. This provides us with a good financial base to build our project on, but without having unlimited funds, the team will have to be mindful of the limited budget.

2.5.3 Related Standards

Below are sections dedicated to standards that are a part of our project at this point.

2.5.3.1 USB

USB, short for Universal Serial Bus, was developed in the 1990s and standardized cables, connectors, and communications protocols used in a bus for communication between electronic devices. Many microcomputers come with USB ports which are usually USB 2.0 HSIC. HSIC, short for High-Speed Inter-Chip, eliminates the analog transceivers found in normal USB and was adopted in 2007. Utilizing HSIC allows the USB to use 50% less power and 75% less board area and have a throughput of 480 Mbit/s.

2.5.3.2 SPI

Serial Peripheral Interface (SPI) bus is a synchronous serial communication method over small distances primarily used for embedded systems. SPI was developed by Motorola and quickly became a de facto standard among the industry. SPI devices communicate in full duplex mode using a master/slave architecture as long as there is only one master. In our case, the microcomputer will act as the master and the AGC voltage, RX audio, and wind direction. This bus will be helpful as it can provide the microcomputer a bit stream of data to interpret.

2.5.3.3 WAVE File

We will be using the WAVE format standard for storing audio data. The WAVE file standard was introduced as a joint standard from the IBM Corporation and the Microsoft Corporation in the “Multimedia Programming Interface and Data Specifications 1.0” standard document released in August of 1991. The WAVE file standard in particular was introduced as a substandard of the RIFF, or the Resource Interchange File Format, standard for storing multimedia. While old we chose this standard because it is the most common form of uncompressed audio, and is recognized across all systems as well as multiple audio centered programs. By using the WAVE format standard, we did not have to commit to a certain form of audio compression standard. This will allow us to directly interface with the raw audio data, as well as compress the data using any of form of audio compression standard in the future if we feel we need to compress the data.

The WAVE file format standard organizes the data it stores using what the RIFF standard defines as “chunks”. Each of these chunks, while having no particular set order to where they are located within the file, contain their own specific sets of fields and parameters. For the WAVE file format, the standards indicate that there are only three chunks that are required for any WAVE file; these three chunks include: the Header chunk, the Format chunk, and the Data Chunk. While there is no set order for these chunks, the adopted standard is to write each of the chunks in the order they were introduced above. This allows for readability, and the ability for programs to know where to look for certain information without the need of including more header information about where data is located. This reduces the file size and the speed in which the file can be processed. Two optional chunks, the List chunk and the Info chunk, can be included in a WAVE file to document the order in which the various chunks appear in the current WAVE file. These two chunks are usually place right after the Header Chunk and are only included for compatibility with software that did not follow the suggested chunk order adopted by the industry.

Each of the required chunks outline the basic needs of any multimedia player. The first is Header Chunk which specifies the multimedia format standard used by the file as well as the particular substandard of multimedia used. In the case of the WAVE format standard, the RIFF standard for multimedia, and the WAVE substandard are always included in the Header chunk. Along with these two fields the Header chunk contains the size (in bytes) of the rest of the file. The next required chunk, the Format chunk, is uses to specify the format in which the WAVE file was being recorded. Along with the standard chunk id and chunk size that outlines which chunk is being read and how large the chunk is, these fields are almost all variable and include the sampling rate, byte rate, number of channels, and bit resolution used to record the audio data. The only other major field to note that is included in the Format chunk is the Audio Format field which is used to specify what audio recording standard is being used to record the data. Because we are using an Analog to Digital converter to sample the audio we are recording, we will use the Pulse Code Modulation, or PCM, standard or audio recording. Lastly the WAVE file format standard requires the data chunk which is responsible for storing the raw audio data sampled in the audio format specified in the Format Chunk. This data is encoded in two’s compliment format and then stored in the Little Endian format.

2.5.3.4 Pulse Code Modulation

We will be using the Pulse Code Modulation audio format standard for recording audio data. This standard is used to digitally represent the analog audio data being recorded. We chose to use the PCM standard for recording audio data, as it directly coincides with how we will be receiving data from the analog to digital converter. The PCM standard requires taking a sample of an analog audio signal and representing it using a decimal number. Because most analog to digital converters use PCM to sample analog data, we will be using this standard to record audio data in the File Format standard we chose to store it.

2.5.3.5 Radio Communication Phraseology/Techniques – P-8740-47

One very important aspect of working with aircraft radio and information communication is using the proper terminology. Communication in the avionics world is drastically different that of every day. This is due to the large amount of noise that a pilot can experience while in the cockpit. For this same reason different information has different methods of communication in avionics communication.

When wind direction and wind speed are being communication it is important to take note that each number must be announce in order. This same process applies to multiples of instances in aircraft communication, but wind speed and direction are the ones that the system will be communicating.

Information Being Relayed	Example Message Content	Non-Avionic Pronunciation	Avionic Pronunciation
Wind Direction	135º	One hundred and thirty-five degrees	One three five degrees
Wind Speed	30 Kts	Thirty knots	Three zero knots

As shown the information being relayed is done so in individual numbers. This is to avoid misinformation in the message. The reason is because one misunderstanding in the information between system and user can result in the pilot using the wrong landing direction, and that can have dangerous, and even lethal results.

2.5.3.6 Traffic Advisory Practices at Airports Without Operating Control Towers – 90-42F

The Traffic Advisory Practices at Airports Without Operating Control Towers defines AIRBUD as a UNICOM system, under the guidelines that it is a “nongovernmental air/ground communication station which may provide information at public use airports.” (Traffic Advisory Practices at Airports Without Operating Control Towers, § 4 (1990). Print.)

In this standard is it state that UNICOM stations can provide wind direction and wind speed information to pilots upon request. Regardless if the UNICOM station shares the same operating frequency as the Common Traffic Advisory Frequency, CTAF. This is important because in small airports like the one AIRBUD will operate in the CTAF can common be assigned to a designated UNICOM frequency operating range. This is ideal for a small airport as the small amount of air traffic can be managed by commercial systems like AIRBUD, but in larger airport where the CTAF is different from the UNICOM frequency this can present itself a challenge as the pilot would have to switch between frequencies to communicate with the UNICOM system.

This standard also calls for communication with UNICOM stations of at least 10 miles from the airport the station is in. This forces out system to be able to operate at such distances in order to comply with standards.

3 Background Information

3.1 Existing Products

There are products currently on the market that meet or exceed our product requirements. They are autonomous or unmanned control tower like services that provide pilots with necessary information like the weather conditions and radio checks our system will provide. These products usually provide way more services for the pilots like monitoring traffic in the airspaces and broadcasting that information as well. Also, there is the more traditional approach of having a dedicated FBO at the airport in question. These are all fine mitigations to the issue of getting necessary information while piloting an aircraft, but they all have a major flaw; cost. These systems have a high acquisition cost and some also require a maintenance cost. This can be a justified cost for airports that either really need the information for air traffic such as remote airports for rescue missions or supply drops, or smaller international airports that don't have enough air traffic to justify the cost of an entire control tower. But for the small regional airport of Apopka, these options are just too expensive.

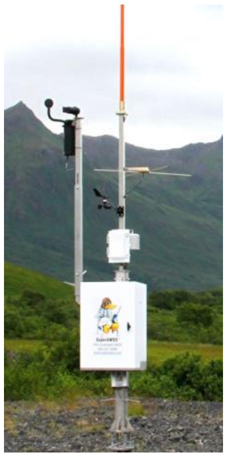


Figure 3.1 Potomac Aviation Micro Tower

The first similar product is the Potomac Aviation Micro Tower (Figure 3.1). The Micro Tower is an all-in-one system that operates on the area's CTAF frequency (Usually UNICOM) and provides the same core services that our system will provide. The Micro Tower can broadcast wind conditions, altimetry, visibility, and runway advice. The Micro Tower can also perform the same comm check our system will have by recording and playing back a pilot's transmission and giving the power level of that transmission.

Where this system exceeds is its AI capabilities with all that information. For example, if the Micro Tower detects an occupied airspace, it can make its messages shorter, as to not take up too much time in the broadcast so other pilots can communicate when necessary. Also, if the visibility is greater than 10 miles, the Micro Tower won't broadcast that with the weather advisory message. The other benefit of this system is how it's powered. The Micro Tower is completely solar powered, meaning it can be set up anywhere in the world and not have to rely on a power source. These features are why there are 150 Micro Towers set up in the world with 100 of those in the United States.

For many rural airports or landing sites, this is an optimal solution. Places that don't necessarily need or can't afford the multiple thousands of dollars cost of dedicated weather and broadcasting equipment can use this and not have to worry about finding power sources or network connectivity. Where the Micro Tower fails Michael Young's needs is the cost. When Young inquired about the Micro Tower, the quoted price for the product was well over $75,000. This was way too much money to spend at an airport such as Apopka with as little air traffic as it has, especially for one individual, which is why we are creating an incredibly more cost-effective solution.

Overall this solution far exceeds the needs of a small airport such as the one in Apopka, and the price is similarly outlandish when the fact that they are mostly self-funded is taken into consideration. Michael Young could not afford the expensive Micro Tower and wanted a similar product without the cost, which is why we are building this low-cost solution called the AirBud.

3.2 Main Control Unit

3.2.1 MCU Options and Selection

3.2.1.1 Arduino Uno

The Arduino Uno is a microcomputer based on the ATmega328P. It has an operating voltage of 5V, 14 digital I/O pins (6 of which provide PWM output), 6 analog input pins, a USB connection, a power jack, a reset button, and runs at 16MHz. The ATmega328P, an 8-bit AVR RISC-based microcontroller, provides the Arduino with 32KB of flash memory, 2KB of SRAM, and 1KB of EEPROM. It also supplies 32 general purpose registers, serial programmable USART, SPI serial port, and an 8-channel 10-bit A/D converter. With the integrated A/D converter and reliably tested and documented hardware this is a good option.

3.2.1.2 Raspberry Pi 3 Model B

The Raspberry Pi 3 Model B is a fully fledged microcomputer with a 1.2GHz 64-bit quad core ARMv8 CPU, 1GB of LPDDR2 RAM running at 900 MHz, 4 USB ports, and naturally runs a distribution of Linux called Raspbian. Other features of this microcomputer include 10/100 Ethernet, 2.4GHz 802.11n wireless, Bluetooth 4.1, 3.5mm audio jack, and a microSD card slot. The Pi's main line of communication, at least for our purposes are the 40 GPIO pins which also provides support for SPI communication. It runs on 5V and can output 3.3V or 5V through the GPIO pins.

The sheer computing power and endless documentation of the Raspberry Pi 3 make this a good option.

3.2.1.3 MCU Selection

When deciding which microcomputer to use we went back and forth deciding what would be best. We knew our system would have a couple analog inputs to the microcomputer so an Arduino Uno would be best suited for that purpose. On the other hand, the immense computing power of the Raspberry Pi 3 was enticing. We ultimately decided on using the Raspberry Pi 3 for our system for a few reasons.

The first and possibly biggest reason for using the Raspberry Pi 3 is the operating system it naturally runs, Raspbian. Raspbian is a distribution of Linux and would allow us to code our project in Python. Using python would allow us to use a database framework like Django to store any data on the Pi itself. The software selection process is further discussed in Section 3.2.3, but having such a malleable piece of hardware to write software on was a useful feature. Using the Raspberry Pi 3 as a basic Linux computer allows us to possibly set up a graphical interface in the future, while also providing us with a headless command setup now.

Another feature on the Raspberry Pi 3 that contributed to its selection is the 2.4GHz 802.11n wireless capabilities and the 10/100 Ethernet port. This allows us to easily install new software and packages directly from a webpage (as long as there’s an internet connection) and set up a local web server. One of the goals of this project is to have a web interface that the user can modify parameters from. Having the Ethernet port lets the user plug in their computer and access a web interface we set up that’s run on the Raspberry Pi 3.

The last big reason we chose to use the Raspberry Pi 3 is for the amount of GPIO pins and the accessories we can use with them. There are a lot of connections going into and out of the microcontroller and having 40 GPIO pins is useful. Without using the Arduino Uno and its native A/D converters, we knew we would have to use an external A/D converter. We found the MCP3008 that worked perfectly on the SPI bus provided by the Raspberry Pi 3 GPIO pins. Using this setup and leaving a multitude of pins open for other uses as we need them is vital to the system’s success.

3.2.2 Communication Protocols

The nature or VHF Radios in aircraft communication has become critical in the communication of information between traffic control towers and aircrafts all around the country. Radios have communication protocols that need to be addressed prior, during and after communications. These protocols dictate who communicates, which signal propagates in the given frequency band and if your VHF will listen or transmit. These signals will need to be filtered and manipulated in such a way that the Raspberry Pi 3 will be able to interpret them and use them to follow adequate protocol for communication.

3.2.2.1 Push-to-Talk

PTT has been a standard of two-way radio communication for quite some time. The nature of half-duplex communication systems is that there has to be some sort of signal flag to alert the transceiver that it is time to stop receiving and ready for transmission. The reason it is called push to talk is that the action required for this stage is top push the button on the microphone and it has been colloquially applied to the vernacular. What the button actually does is pull the PTT relay in the KX-170B radio to ground, thus setting it into transmit mode. For the case of this system what will be done is that through one of the GPIO pins of the Raspberry Pi 3 and a PTT circuit in the interface board, the MCU will ground the relay and set the radio into transmit mode.

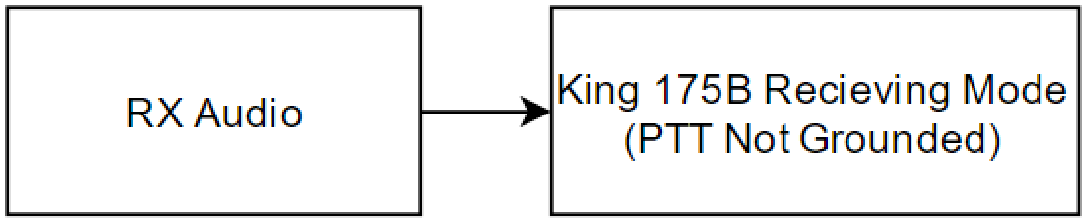


Figure 3.2 KX-170B Receive Mode

Because the KX-170B VHF Aircraft radio is a half-duplex communication system it can only do one of the two communication functions at a time. When the PTT is not grounded the KX-170B is in 'Receive Mode" and can receive incoming audio signals.

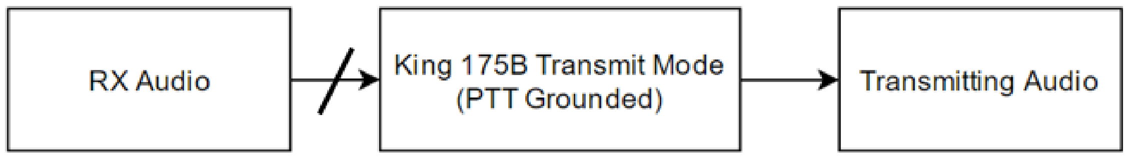


Figure 3.3 KX-170B Transmit Mode

But when the PTT is grounded the KX-170B switches to 'Transmit Mode'. In this mode the system cannot process any received audio and any communication to it is essentially lost.

3.2.2.2 TX Signal

Of the two functions a half-duplex system can make during communications is to transmit a message to a receiver. The TX signal is signal that is send out and carries that transmitted message. For the intended operation purposes of this airport the transmission frequency will be 123.05MHz. During transmission the half-duplex system will by nature be unable to receive any kind of transmissions.

3.2.2.3 RX Signal

The received signal from the KX-170B Radio will be sent to the interface board through soldering into the positive end of the volume potentiometer. This signal will be received at 123.05MHz as assigned to the airport that we will use the system in. The importance of this signal is that it will allow the Raspberry Pi 3 to interpret what the pilot wishes to do with our device, and which steps will be taken later as a result of that.

3.2.2.4 Carrier Detect

Carrier Detect, in communications, is present in the squelch circuit with the function of suppressing the audio output of a receiver in the absence of a higher amplitude and strong input audio signal. The squelch can be opened, allowing all audio signals entering the receiver tap to be heard. This circuit can be useful when attempting to hear weak or distant audio signals. Squelch operates alone on the detection of the strength of the signal; when a device is set to mute, there is no audio signal present. Knowing if there is a carrier detect present, at the squelch, will allow the MCU know when there an audio signal present.

3.2.2.5 Automatic gain control (AGC)

Automatic Gain Control is a closed loop-feedback circuit where a signal is fed into and it’s expected to maintain and regulate to certain level of amplification. This signal can be sound or radio frequency. The AGC can give us two different cases for output. The first case is if the level of the input signal is too low, the designed system will output an amplified signal to the desired level. The second case is if the input signal is too high, the designed system will output a lowered signal to the desired level as well. The purpose of this system is to maintain a constant level for the output signal giving a wider range of input signal levels. AGC is commonly used is radio receiving to help equalize the desired average volume due to different levels received in the strength of signals and fading of the same. One of the consequences of not using an AGC is seen in the relationship between the signal amplitude and the sound waveform – the amplitude of this signal is proportional to the radio signal amplitude. The information contained by the signal is carried by the changes of the amplitude of the carrier wave. If the circuit were not fairly linear, the modulated signal could not be recovered with reasonable fidelity. However, the strength of the signal received will vary widely, depending on the power and distance of the transmitter, and signal path attenuation. Overall, the AGC circuit keeps the receiver's output level from fluctuating too much by detecting the overall strength of the signal and automatically adjusting the gain of the receiver to maintain the output level within an acceptable range.

3.2.3 Language Options and Selection

3.2.3.1 MCU

For the main control unit, or MCU, there are a few options as far as what language to choose. Since we are utilizing the Raspberry Pi 3 for the MCU the first priority, was making sure that the programming language that we selected was directly compatible with the Raspberry Pi and had libraries in which to access the multiple General Purpose Input and Output pins, or GPIO pins. Having a library for the Raspberry Pi's GPIO pins allows us to not have to work from the ground up, and strictly focus on how we are going to program the GPIO pins specifically. This saves us a lot of time and effort that we don't have to put into a lot of code that's only purpose would be to allow us to access the pins.

For this design we chose to go with Python as our programming language for the MCU. Using python solves the initial requirement of having a default library for interfacing with the Raspberry Pi's GPIO pins through the RPI.GPIO library. This allows for basic reading and writing to the pins without having to create those initial functions ourselves.

Another reason we chose python as our main language was because the Analog to Digital Converter, known as an ADC, that we are choosing has a python library that allow for easier reading and writing directly from the chip without having to do a lot of initial handshake messages and procedures to receive and send data. Because reading from most particular ADC's can be complicated, as they have certain bit patterns in which are needed to configure and choose which of the devices' functions are being used, it is nice to have an extra bit of encapsulation in which instead of building these bit patterns ourselves, we can simple call a read or write method. This not only shortens the amount of code written by us but again allows us to focus more on the actual implementation of our system rather than having to deal with a lot of headache simply reading from the ADC. This library is also open source so it is free to use and heavily supported by the community in case we run into any issues.

Python was a good choice compared to other languages such as C as not only is it inherently Object Oriented and allows for a more modular structure to our code. The Object Oriented nature of Python allows us to create objects in which to delegate the functions of reading and writing to certain components and sensors. This also allows us to give control of certain components to certain objects and much more easily debug our code. Python is also a scripting language which makes it highly flexible in where and how it is actually run and implemented. This means that no matter how we structure the system and integrate the various other components (i.e. the HTTP server, DCHP server, etc.) the usage of our code can be kept relatively independent. This allows us more freedom to change certain modules and components in the system if we have to and not have to overhaul our python scripts too much. In other languages like C, it can be much more difficult to configure the code with all of these different components, as it has to be recompiled and is only set to run a certain way. There is not a lot of flexibility there, which is ideally what we find to be valuable in the structure of this system.

3.2.3.2 Web Server

As the main focus of this project will be focused around the backend of the system capturing weather data, parsing that data, and formatting it in a way in which to be broadcasted back out, we did not want to have to put a lot of unnecessary effort into designing and implementing a huge relational database. Doing this would require a lot of work just grabbing the required data and passing that data to and from the front end of the system. This is where we started looking into using a web framework that would allow for a Model View Controller, style of web design. Using such a framework would allow us to simply pass the necessary information from our models to our views and not have to make complicated PHP scripts to send queries back and forth to the database. Not only will using such a framework make our lives easier, but it will improve the performance of the system as data will be populated into the views as they are created as opposed to loading the web page and then recommunicating to the backend for the required data.

With these ideas in mind, we decided to go with the Django Web Framework for the backend and the AngularJS Web Framework for the front end. These two frameworks are MVC, or Model View Controller, frameworks. Django is created and utilized in Python, html, CSS and JavaScript. This MVC framework works perfectly without system since we will be using Python for our main MCU functions and scripts to interface with the GPIO pins. Another major reason for using this framework is that it abstracts a lot of the database behind models represented as Python objects. This means that we can write our code in an object oriented fashion and have the framework build the database tables for us behind the scenes. This is helpful for us in a couple of ways, the first being that when interacting with the database, we can treat each table record as an object that we can interact with. This means we can change the fields of that record as if they were general variables belonging to an object. Secondly this allows us to run functions on the records in our database in which we can sort, search, filter and build custom criteria in which to get, update and store records into the database without having to write a line of SQL. This leads me to the main strength of this framework, which is the fact that we can choose almost any database we want for the backend and not have to change a line of code for our models. This again adds to the flexibility of the system, as we can change out databases at any point if we want something lighter, faster, smaller, or more robust.

Besides just having the database abstracted, we can import the schema of our models directly into our Views and again use the database table records as if they were objects and directly inject them into our html code. This means that in the HTML code I can use the frameworks custom tags to preload the needed data and utilize the database objects directly in the HTML instead of having to make a client, formulate a SQL statement, make a call to the backend and wait for the data to come back. The framework also makes use of custom tags to reduce the amount of html we write. By utilizing certain template tags, we can repeat a block of html for any number of database objects, which prevents us from having to write out multiple blocks of the same html code and do extra processing on data received from the database to populated each of those blocks separately.

The last major reason for Using Django lies in the fact that there are loads of useful add on modules that will aid us in development. Two that we will be using for this system include Django Celery, and Django REST Framework. Django Celery is a job queue that will allow us to asynchronously queue, schedule, and run functions we write as tasks. This will greatly improve how resources are managed on the backend, which is immensely important since the Raspberry Pi is a powerful but small machine as far as memory is concerned. Because we want the majority of the resources on the Raspberry Pi to be handling the actual software operations of the system, instead of primarily the web server, it is important to make the webserver as light and fast as possible on the server side so we do not bog down the system with http requests and web processing.

Django Rest Framework is a plug in that will allow us to quickly write and utilize a RESTful API in which our frontend can “hook into.” REST stands for “Representational State Transfer” and is used to create web endpoints that provide database data to frontend web frameworks. This plugin to Django will allow us to take advantage of Django’s ability to abstract away the database as objects, and provide that information to our frontend as a JSON, or JavaScript Object Notation, object that can be utilized in the JavaScript as if we had gotten it directly from Django. This is important as it is the main and only point of communication between Django and our frontend framework, AngularJS.

AngularJS is another MVC Framework whose strong suit lies in its ability to create single page web applications. It might seem like overkill to have two different web frameworks for the backend and frontend of the system, however this is extremely important since we want to remove as much stress from the Web Server on the Raspberry Pi as we can. The more time the Raspberry Pi is handling web requests; the more time it is not properly managing the system resources that are essential to the system. This is where AngularJS comes in; AngularJS uses JavaScript to break up a single page and separate that into multiple views. It does this by grabbing data from a RESTful API and storing that information as models. It then uses these models in conjunction with its views to display the requested information on the screen. The real selling point here is in the fact that ALL views are loaded at once, but the framework only shows the selected view and only pulls in the data it needs, when it needs it. This allows us to consolidate all the views from the Django end into one view on the AngularJS end and have the client’s browser handle which view and information is being requested. In this architecture, instead of requesting a new view from the Raspberry Pi every time and waiting on the response, all of the views are already loaded at once on the client machine and the browser simply switches what content is being display and what content is hidden. This drastically reduces the traffic and computation on the Raspberry Pi and pushes all of the responsibility to the client’s browser (usually located on a machine more suited to handle the load).

Database

Even though the database will be mostly abstracted away from the system by the Django Web Framework, it is still an important part of the system that must be carefully chosen. Just looking at the line of Open Source SQL products there are many options ranging from very advanced and feature rich to simple and lightweight options. For this project we will not nearly need as many features as PostgreSQL provides as the models for this project will not be overly complex. Also since there is not really any form of user related security, it is a bit overkill to utilize a lot of the security features that come with a lot of SQL databases. Because this system can be classified as “embedded” and security are not as important in this system, the focus on what database we needed focused more on speed, memory load, portability. These criteria made SQLite the perfect option to use for this system. SQLite is stored in a single file format as opposed to the traditional database design. This allows for incredible portability, as simply backing up this file at certain stages will give the system administrator a lot of functionality as far as rolling back the database if it were to get corrupted. Also the fact that the file can be copied to a completely different system and, as long as it is configured correctly, will be a complete copy of the database. Secondly, as the name implies, this style of database is extremely light weight and does not take up nearly as much space as traditional database. Its small size allows for the database to be very fast as well, making this database the perfect selection for this system.

If there are any fallbacks to mention here, we would probably have to say that this particular database does not include security and can only allow one “write” action to happen at one time. As discussed above, the security features of the database are not necessary for this project and this drawback can be ignored. As far as only being able to write one entry at a time, this is not so much of a problem for us considering that writing to the database is only done under certain circumstances in which data is being read from the various components of the system. Here we do not have to really worry about multiple entries being written at the same time since each request sent to the system can only be processed one at a time anyway. This problem will only be an issue if multiple users are trying to change the configurations settings at the same time over the User Interface. This again is not a major problem since the Raspberry Pi will only be on a local, very small network and not normally accessible to the outside world. This in and of itself is a form of security measure due to the fact that not many people will be changing the configuration settings, and have to be on the Raspberry Pi’s local network run by the Raspberry Pi’s DCHP server to even access the page.

Job Queue

Lastly, we need to talk about the Job Queue software that the Django Celery Module (discussed above) will be utilizing. There were two major options as far as which job queue to use, RabbitMQ, and Redis. Both of these options are inherently compatible with the Django Celery plug in and are both recommended options by default. RabbitMQ is a very fast, lightweight, and persistent job queue that would be the better option as far as the most lightweight option. However, in this case we decided to go with the more robust Redis job queue for a couple of reasons. Redis has a few more options as far as configurability in which we have the potential to have multiple jobs ques clustered together if we chose. We do not foresee the issue of overflowing the one job queue to be a problem, however it is nice to know we have the option to play around with different configurations to see what gives us the best performance. The main reason for choosing Redis was in the fact that Redis can also serve as a Key-Value pair dictionary that is store in the systems persistent memory. This can come in handy since we do not necessarily want to implement the Python jobs to read and write to the GPIO pins to be governed by the Web server and the Django framework completely as that doesn’t really make sense from a structural standpoint. The Redis Key-Value store would allow for standalone python services to fetch the various sensor data from the external components and then store them to the Redis Key Value Dictionary. Then the tasks governed by the Django Celery plug in can asynchronously grab these values from the stores they were saved in and save them to the database (that is governed by the Django Web Framework). This makes more sense structurally and allows us to keep a lot of the functionality separate and modular. This will make testing and debugging the system much faster and easier to do. While RabbitMQ may be lighter and faster on the system, the extra bit of functionality we get from Redis will be more useful to us in linking all of the different parts of the system together.

3.2.4 Text-to-Voice Software

One of the most important pieces to this system is the Text to Speech software. This software will be what takes all of the weather and transmission data that is taken in from all of the sensors and creates the voiced broadcast that will be played over the radio channel and heard by the pilots. Needless to say that there are a few very important specifications that the text to speech software must follow. First and foremost, the voice that is created by the text to speech software chosen must be indisputably clear and concise. Even if a certain Text to Speech software creates an understandable output over laptop speakers, it must be understood that this signal will be running through various amplification and compression circuits and will eventually be broadcasted over the radio channel as radio waves. Throughout this process there can be a lot of interference and the signal may be heavily influenced before it is even heard by the pilot. This is why it is so important to have a clear and clean output. Secondly, the output of the Text to Speech software must be configurable. This means that the language, pronunciation, and speed of the output signal must be configurable. A clear output is still no good if the output is too fast to hear or mispronounced. Also some voice settings may be more appealing to other users and improves the user experience of the system. The next requirement for the Text to Speech software is that the output must be easily stored in a file. This is so that the history of the system can be kept track of if the user wants to see previous broadcasts. Lastly, and arguably most importantly, whatever Text to Speech solution we choose must be able to be used without an internet connection. This system will be placed in an environment where a solid internet connection may not be reliable. In that case we cannot have a major part of the system be reliant on something that may not be provided.

Looking at the above requirements, we came across a few good options. IVONA Text to Speech was easily the clearest can best voice out of all of the Text to Speech software we looked at. It was highly configurable and had many configurations that could be set including nationality, language, gender, and pronunciation. We would have easily gone with this solution, except for the fact that it is a service provided over the internet, the output could not be saved to a file by default, and cost a good bit of money a month for the service. Secondly we looked at Festival TTS. This particular solution is free, open source, is directly compatible to Linux, and run directly on the Linux system. Festival was by far the most configurable system we saw with tons of different configurable voices. What I liked most about this solution is that other language packs found or created by the community could be imported into the system, as well as tested on the fly. The downside to Festival came in the form of its clarity and storability. While its configurability is unparalleled, it is not the clearest solution and difficult to work with. It is the trickiest solution to set up and the lack of clarity knocks it out as a viable option for this system. Lastly, there was PICO TTS. PICO is a stripped down, barebones version of the Text to Speech project used in googles android products. While very simplistic, this solution is open source, free, and can be run directly on any Linux system (including the Raspberry Pi). While the initial set up can be a little unorthodox, it is only a matter of downloading, unpacking and “make installing” the correct package found online. Anyone familiar with the Linux operating system will not have too much trouble here. We ultimately chose this option because the voice output is clear and can be store directly as a wav file. While there is not a ton of configuration as far as the voice there is enough for our purposes here. The default gender of the voice is female and there really isn’t anyway to change the gender. However, there are a good amount of different languages available and the pronunciation and pronunciation speed can be changed by editing the text you send to the engine. While not the most flexible, the PICO TTS engine is clear, free, and can be run on the machine natively without an internet connection, which covers the majority of the requirements necessary for our system.

3.3 Interfaces

This sections discusses how the components are connected in the system block diagram and the communication between them. Also provided is an overview of the data that will go between the two components and what that data will be used for in the context of the entire system as a finished product.

3.3.1 To Radio

These connections are ones that go from the interface board to the radio.

3.3.1.1 TX Audio

The transmit audio will be an analog output coming from the 3.5mm jack of the Raspberry Pi 3. This signal will be run through the interface module. This audio will be processed so it can be transmitted by the KX-170B Radio without clipping.

3.3.1.2 PTT

This PTT block will put the KX-170B radio into transmit mode prior to audio being communicated. The purpose of this signal is to simulate the action that is pushing the mic button to talk over the radio. In this circuit there will be a testing switch that will allow for input simulation in system testing.

3.3.2 From Radio

Like the signals being used to be able to transmit through the KX-170B Radio, there is also a need to analyze signals coming from it. These signals will go through the interface board and allow for the Raspberry Pi 3 to analyze what is being needed by the pilot at the other end, as well as allow the Raspberry Pi to receive the actual audio from the pilot.

3.3.2.1 RX Audio

As previously mentioned the Raspberry Pi will need to be able to receive the audio being transmitted by the pilot. This RX audio signal will be picked up from the top of the volume potentiometer and run through the interface board. This is to prevent the volume setting on the actual KX-170B radio to effect the RX audio signal being transmitted to the interface board. To interface all analog signals with the Raspberry Pi 3's the signals need to be converted from analog to digital. The important detail on this conversion is the Raspberry Pi 3's SPI BUS requirement to have the output signal's logical 1 to have a 3.3V maximum input to be able to properly communicate with the Raspberry Pi 3. This means that some degree of voltage adjustment will have to be done to the output for a high resolution, but relatively low voltage analog to digital conversion.

3.3.2.2 Carrier Detect

The Carrier Detect in our system was identified from the main radio. The carrier detect levels were obtained by connecting the radio, at the squelch circuit output, to the oscilloscope and examining the output voltage when there is a RX signal detect present and when there is RX signal detect present.

Carrier Detected	7.83 Volts
No Carrier Detected	4.38 Volts

3.3.2.3 Automatic Gain Control

The AGC will be measured from the radio during a transmit radio check. A conversion table will be used to tell the microprocessor what the power level is during the transmission.

3.3.3 From Microcomputer

For output the Raspberry Pi 3's interface is more forgiving. The manipulations of the signals are more for convenience and practicality rather than must haves or else the system will be rendered useless. The two signals that the Raspberry Pi 3 will output to the interface are the two key communication signals in half-duplex communication. The PTT and the TX Audio signals. These are then received by the KX-170B Radio and broadcasted to the user on the other end of the communication channel.

3.3.3.1 PTT

The PTT line will be activated from one of the Raspberry Pi 3's GPIO pins to a circuit that will pull the PTT line of the KX-170B Radio to ground. This will engage the transmit mode on the KX-170B radio as indicated by the Raspberry PI 3's MCU.

3.3.3.2 TX Audio

The transmit audio by the Raspberry Pi 3 will be through the 3.5mm jack already built in the system and pass through the interface board. This audio signal will contain what the pilot will hear on the line. The Raspberry Pi 3 will transmit the received audio from the RX audio signal as well as a message about the RX audio signal's strength coming into the system, the wind information, and wind direction. This audio will be pushed through some signal conditioning and compression to result in a clear signal for the radio to transmit to the user.

Audio Limiting

There are two kinds of AC signal compression that were considered for this application. There is the limiting and clipping. The consideration for clipping stems from the idea that if a voltage were high enough it could end up damaging the radio beyond repairs. Voltage clipping is done to ensure that an AC voltage range does not exceed a pre-set maximum, and if desired minimum as well. This is mostly done using diodes which activate once the threshold voltage is reached and then redirect all excess voltage to ground or somewhere else in the circuit. In the end what happens is the signal is very distorted but it stays within the preset range regardless of the voltage pushed it, as long as it's nothing excessive. Typical clipped signals look like Figure 3.5.

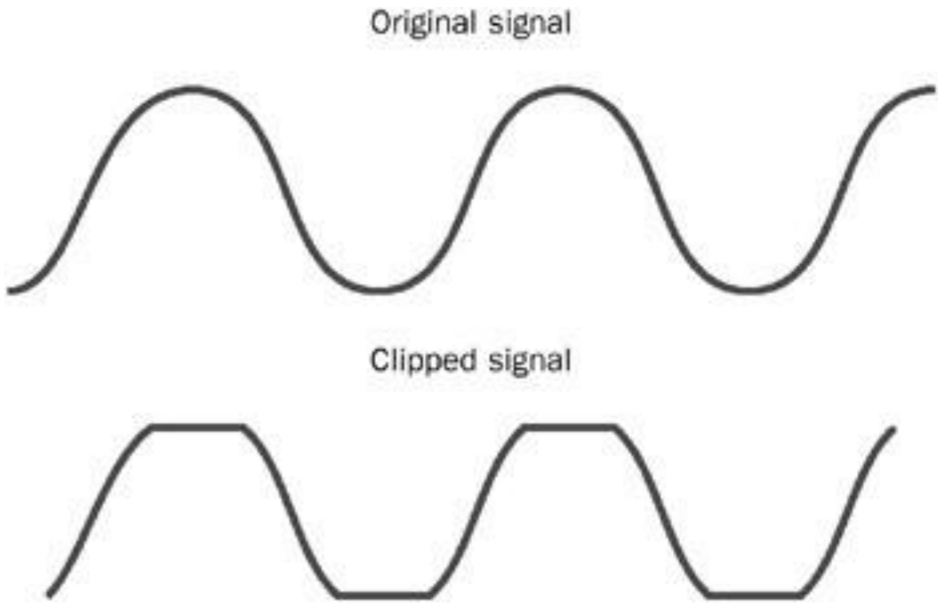


Figure 3.4 Original vs. Clipped Signal

As it can be clearly seen the original signal has been distorted losing some of the fidelity but if a system were to require a very specific maximum input this would ensure that maximum is not reached.

On the other hand, there is limiting, soft clipping, which is meets the system requirements in a more suiting manner. This is because soft limiting is similar to a clipper, but it reduces signal amplitude instead of re-routing it. Keeping signal fidelity, but removing the voltage ‘loudness’ from it. This is done by using an active circuit with an operational amplifier element. Using something like a FET and diodes to create variable gain that is dictated by the amplitude of the input. This created a relationship between the input signal and the gain of the system. Once the input signal reaches a certain threshold the gain, along with some voltage offset input will lower the amplitude of the system to keep the signal within the desired range. Figure 3.6 demonstrated how voltage limiter will have it output look once input surpassed the desired threshold compared to a hard limiter and the original input signal.

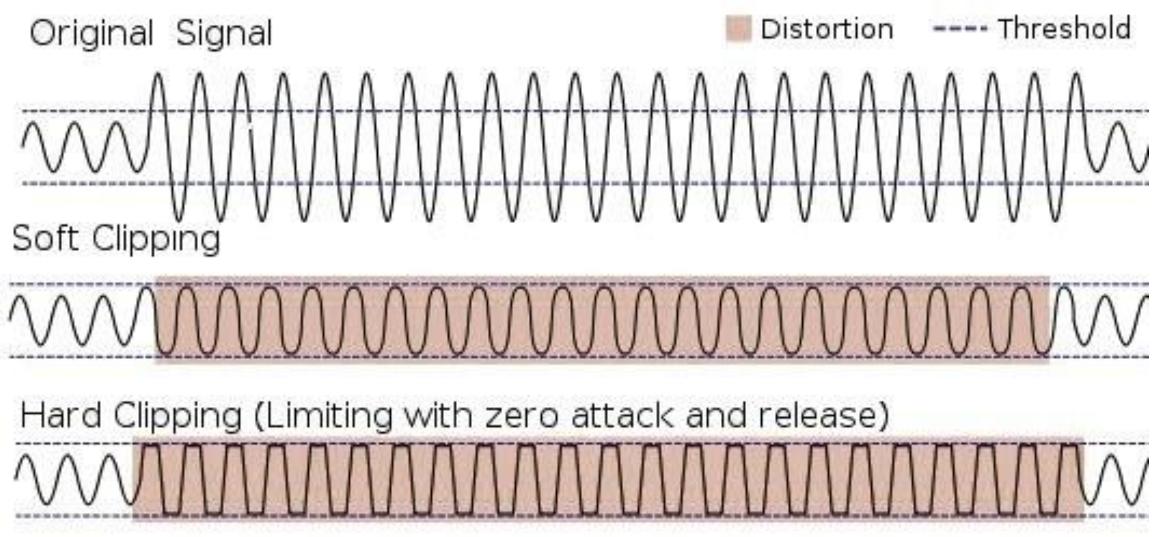


Figure 3.5 Hard Limiter vs. Soft Limiter Output

As demonstrated by the figure soft limiting keeps better signal fidelity which for the purpose of playback better suits the unmanned field base operator being designed. This is from the much slower attack and release times from the soft limiter. This gives the signals the soft edges instead of the hard cutoff at the threshold voltage.

3.3.4 To Microcomputer

These connections will go from the interface board to the microcomputer and will allow the microcomputer to analyze what the user is asking for. Most of these inputs come from the SPI bus that can be read and used by the microcomputer.

3.3.4.1 SPI Bus

The serial peripheral interface bus is a synchronous serial communication interface often used for short distance communication. In our system, the AGC voltage. RX audio, and wind direction are sent to the microcontroller using the SPI bus. The bus can operate with a single master device and with one or more slave devices. This means that multiple inputs can be connected to the SPI and they will be separated and transmit independently. The picture below shows how multiple inputs can be connected to the SPI bus at once.

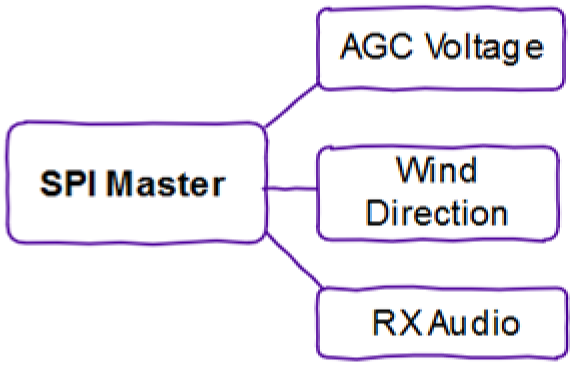


Figure 3.6 SPI Block Diagram

<u>AGC to SPI Bus</u>

After receiving the AGC voltage, conditioning it through the TL084 Op-Amp, and converting to a digital signal through the ADC, the AGC is sent to microprocessor through the SPI bus. The AGC voltage is one (1) of three (3) signals sent via SPI.

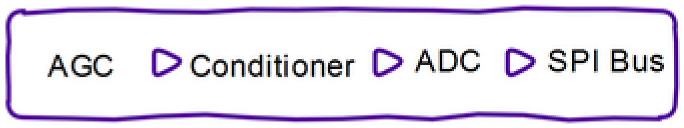


Figure 3.7 AGC Data Flow

<u>Wind Direction to SPI Bus</u>

The process that the wind direction follows is similar to the AGC voltage. After receiving the wind direction, conditioning it through the TL084 op-amp, and converting it to a digital signal through the ADC, the wind direction is sent to the microprocessor through the SPI bus. This is the second signal of three (3) sent to the raspberry pi via the SPI bus.

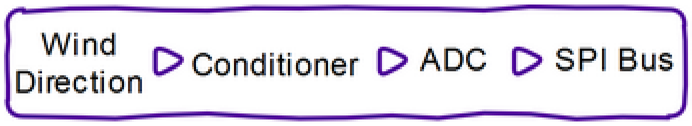


Figure 3.8 Wind Direction Data Flow

3.3.4.2 Carrier Detect

After the conditioning of the carrier detect (when present), the logical output of 3.3V continues to the microcontroller. The microcontroller receives this at one of the its general purpose input/output (GPIO) pins. It is important to note that the comparator output is 3.3V, which is the only input the Raspberry Pi can take. This continues to elaborate on why is important to have a logical output from the comparator that indicates the presence of a carrier detect regardless of its amplitude. In this case, 5V to 8V.

3.3.5 From Anemometer

These connections are coming into the interface board from the anemometer to be directly routed to the microcomputer.

3.3.5.1 Wind Speed

The anemometer provides us with the wind speed, after condition of this signal, the output of the PNP transistor continues to the microcontroller. This output, if wind is present, will be 3.3V. This value will be connected to a second GPIO pin on the raspberry pi. The important notes when measuring this value are the amount of times and how often this value is sent to the microcontroller. This will determine the value in miles per hour (MPH).

3.3.5.2 Wind Direction

The second important value that the anemometer provides us with is the direction of the wind. The anemometer uses a potentiometer to determine direction. The potentiometer can take up to 5V with a current limit of 20mA.

3.4 Power

It’s important to record the power consumption of the different devices used in the system. This provides us with information necessary to choose the right power supply. It’s important to power the devices with the correct needed voltage and current for best functionality and stability.

3.4.1 Voltage Source

The power source used for the AirBud is a 13.8V 5 Amp power supply

3.4.2 DC to DC Converters

There are several DC to DC voltage converters in our system. These are used to step down the 13.8V source to the desired outputs. As seen in table XY, the components require different voltages; 3.3V, 5V, 12V.

3.4.2.1 3.3V Regulators

The 3.3V source is used for 2 different things. The first one, as previously mentioned, is for the carrier detect. This 3.3V regulator provides the comparator with the necessary output when the comparator outputs a logical 1. When there is a carrier detect present, the output will be 3.3V; otherwise, 0V.

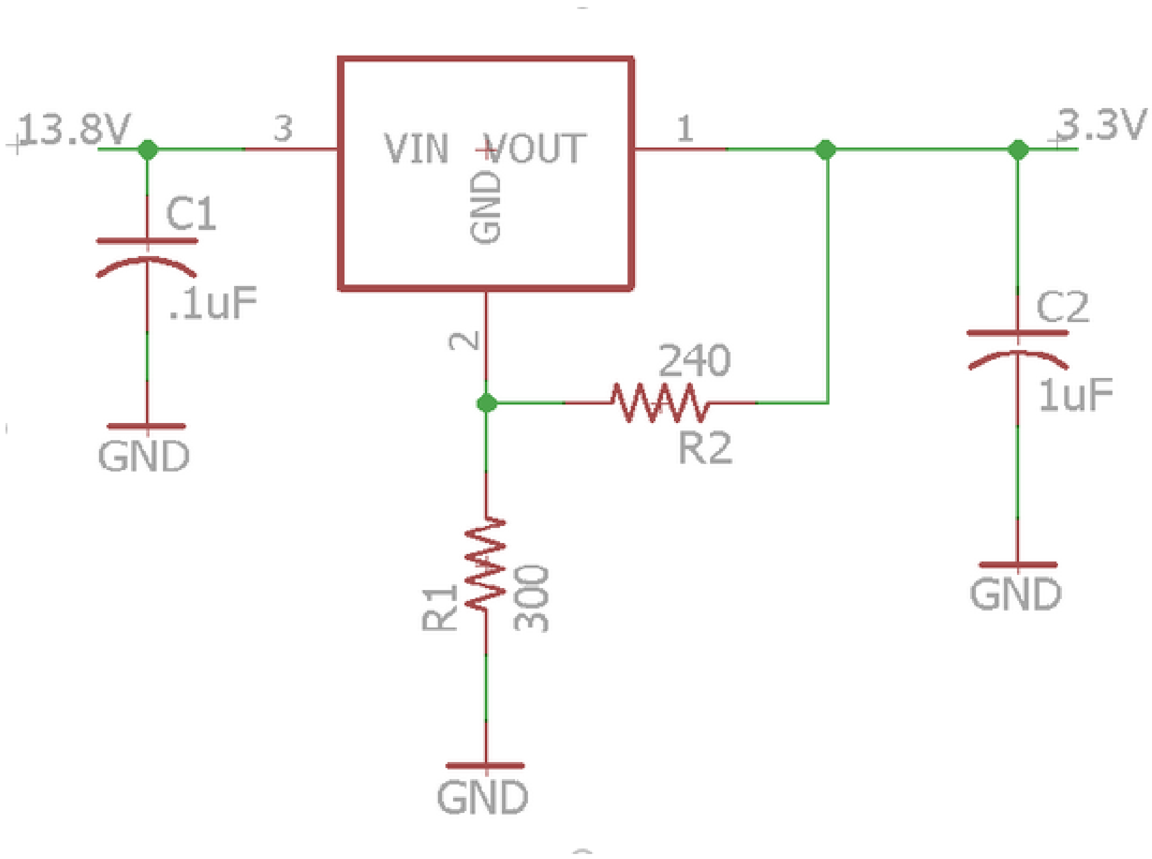


Figure 3.9 3.3V Regulator

3.4.2.2 5V Regulators

A second voltage regulator used in our system is for 5V. This 5V are used in the anemometer when measuring the wind direction and also to power the analog to digital converter MCP3008. The ADC has the option of also being powered with 3.3V but at 5V we are able to obtain a higher resolution when converting. The regulator has connected at its input a XY pF decoupling capacitor. This capacitor adds fast charge storage, commonly used when connected to a power supply which are relatively slow. The second pin of the regulator is connected directly to ground and then the third pin, output is connected to a XY capacitor that is also connected to ground. This second capacitor serves the same purpose of adding fast charge storage. Below is the schematic for the 5V regulator setup.

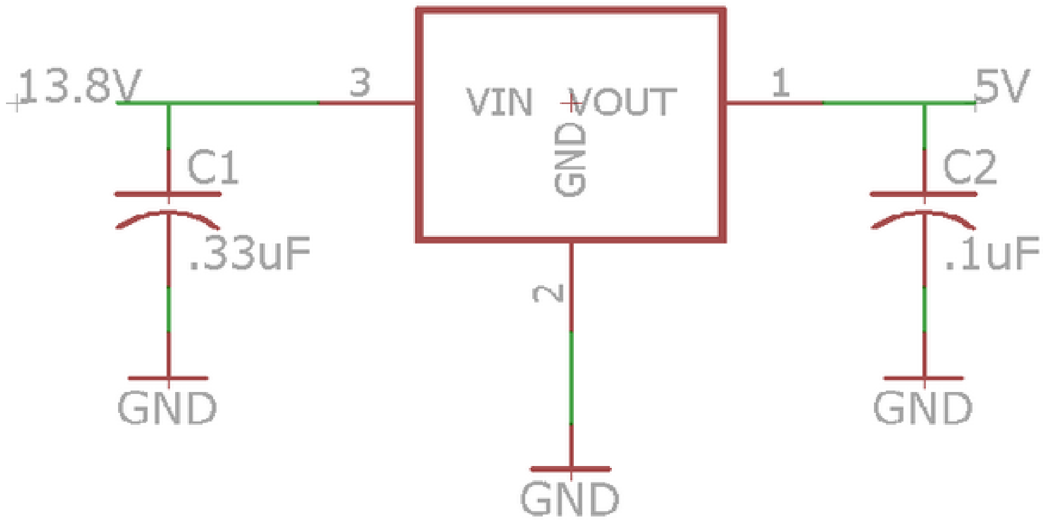


Figure 3.10 5V Regulator

3.4.2.3 12V Regulators

The third voltage regulator used in our system is for 12V. We obtain 13.8V from the source and convert it to 12V. This 12V are used to power the op-amps. The regulator has connected at its input a XY pF decoupling capacitor. This capacitor adds fast charge storage, commonly used when connected to a power supply which are relatively slow. The second pin of the regulator is connected directly to ground and then the third pin, output is connected to a XY capacitor that is also connected to ground. This second capacitor serves the same purpose of adding fast charge storage. Below is the schematic for the 12V regulator setup.

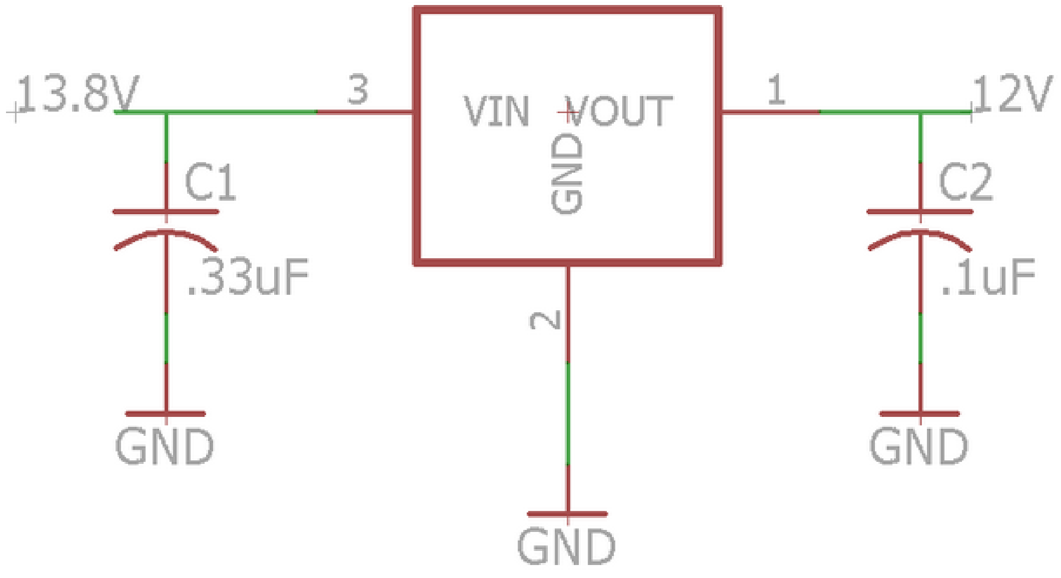


Figure 3.11 12V Regulator

3.4.2.4 -12V Voltage Inverter

The need for a 12V negative power supply comes from the desired soft limiter being used for the interface board. This soft limiter will be applied for the TX Audio. As Figure 4.2.5 C shows there is need for a -12V DC power supply. The desired part for this implementation is the TPS63700 DC-DC Inverter.

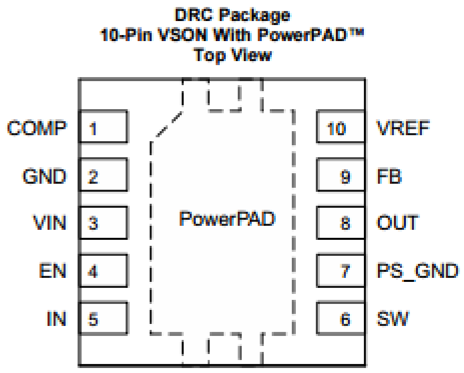


Figure 3.12 TPS 63700 Pin Layout

This is a TI component who's input and output voltage ranges match the system requirements. Though it could be supplied with a 3.3V input from one of the regulators it is preferable to supply it through the 5V regulator. This is because it provides the highest efficiency in the output signal, and supply an output of -12V for the interface board.

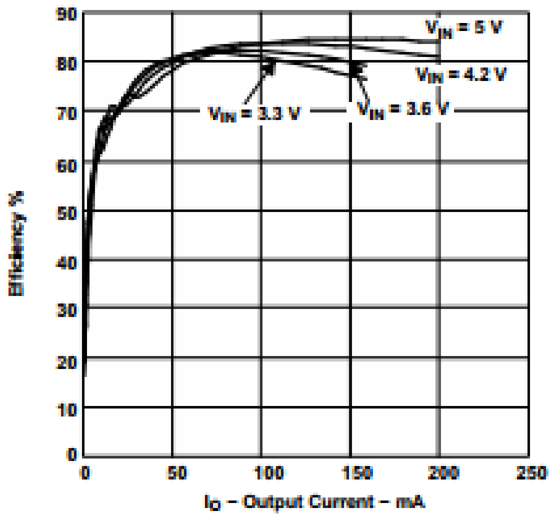


Figure 4. Efficiency vs Output Current, VOUT –12 V

Figure 3.13 Efficiency at 12V output

The layout for the TPS63700 DC-to-DC inverter requires different components depending on the desired voltage output. Since the interface board requires the -12V output the following schematic is appropriate:

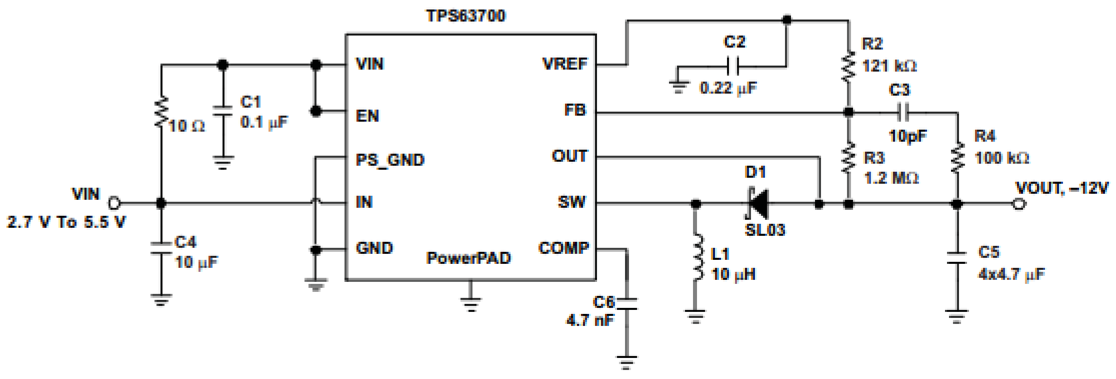


Figure 15. Circuit for –12-V Output

Figure 3.14 -12V Supply TI Reference Schematic

4 Interface Board Design

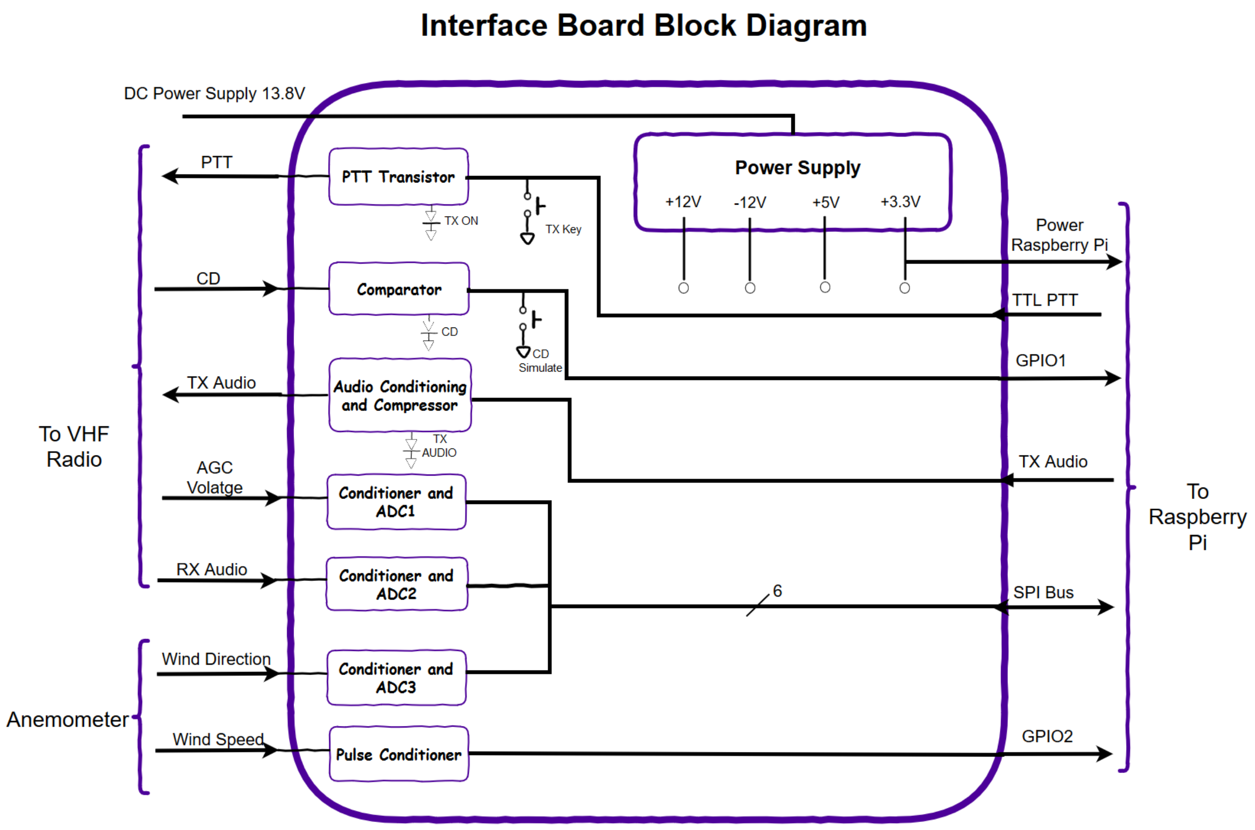


Figure 4.1 Interface Board Block Diagram

The interface board will interpret all of the incoming and outgoing signals between the radio and the Microprocessor. This will be handling the TX and RX signal conditioning, conversion and amplification between the two systems. This will also push the PTT signal into the radio for whenever a transmission is going to be sent out to the pilot requesting information. Inputs will be received from directly tapping into the radio at specific solder points or through the back pins of the KX-170B VHF Aircraft radio. In this the communication between the UNICOM programmable HUB, the Raspberry Pi 3, and the broadcasting hardware, the KX-170B VHF Aircraft radio

4.1 Power

Due to the Florida heat the operating temperature of components will be assumed to be at 30ºC for power consumption calculations unless otherwise stated.

Component	Part Number	Power Consumption at 30ºC	Quantity
Operational Amplifier	TL084	490 mW	1
Operational Amplifier	LM234	800 mW	3
PNP Transistor	2N4403	600 mW	2
NPN Transistor	2N4401	625 mW	1
FET	2N3820	360 mW	1
NPN Transistor	BC546	625 mW	1

4.2 SPI Bus

After the AGC, RX Audio, and Wind direction signals have been converted to digital signals this SPI Bus will be used to deliver information to the Raspberry Pi 3. This bus will carry information in the form of a 3.3V maximum voltage digital signal.

4.3 PTT Circuit

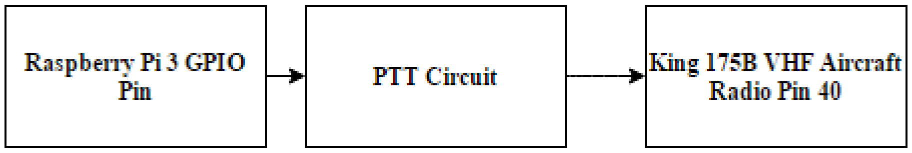


Figure 4.2 PTT Block Diagram

In order to communicate to the Raspberry Pi 3's intent to transmit a signal needs to be pushed so the KX-170B Radio in order to get it in a 'Ready to Transmit' state. This signal is going to be generated by the Raspberry Pi 3's GPIO pin and a DC power source for system testing. This will require two inputs: one for system use and one for system trouble shooting. The input from the Raspberry Pi 3's GPIO pin will be for practical use, thus the DC voltage source will be used for testing. During testing the GPIO pin will act as a ground and part of the current will be sent through there and the rest will be sent to the PTT input of the radio. This will allow the user to check if the circuit is bad or if there has been a programming error in the Raspberry Pi 3 system. The intent of this circuit is to simulate the PTT signal generated by the microphone interface in the radio. The idea is to act grounded when not transmitting and to input a current when ready to transmit in order to open the mic channel and set the KX-170B in a ready to transmit mode.

4.3.1 PTT Circuit Design

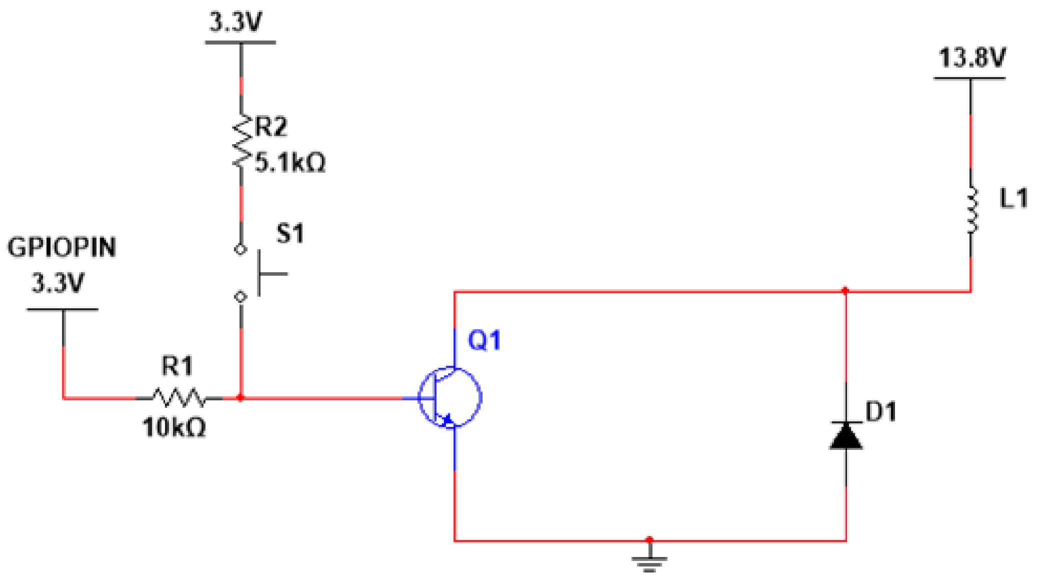


Figure 4.3 PTT Circuit

The Push-To-Talk (PTT) circuit is going to be responsible for setting the KX-170B radio into transmit mode. This is done by using the GPIO pin in the Raspberry Pi 3's pins as a 3.3V source. This voltage being pushed through the NPN transistor, Q1, pulls the PTT relay day to ground. The action of pulling the relay to ground results in the collapse of the magnetic field around the inductor. This will send a large voltage back from the PTT relay to the Q1 transistor. This is where the reverse biased diode will re-route that voltage to ground, thus not burning the transistor. When it comes to testing the system switch, S1, will have the 3.3V source from the interface board act as the Raspberry Pi 3's GPIO input. This will simulate the act of readying for transmit on the KX-170B. Though the design shows the GPIOPIN power source as a 3.3V power source it must be noted that this is a pin from the Raspberry Pi 3's interface. This will act as a ground when being tested as the system will be inactive, or turned off, when being tested. The circuit takes full advantage of the Raspberry Pi 3's architecture to reduce the amount of components required to achieve the same function. When using the Raspberry Pi 3's GPIO as a ground its current limits is around 16mA maximum current before burning the microprocessor. Therefore, the current running from the interface board power supply is split using resistors R1 and R2 above. Thus ensuring that the Raspberry Pi 3 operates as a ground with proper current flow.

4.4 Carrier Detect

The consolidation between the Radio and Interface Board serves as the bridge to be able to condition the carrier detect and identify when there will be transmission. Since we only have two (2) levels for identification, a comparator is being used to compare and determine which level, that indicates transmission or no transmission, is being received.

4.4.1 Comparator Circuit

The comparator being used is the LM393 Dual Differential Comparator. The purpose of this device is to compare two (2) voltage values, and output a digital signal indicating which of the two is larger to the main control unit through a GPIO.

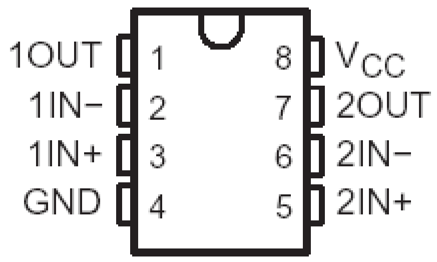


Figure 4.4 LM393

The differential comparator consists of a high gain differential amplifier. These devices are commonly used in systems that measure and digitize analog signals such as analog to digital converters, as well as relaxation oscillators.

In our application we compare the received signal, carrier detect present or carrier detect not present, with a reference voltage.

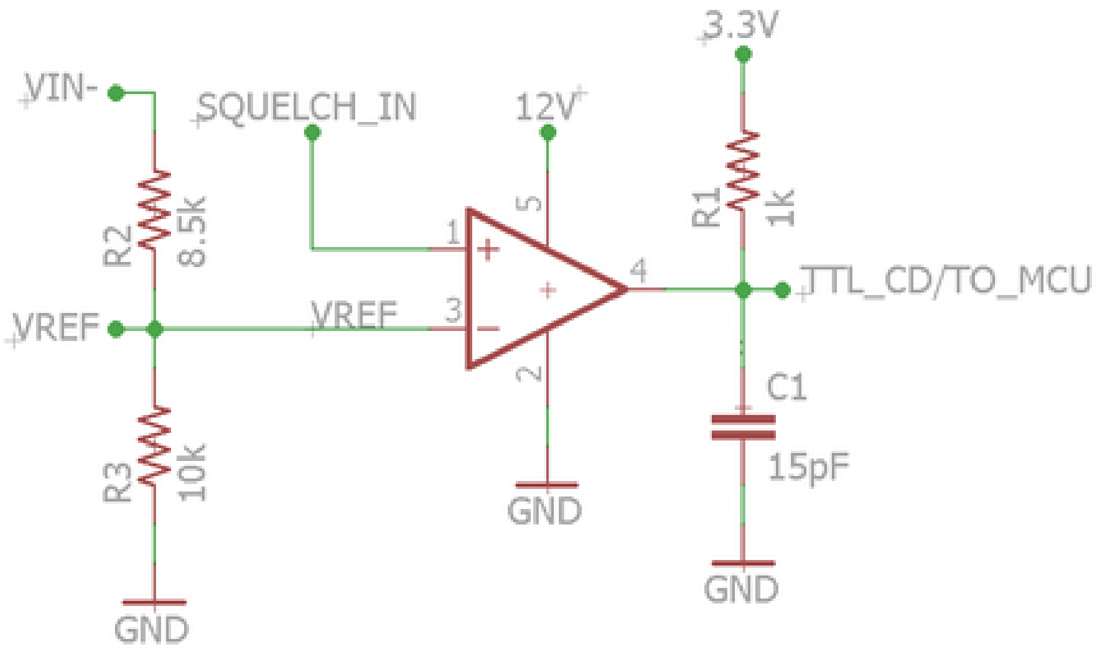


Figure 4.5 Comparator Circuit

The voltage measured for RX audio signal (CD) being present was 8V+; meaning, when compared to the reference voltage, 6.5V, the comparator will output a logical 1, allowing the 3.3V become the output to the next stage of the circuit. Next stage of the circuit being to a GPIO pin of the microcontroller. The voltage measured for RX audio signal (CD) not present was 5V+; meaning, when compared to the reference voltage, 6.5V, the comparator will output a logical 0, this output will not allow the 3.3V become the output to the GPIO pin. See image above.

$$V0 = \begin{cases} 0, & V+ < V- \\ 1, & V- \geq V- \end{cases}$$

The 6.5V for reference are achieved through a voltage divider circuit. The input (Vin-) is 12V which is then divided through both resistors of 10k ohms and 8.5k ohms. The reference is then then compared to the ground at the 10k ohms resistor. This will create a constant output of 6.5V since the 12V is being provided by a voltage regulator.

4.4.2 Operational Amplifiers

The use of operational amplifiers is necessary for the many active circuits being used in this interface board. Since the interface board will be handling two analog signals, TX and RX audio. In order to properly condition those signals into ideal signals for the Raspberry Pi 3 they must go through some active circuits. Initially the interface board was planned to not include any negative power supplies and the TL324 would be the ideal operational amplifier to handle the signal processing. Though as the design was further looked into and more complex circuits were deemed fit for the practicality of this system a need for a negative power supply was required. This meant that the LM324 could be replaced for a TL084 since dual power supplies would be possible. This decision to switch would have required to a redesign of all circuits. What was decided upon was to simply use both circuits. Since the LM234 would simply use the positive 12V supply and the TL084 would use both the positive 12V and the negative 12V supplies.

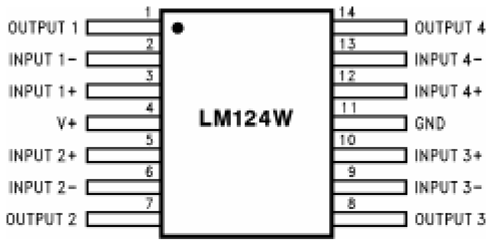


Figure 4.6 LM124W Pinout

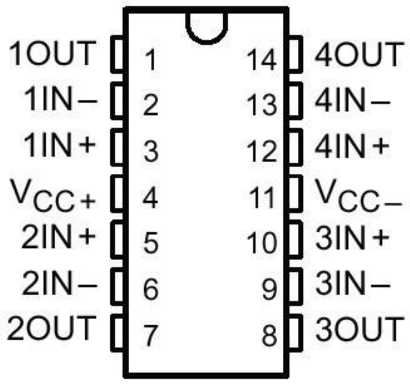


Figure 4.7 TL084 Pinout

It can be noted that both the TL084 and the LM324 have the same pin layout with the only difference being at the negative input, pin 11. Where the TL084 takes in a negative supply the LM234 will have that pin to ground. This makes them ideal for breadboarding. Dues to ease of access breadboarding tests will be done will the TL084 operational amplifier, since their function is interchangeable and the circuit would remain unchanged once they are swapped out.

4.5 TX Audio Circuit

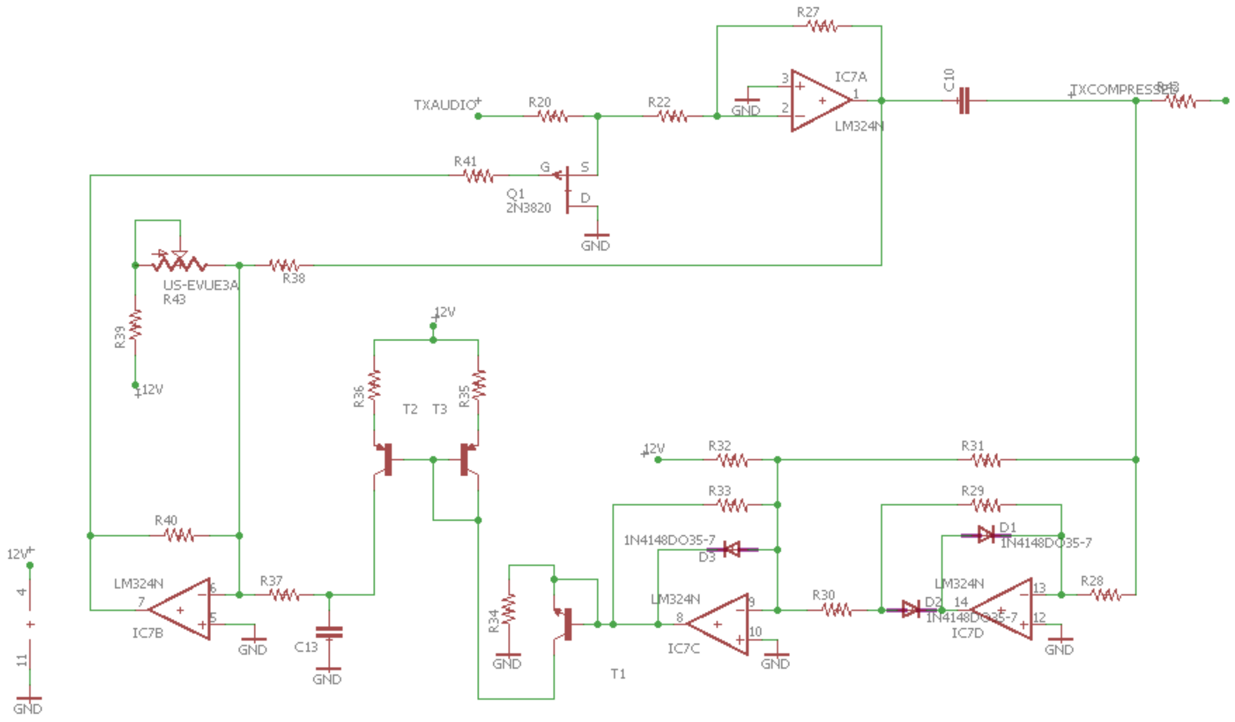


Figure 4.8 TX Audio Schematic Section 1

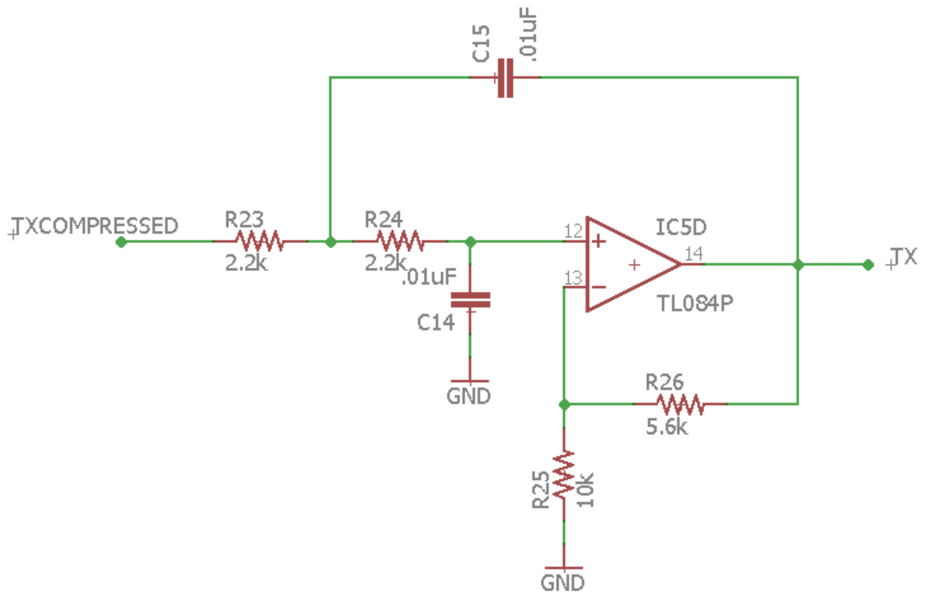


Figure 4.9 TX Audio Schematic Section 2

The KX-170B VHF Radio usually is used of a carbon microphone. The purpose of this circuit is to inject the transmit audio coming into the radio from the Raspberry Pi 3's 3.5mm audio jack. The transmitted audio will be required to be conditioned and compressed. The conditioning and compression of the transmission audio signal is to prevent over modulation when being transmitted by the KX-170B Radio.

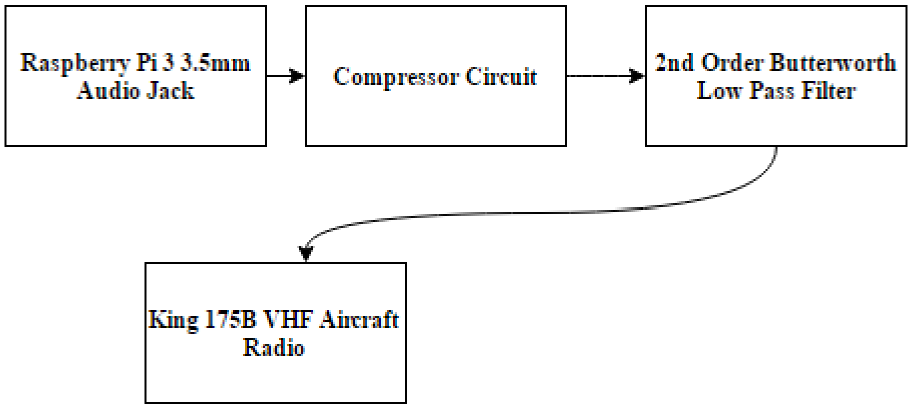

Raspberry Pi 3 3.5mm Audio Jack
Compressor Circuit
2nd Order Butterworth Low Pass Filter
King 175B VHF Aircraft Radio

Figure 4.10 TX Audio Circuit Block Diagram

4.5.1 Audio Compression Circuit

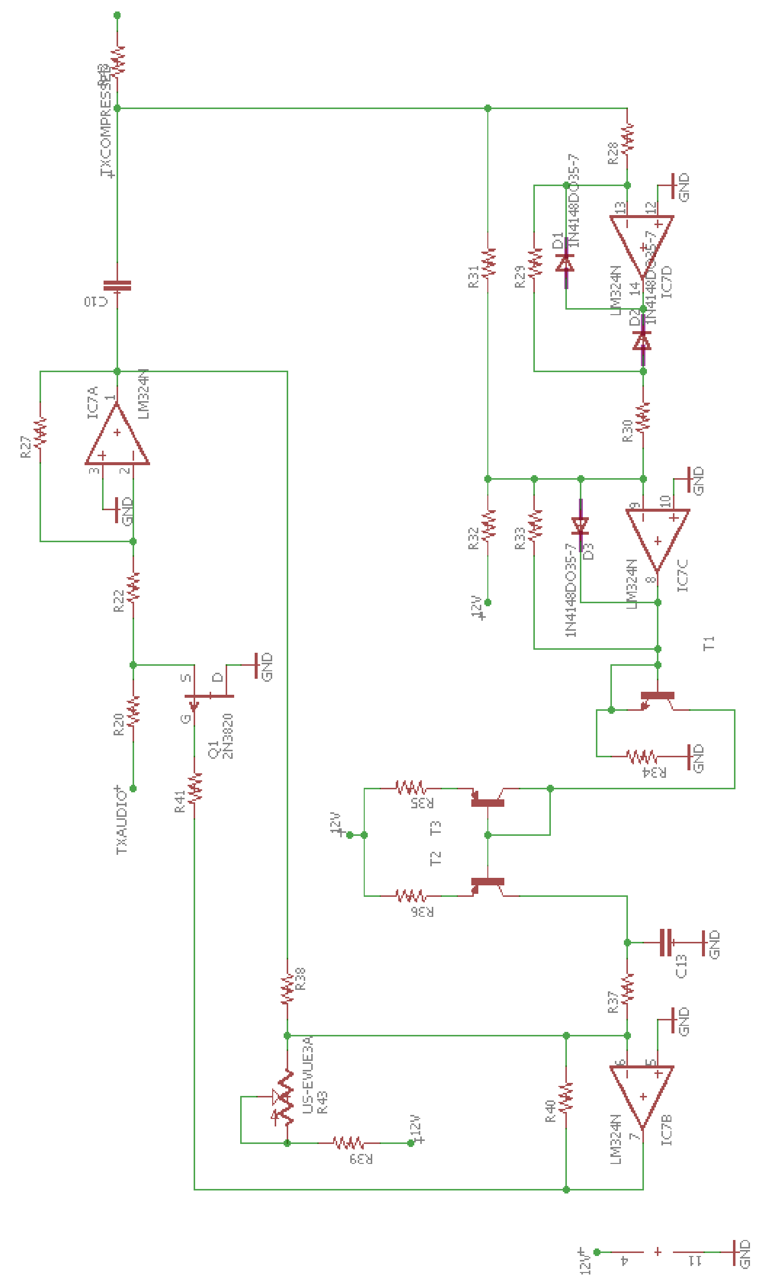


Figure 4.11 TX Audio Compressor Schematic

The use of audio compression is to normalize the varying amplitude of the outgoing transmission signal, through the interface board. This will protect the transmitted audio signals from over modulation and distortion, but keep some degree of fidelity when transmitting. The reason being, when the Raspberry Pi 3 does the RX audio retransmission to the pilot, the pilot will then hear his transmitted audio the same way the Raspberry Pi 3 received it. To do this the circuit will use a combination of diodes, one p-channel JFET, two NPN transistors, one PNP transistor and four operational amplifiers. The TX audio signal will be pushed through the operational amplifier with an almost unit gain, due to resistor errors. This TXCOMPRESSED output will then go through a full-wave detector circuit where it will activate upon all large increases in amplitude of the circuit. When the TXCOMPRESSED output is large enough and trips the full-wave detector that in turn will activate the collector on transistor T1. This then gets mirrored by the T2 and T3 circuit which then pushes it onto the negative input of IC7B, the fourth operation amplifier in the circuit, which pulls a negative voltage through the JFET, and thus lowering the input amplitude to the circuit.

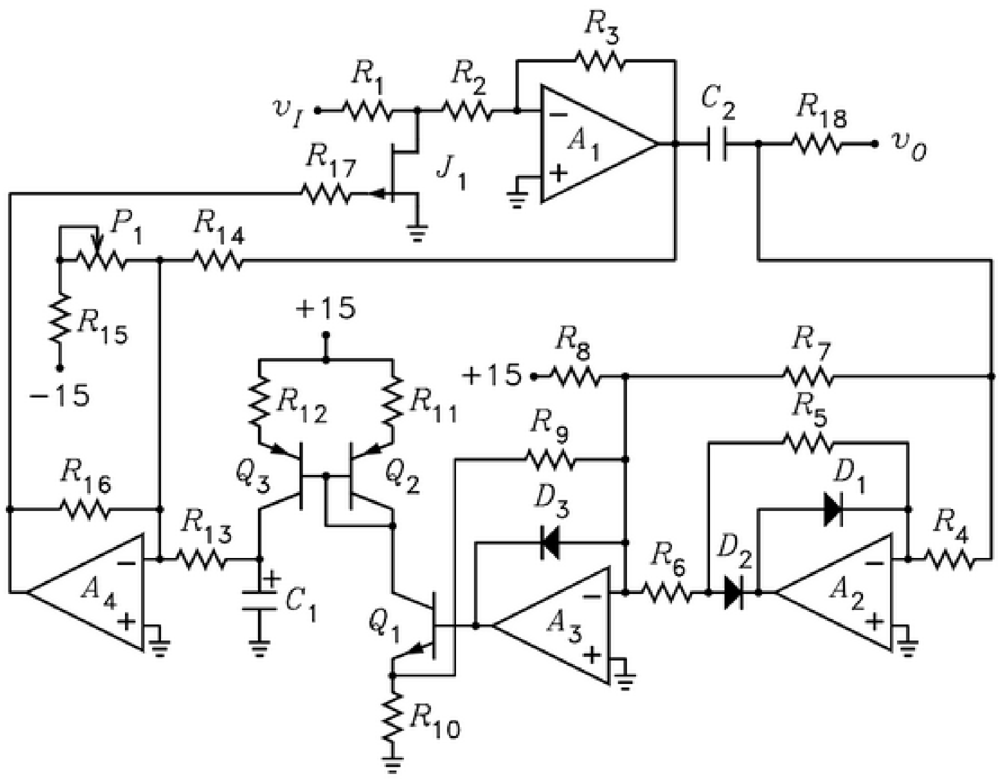


Figure 4.12 FET Audio Peak Limiter (Georgia Tech Permission Pending)

The circuit design being used for the TX Audio Compressor is a peak voltage limiter designed by Marshall Leach, a professor at the Georgia Institute of Technology. The reason this circuit is going to be used to audio compression is that the attack time for the limiter is fast enough for audio transmission without any noticeable delay. The parameters for the voltage activation for the full-wave detector is determined by the resistors R1 and R2 in tandem with the pinch-off voltage of the FET J1. With the FET pinch off voltage being denoted as VP and its saturation current as IDSS the formula for determining appropriate values for R1 and R3 are (CITATION):

$$R_1 \| R_2 = \frac{9\,|V_P|}{2I_{DSS}}$$

$$R_2 = \frac{|V_P|}{40V_L} R_3 = \frac{|V_P|}{40} R_3$$

$$R_3 = k\,(R_1 + R_2)$$

Where the parameters of the pinch-off voltage and saturation current are depended on the individual FET and should not be considered universal. This is because different FET with different values for the latter will result in different optimal values for R1, R2 and R3 respectively. The circuit activation is intended to be around one tenth of the pinch-off voltage. The 2N5450 p-channel JFET is around 3V making the 300mV TX Audio optimal for conditioning since anything above 300mV would get reduced accordingly. Ideally for this kind of design the values for R1, R2 and R3 would be chosen such that the limiter slope would be around 20dbm. Though it is not large for audio conditioning and due to the range of voltage the TX Audio will be ranging in that slope is more than enough to limit the incoming audio properly. Adjusting for larger values would require larger resistors which in turn would increase the noise due to thermal noise from the larger resistors. To ensure the proper offset is fed back into the input through the FET resistors R16 and R14 must meet the following condition:

$$\frac{R_3}{R_2} \times \frac{R_{16}}{R_{14}} = \frac{1}{2}$$

The gate voltage coming into the FET must be around the same as the pinch off voltage, thus values for R15 and P1 must be chosen such that it can range not too far off from 3V (an average pinch off voltage for the 2N5460). Having the negative voltage supply as reference for that voltage divider:

$$V_L = V^+ \frac{R_7}{R_8}$$

Where V+ is supplied by the 12V power supply for our implementation thus it must also be reflected in the other component values for the circuit, but it does not affect the efficiency as the circuit will only require voltage ranges no larger than 1V. Lastly the attack time on the limiter has its properties defined by a combination of resistors R10, R9, R13, R1 and R2, capacitor C2 and the properties of the FET.

$$R_9 \| R_{10} = \frac{2R_{13} I_{DSS} V_L R_1 \| R_2}{V_P^2 \left(\tau_r / \tau_a - 1\right)} = \frac{2R_{13} I_{DSS} R_1 \| R_2}{V_P^2 \left(\tau_r / \tau_a - 1\right)}$$

4.5.2 Audio Conditioning Circuit

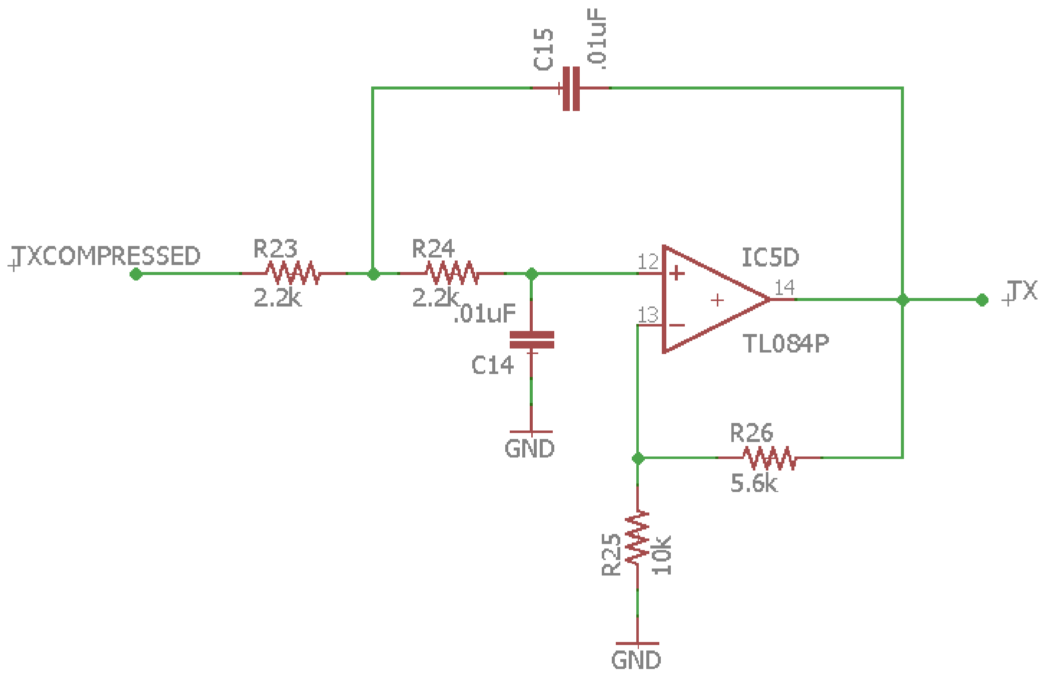


Figure 4.13 TX Audio Low Pass Filter Schematic

The TX Compressed Audio will be run through a 2 stage Butterworth low pass filter. This active filter uses its capacitors, C1 and C2, as well as its' resistors, R1, R2, R3, and R4, in order to filter any signals with a frequency higher than 7.2kHz. This is done as a final measure to ensure fidelity in the signal being transmitted, thus blocking out any high frequency noise that may have leaked into the signal prior to transmission

4.6 RX Audio Circuit

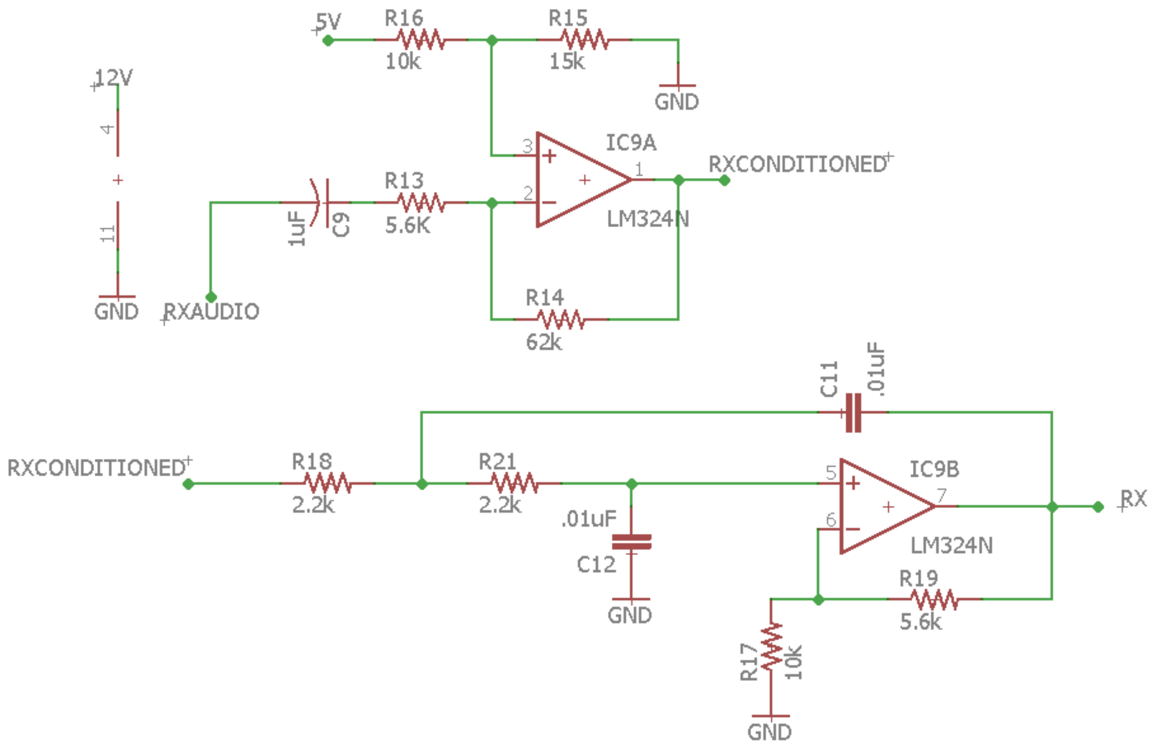
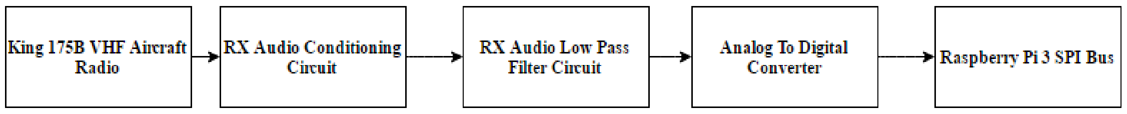


Figure 4.14 RX Audio Schematic



King 175B VHF Aircraft Radio
RX Audio Conditioning Circuit
RX Audio Low Pass Filter Circuit
Analog To Digital Converter
Raspberry Pi 3 SPI Bus

Figure 4.15 RX Audio Circuit Block Diagram

The received audio signal is what is going to allow the Raspberry Pi 3 to later re transmit the signal in a later stage of the equipment check function. While the AGC will provide the Raspberry Pi 3 the received signal strength the RX audio will provide the voice recording that will be played back to the pilot during the equipment check. This will let the pilot know that his microphone is working and the signal strength the radio is receiving. The received audio signal will only need to be amplified enough so that the analog to digital converter can properly convert it into the bit stream that will pass through the SPI bus and into the Raspberry Pi 3. The received audio signal will be amplified to a maximum of 2.92V. The RX Audio signal will be offset by 3V in order to provide an even 2 range for both the positive and negative end of the audio signals. This is to avoid reaching the 1V input voltage to ensure accurate readings in the analog to digital converter. This will go through the analog to digital converter that will have a 5V high resolution digital output. This output will then be reduced to 3.3V through some voltage dividers. The Raspberry Pi 3's programming will then interpret that into the appropriate digital bit stream for play back and message synthetization during the transmission stage of communications.

4.6.1 Audio Conditioning Circuit

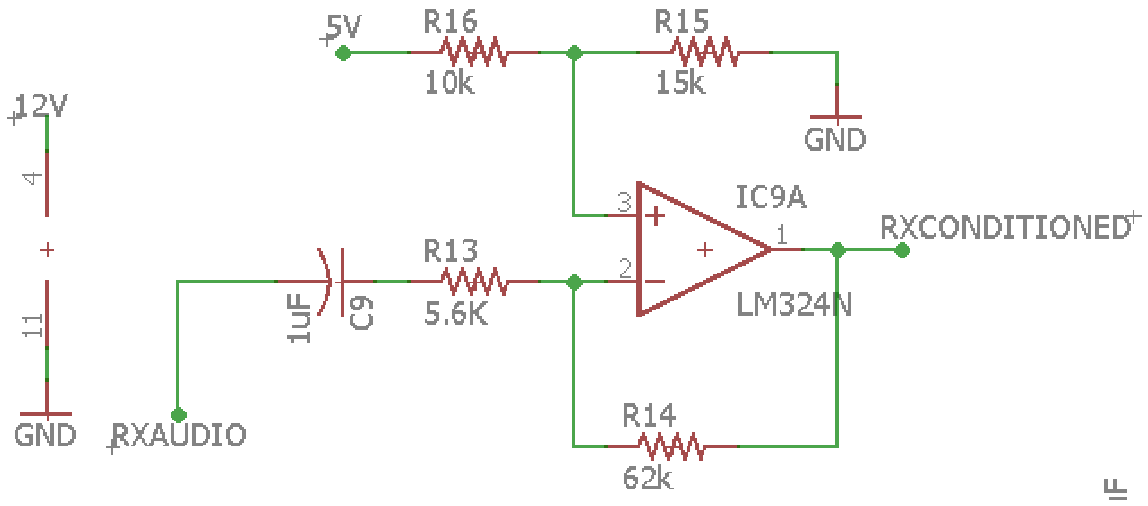


Figure 4.16 RX Audio Conditioning Schematic

The analog conditioning interface circuit will condition the RX audio such that it no longer has a ± voltage amplitude but will instead be a completely positive voltage AC signal. This signal will be amplified and filtered so no more noise is going through it before making into the analog to digital converter. This circuit has no gain as the final filter will provide the gain and dynamic range for the analog to digital converter. In Figure 4.2.6 B the polarized capacitor is there to remove all dc biasing from the received audio. This will provide the interface board with an unbiased, analog audio signal that will be conditioned by the circuit. Using the 5V power supply from the interface board a positive bias is given to the RX audio signal through the operational amplifier IC9A. Though because a 5V bias is too much for the analog to digital converter it is then divided using resistors R16 and R15. Using values of 10k and 15k respectively the resistors split the interface 5V input into a 3V reference voltage that will go into the negative op amp input. The signal is not only offset but also amplified by the operational amplifier. Though because the signal cannot exceed 5V once being pushed into the analog to digital converter it is only amplified by a factor of 11.

4.6.2 Low Pass Filter Circuit

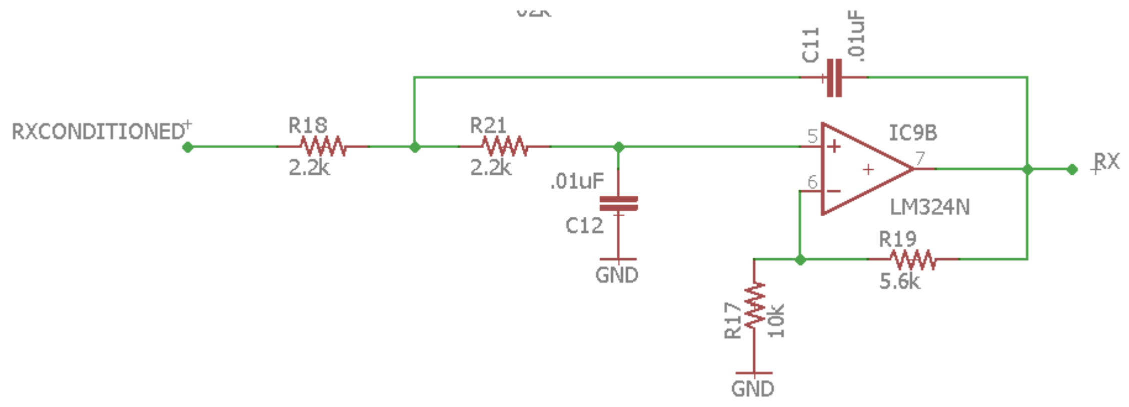


Figure 4.17 RX Audio Low Pass Filter Schematic

The 2 stage Butterworth low pass filter will be the second stage of the Rx Audio circuit. This will filter out any high frequency noise in the signal. Any frequency above 7kHz will be filtered at a rate of -40dB/dec. This will ensure that the audio going into the Analog to Digital converter is only the RX audio and no other noise that could be interfering with the signal. This circuit does have a considerable gain on it already, but since the filter has such a small gain it is considered negligible.

4.7 Automatic Gain Control Voltage

The AGC Buffer Amplifier in our system is used to maintain a constant amplitude level for the received signal (sound). Like the comparator, the bridge between the AGC and the interface board, allows us to condition the signal before using it.

4.7.1 Conditioner Circuit

As previously discussed, the input can have a wide range and it’s necessary to maintain the output constant. Filtration and amplification were achieved using an operation amplifier with multiple stages. The chosen operational amplifier was the TL084. The AGC buffer amplifier is composed of a low pass filter and a second stage amplifier that splits the gain of the first stage. The Op Amp will produce an output potential larger than the potential difference between its input terminals. The first stage, low pass filter, will pass signals with a frequency lower than a 7kHz, cutoff frequency, and attenuate signals with frequency higher than the cutoff. The signal will then fall under the second stage of the circuit, where its previous gain of half (.5) will be split, giving an output of unity gain (1), using a negative feedback loop to achieve this result. The input of the AGC will go to the negative input of the op amp. The constant output of the op amp will continue to next stage of the interface board. The AGC buffer design also takes into account the analog to digital converter ideal maximum and minimum values of 1V and 5V respectively. Thus the gain of the first stage and the voltage division of the second stage is designed to get as close as possible to those ranges, without hitting rail on the operational amplifiers not going outside of them. Then the second stage of the AGC will divide it such that when a strong signal is present the analog to digital converter gets close to 5V in input, but when the signal is at its weakest it still gets above 1V of input from the AGC circuit.

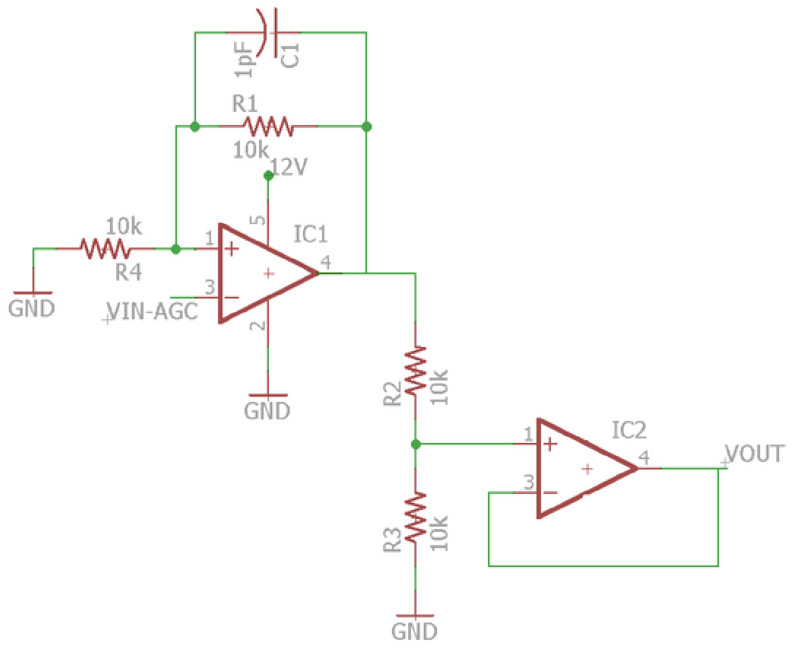
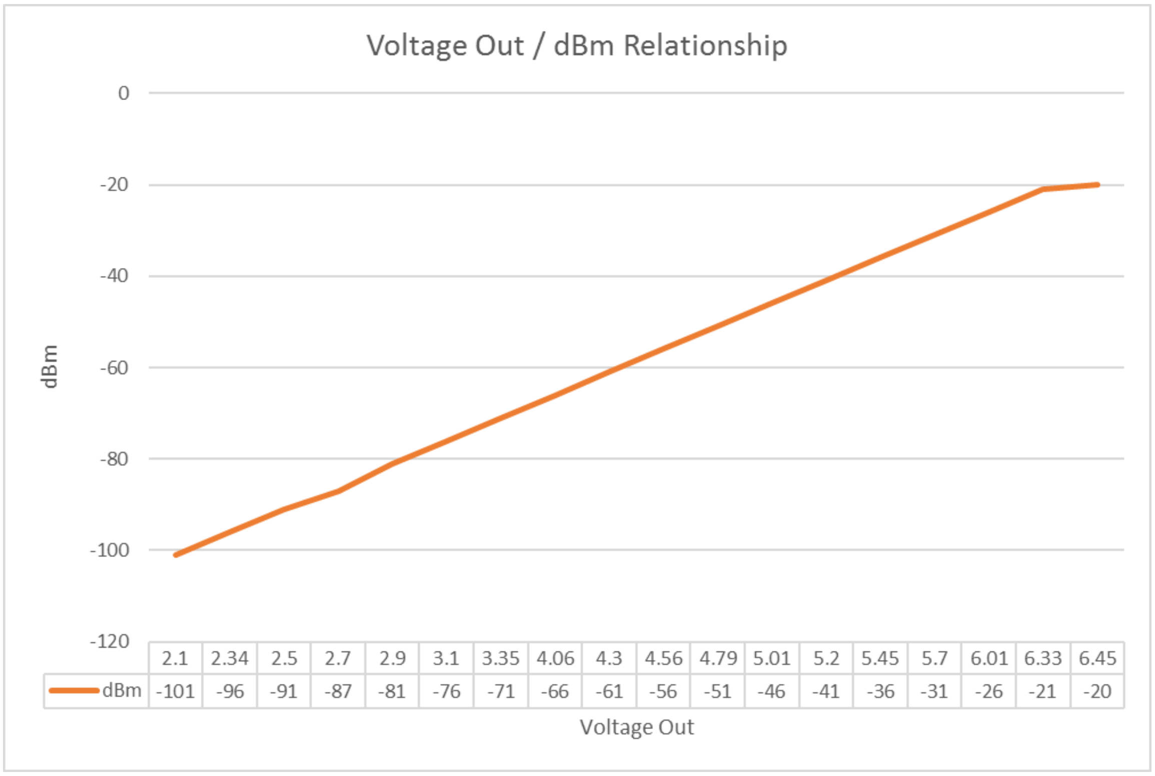


Figure 4.18 Interface Board AGC Schematic

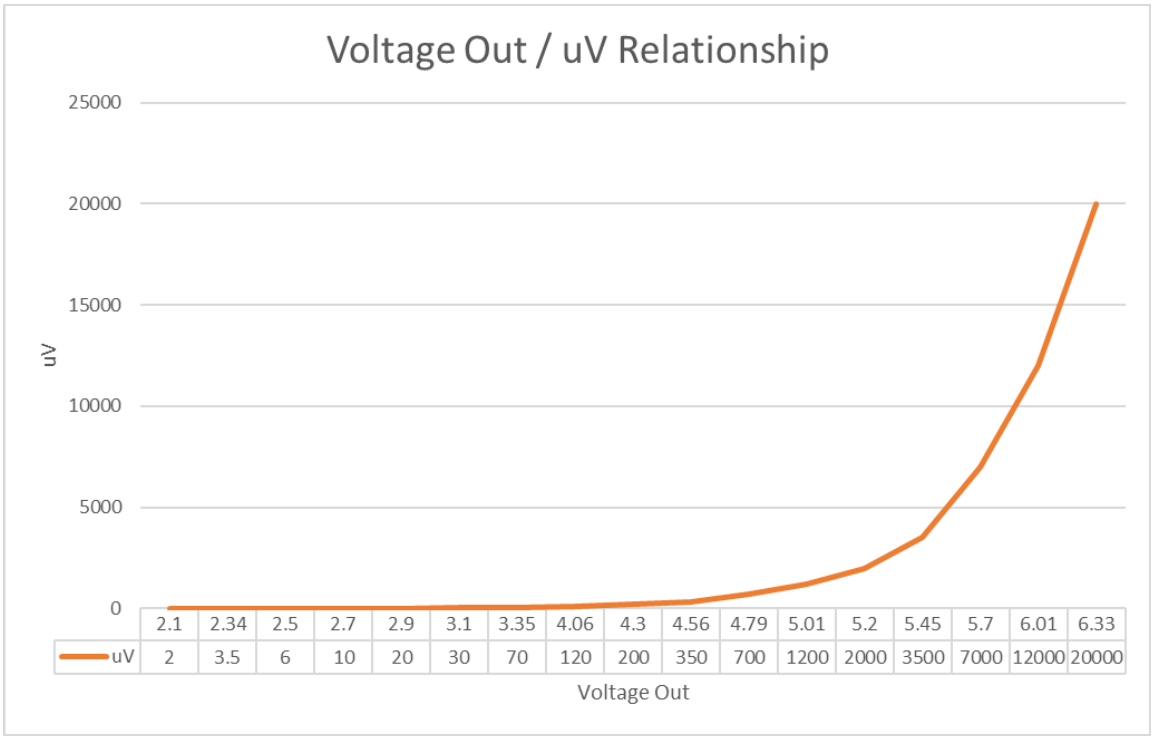
The range of inputs (measured) for the amplitude gain control from the radio are the following:

Voltage Out	dBm	uV
2.10	-101	2
2.34	-96	3.5
2.50	-91	6
2.70	-87	10
2.90	-81	20
3.10	-76	30
3.35	-71	70
4.06	-66	120
4.30	-61	200
4.56	-56	350
4.79	-51	700
5.01	-46	1200
5.20	-41	2000
5.45	-36	3500
5.70	-31	7000
6.01	-26	12000
6.33	-21	20000
6.45	-20	20000+



Voltage Out	2.1	2.34	2.5	2.7	2.9	3.1	3.35	4.06	4.3	4.56	4.79	5.01	5.2	5.45	5.7	6.01	6.33	6.45
dBm	-101	-96	-91	-87	-81	-76	-71	-66	-61	-56	-51	-46	-41	-36	-31	-26	-21	-20

Figure 4.19 Relationship between Voltage Out and dBm



Voltage Out	2.1	2.34	2.5	2.7	2.9	3.1	3.35	4.06	4.3	4.56	4.79	5.01	5.2	5.45	5.7	6.01	6.33
uV	2	3.5	6	10	20	30	70	120	200	350	700	1200	2000	3500	7000	12000	20000

Figure 4.20 Relationship between Voltage Out and uV

4.7.2 Analog to Digital Converter

After the AGC voltage is received and conditioned it goes to the analog to digital converter. The Analog to digital converter chosen was the MCP3008.

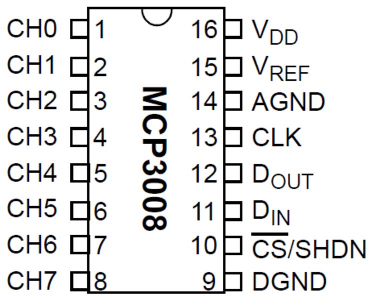


Figure 4.21 MCP3008

The purpose of the analog to digital converter (ADC) is to provide the microcontroller with a digital number that is proportional to the magnitude of the signal, voltage or current, sent from the AGC. The conversion of this signal involves some error parameter. The higher the number of bits, resolution, available on the ADC, the more precise the conversion can be. The MCP3008 allows a precision of 10 bits, this indicates the number of discrete values it can produce over the range of analog values. An ADC is defined by the bandwidth available, range of frequencies, and its signal to noise ratio.

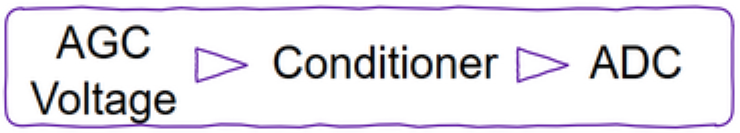


Figure 4.22 Data Flow

4.8 Wind Data

4.8.1 Wind Speed

The anemometer provides our system with 2 important values. The first one being the wind speed. The anemometer uses a reed switch that is opened and closed by a magnet that passes over the switch after rotating due to the wind. The wind speed is determined by counting the closures over a sample period of 2.25 seconds. This sample is then converted to miles per hours (MPH).

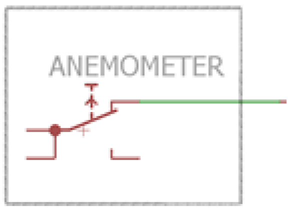


Figure 4.23 Anemometer Switch

The black and red wires are used to measure the wind speed. The red wire represents ground. Connected to the black wire the reed switch can be seen open.

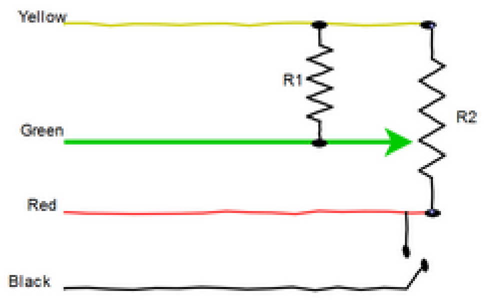


Figure 4.24 Anemometer Schematic

Since the anemometer is not powered directly by any source; in order to read from the reed switch and convert these reading to real values for the next stage of the circuit, a PNP transistor is used. The transistor chosen for pulse conditioning was the 2N3906.

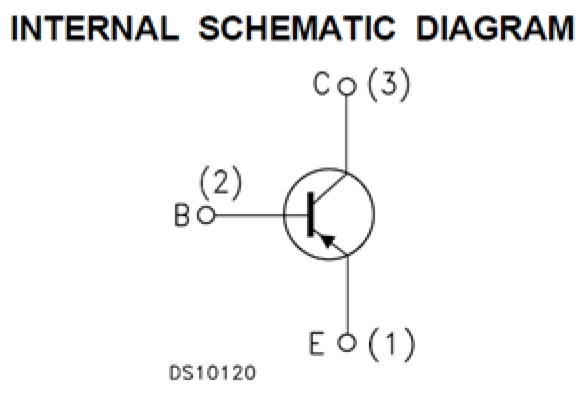


Figure 4.25 Internal Schematic Diagram

The base of the transistor is connected to a 1k ohm resistor; this resistor ends connected with the reed switch from the anemometer. The emitter of the transistor is connected to a 3.3V voltage source, and the collector is connected to a 1k ohm resistor and then to ground. The functionality of this transistor follows 2 paths. The first path being that if the switch is open (no wind) then the base of the transistor is not connected to ground; hence, transistor is off, no voltage output at the collector from the source. The second path being if the switch is closed (wind present) the resistor at the base will be connected to ground, allowing the 3.3V source run to the collector and communicate this output to the next stage of the circuit.

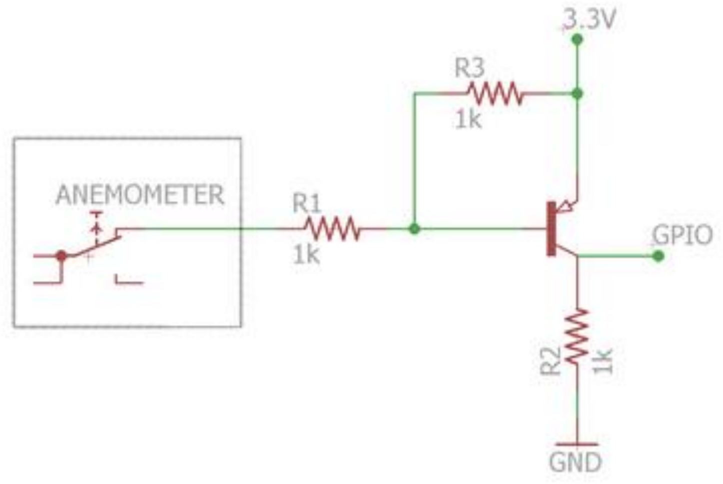
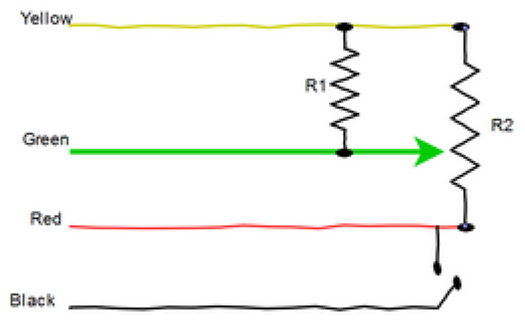


Figure 4.26 Wind Speed Circuit

4.8.2 Wind Direction

The second important value that the anemometer provides us with is the direction of the wind. The anemometer uses a potentiometer to determine direction. The potentiometer can take up to 5V with a current limit of 20mA. The wiring of the anemometer, to measure wind direction is the following:



R1= 909k ohm
R2= 0 to 20k ohm Potentiometer

Figure 4.27 Anemometer Schematic

The green and yellow wires are used to measure the wind direction. The red wire represents ground.

In order to condition the value received from the anemometer, an op amp is used again. The TL084 is an op amp with four (4) stages. Two (2) of these stages were previously used in the conditioning of the AGC voltage. The input from the wind direction is pushed into the positive input V+ of the op amp while the 3.3V is pushed into the negative input of the op amp, V-.

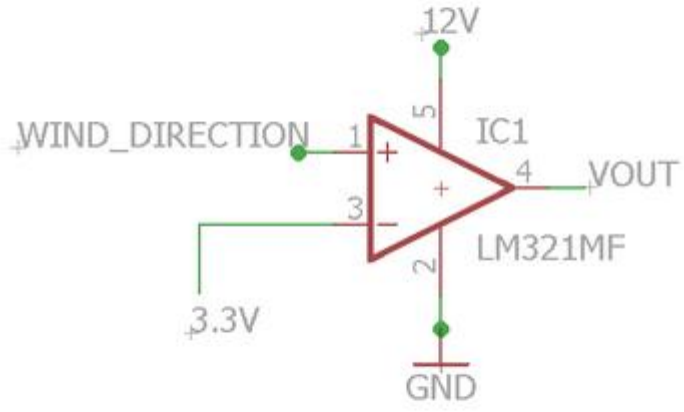


Figure 4.28 Op-Amp

The output of the op-amp is then sent to the analog to digital converter, like the AGC voltage, this value needs to be transformed into a value that can be accepted by the raspberry pi. The analog to digital converter provides the microcontroller with an isolated voltage value that is then adjusted in the program to determine the right wind direction.

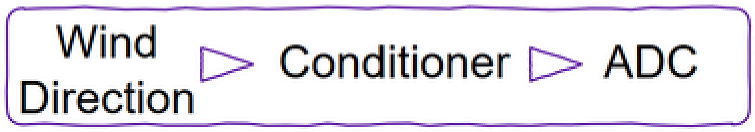
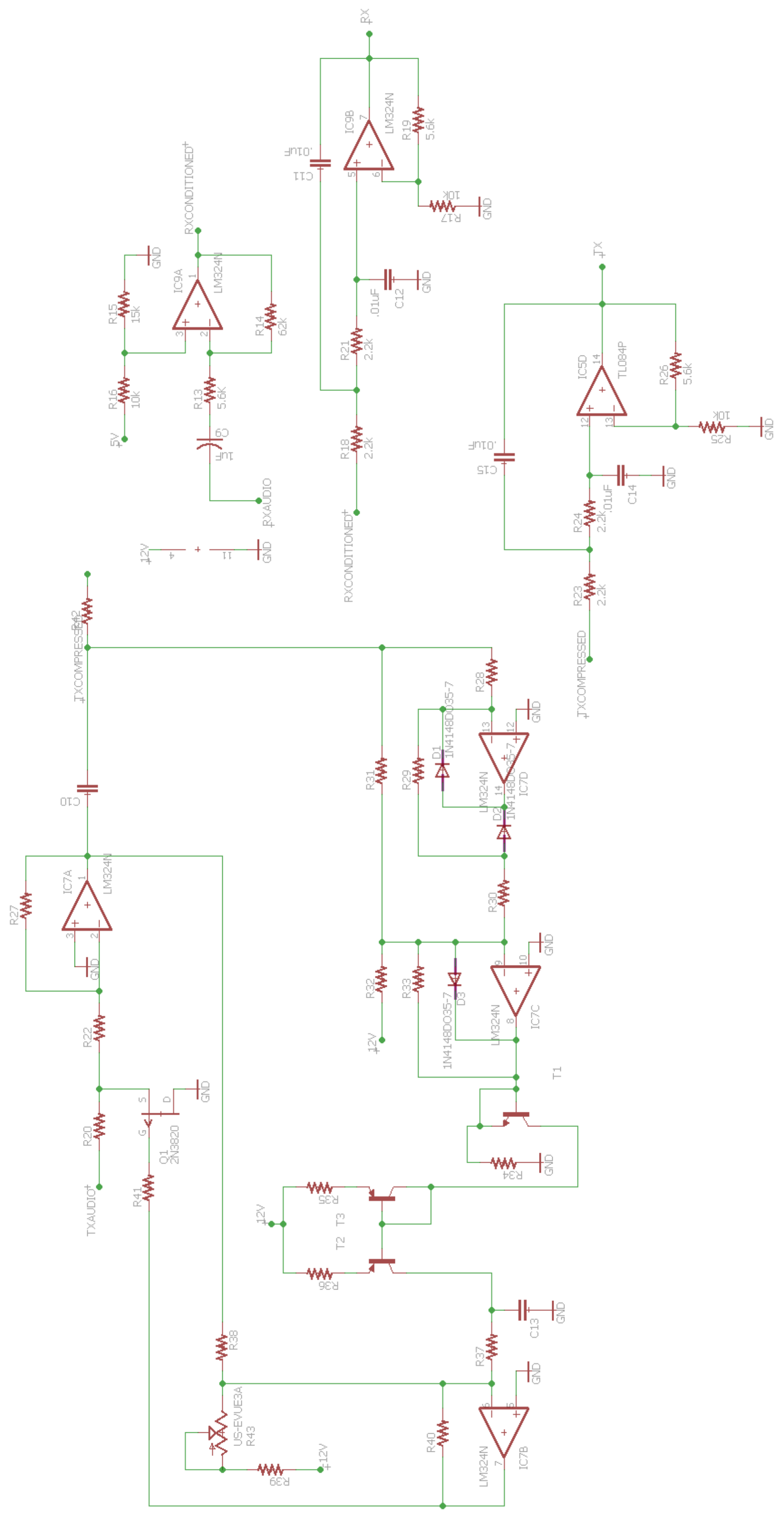
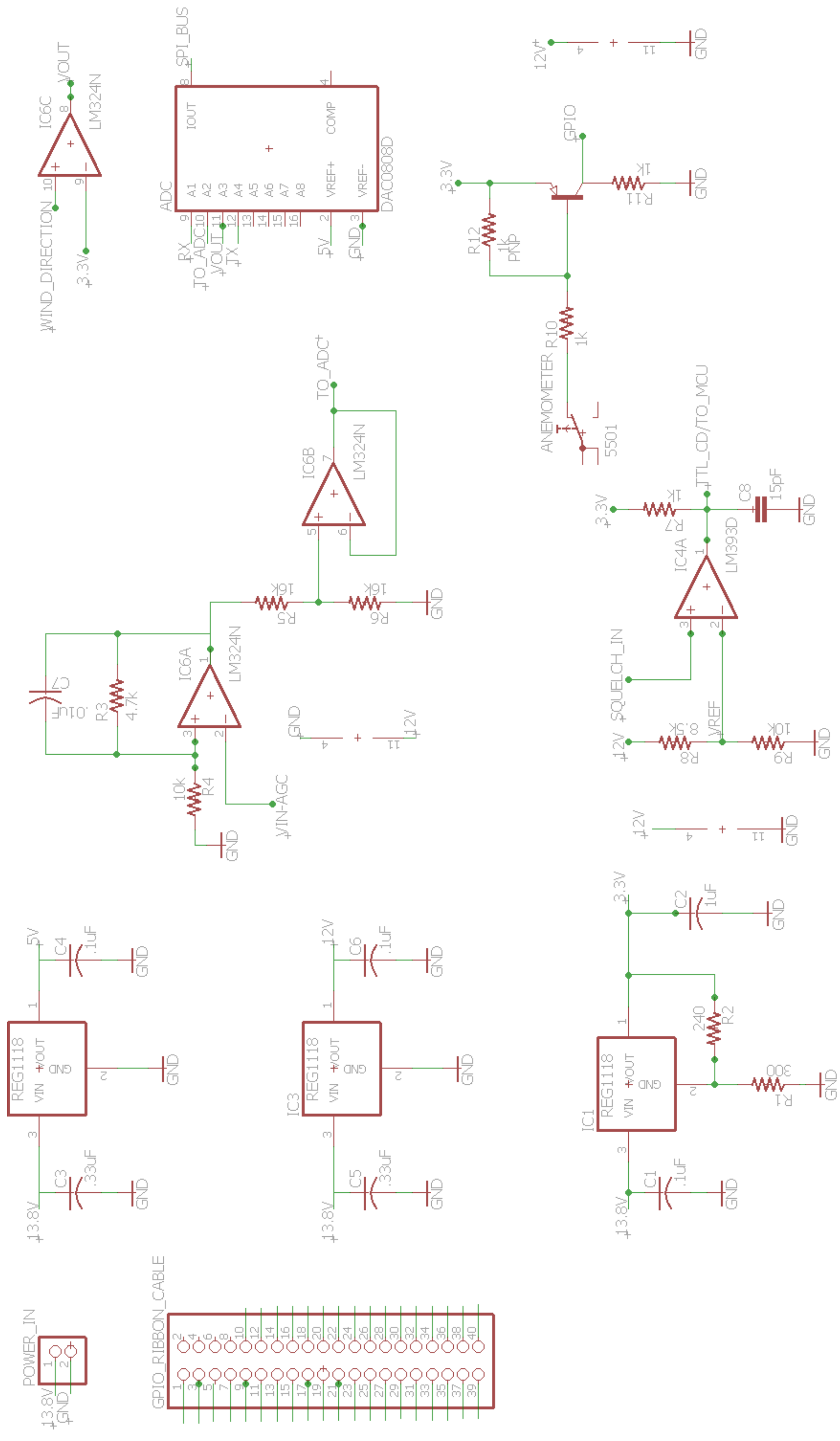


Figure 4.29 Data Flow

4.9 Master Schematic

All the circuits shown on the next pages will be implemented into the interface board. They will share the same power supply, and will have their inputs coming from the King KX-170B VHF Aircraft radio or the anemometer.


RX
IC9B
LM324N
R19
5.6k
R17
10K
GND
C11
.01uF
C12
.01uF
R21
2.2K
R18
2.2K
RXCONDITIONED
IC9A
R15
15k
R16
10k
5V
R14
62k
R13
5.6K
C9
1uF
RXAUDIO
12V
TX
IC5D
TL084P
R26
5.6K
R25
10K
C15
.01uF
C14
.01uF
R24
2.2K
R23
2.2K
TXCOMPRESSED
R42
C10
IC7A
R27
R22
R20
TXAUDIO
Q1
2N3820
R41
R28
R29
R31
D1
1N4148DO35-7
D2
IC7D
R30
R32
R33
D3
IC7C
T1
R34
R35
R36
T2
T3
C13
R37
R38
US-EVUE3A
R43
R39
R40
IC7B


WIND_DIRECTION
VOUT
IC6C
LM324N
3.3V
SPI_BUS
ADC
DAC0808D
IOUT
COMP
VREF+
VREF-
A1
A2
A3
A4
A5
A6
A7
A8
RX
TO_ADC
VOUT
TX
5V
GND
12V
GND
3.3V
GPIO
R11
1k
R12
1k
PNP
R10
1k
ANEMOMETER
5501
TO_ADC
IC6B
LM324N
R5
16K
R6
16K
GND
C7
.01uF
R3
4.7K
IC6A
LM324N
R4
10k
VIN-AGC
GND
12V
3.3V
R7
1k
TTL_CD/TO_MCU
C8
15pF
IC4A
LM393D
SQUELCH_IN
VREF
12V
R8
8.5K
R9
10K
GND
12V
GND
5V
C4
.1uF
REG1118
VIN
VOUT
GND
C3
.33uF
13.8V
12V
C6
.1uF
IC3
REG1118
C5
.33uF
13.8V
3.3V
C2
1uF
IC1
REG1118
R2
240
R1
300
C1
.1uF
13.8V
POWER_IN
13.8V
GND
GPIO_RIBBON_CABLE

5 Software Design

As mentioned in Chapter 3 our software will be running solely on the Raspberry Pi in the Python language. The code will utilize the Django framework for the database aspect of the software. This will require a model for the database structure. This model will include aspects of wind conditions that our sponsor wants saved. These attributes include date and time, wind direction, wind speed, variable wind conditions if detected, and wind gust if detected, but this list can be expanded in the future.

One apparent issue with the software design is the need to constantly check for two things – carrier detect and wind conditions – constantly and share variables between these processes. Redis will be used to fulfill this need. This will use some RAM on the Raspberry Pi, but not enough to decrease the system's performance and is much better than the alternative of constantly writing and reading to a shared file between the processes.

This chapter is focused on the overall design of the software using structured flowcharts. Every process past the main logic loop is called on by a previous process. And with the exception of some initialization function calls, every process has one entry point and one exit point. Using this type of flowchart allows easy understanding by all parties and a direct correlation to pseudocode.

5.1 Main Logic Loop

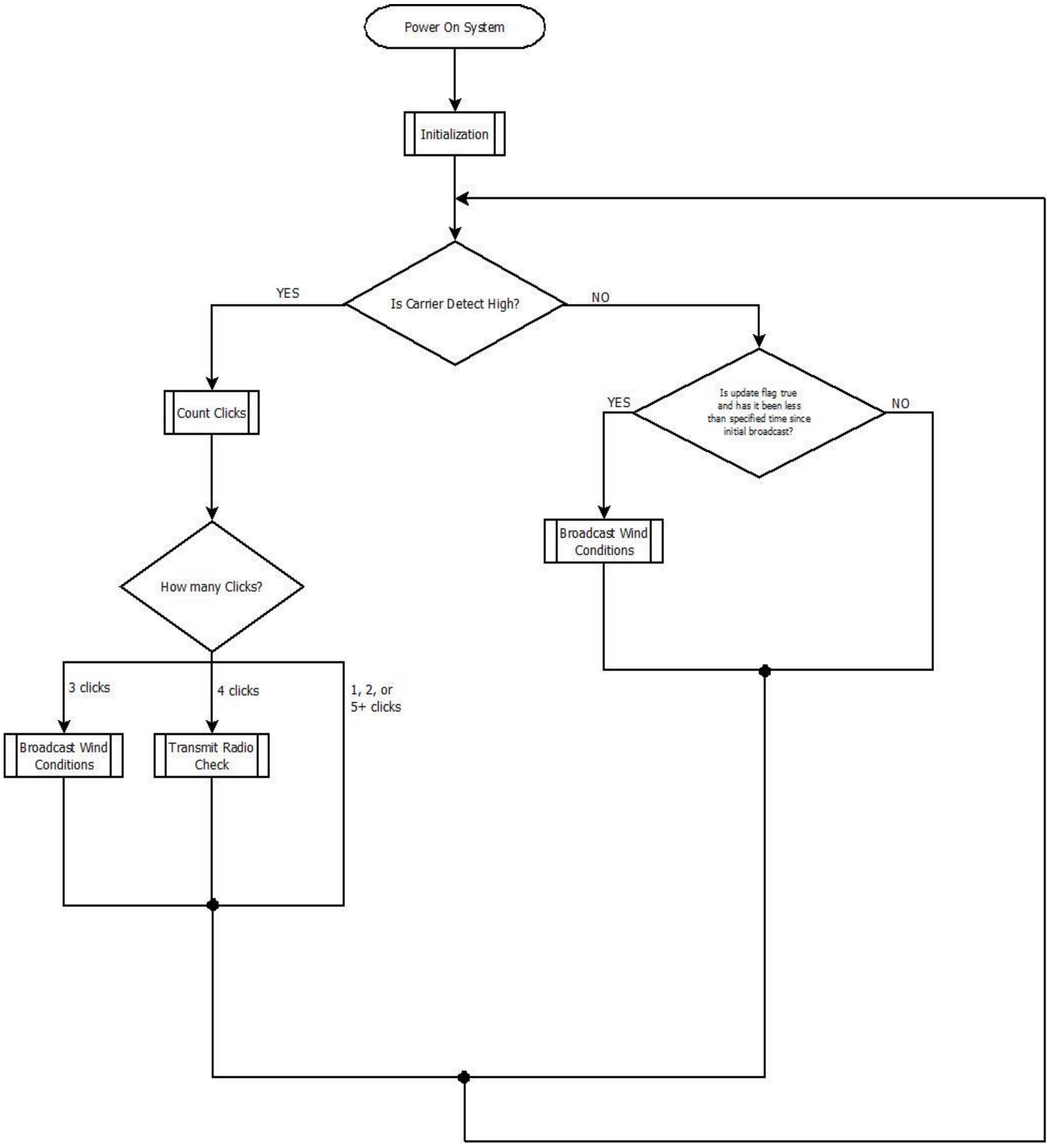


Figure 5.1 Main Logic Loop

This is the main loop of our system's software. It starts with powering on the system. Once the system is powered up the software will run through an initialization process. At this point the software will check if the Carrier Detect (henceforth referred to as CD) signal is high. If the CD signal is high, the software will start the process of “Count Clicks”. This process returns the total number of clicks counted and is explained in more detail in Section 5.2.

Now that the software has the number of clicks from the “Count Clicks” process, the software will determine what action to perform. If there were three clicks, the software will run the “Broadcast Wind Conditions” process, which is explained in Section 5.1.2. If there were four clicks, the software will run the “Transmit Radio Check” function, which Is explained in Section 5.1.3. If the software detects there were less than three or more than four clicks, the software will do nothing and go back to check if the CD signal is high. This ensures that if a pilot initiates a function or keys their mic for other reasons, the software will perform the necessary function and go back to checking for CD.

However, if the CD signal is not high, the software will check if the Update flag (a variable set by a separate wind data collecting process) is TRUE and that it has not been longer than a specified time since the initial wind conditions broadcast. If both of these conditions are true, the software will run the “Broadcast Wind Conditions” process. But if either condition fails, the software will do nothing and go back to checking for CD. For example, if the specified timeout for wind condition updates is 5 minutes, and the update flag is true but it has been 6 minutes since the initial wind conditions broadcast, the software will not broadcast an update and will go back to checking for CD.

5.1.1 Initialization

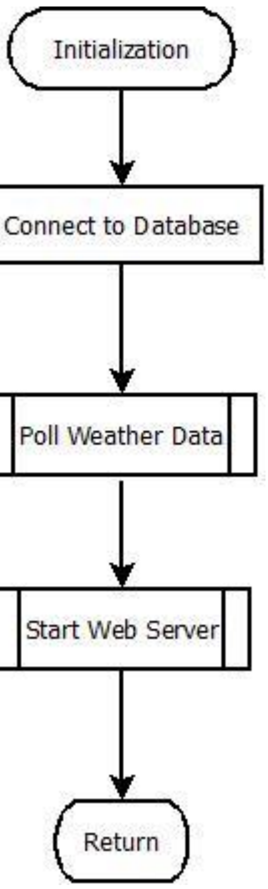
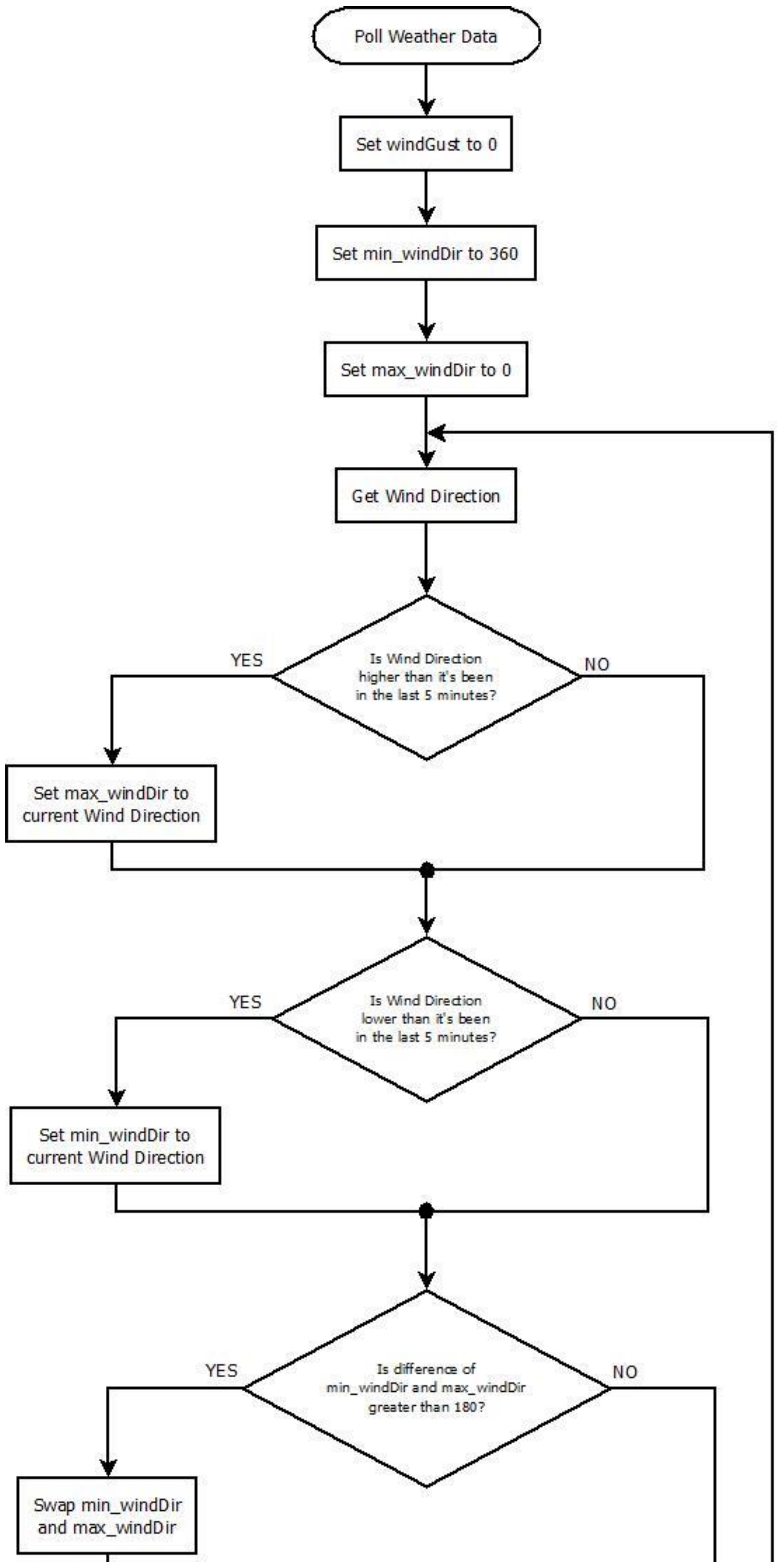


Figure 5.2 Initialization

This is the Initialization process of the software. Once the system is powered on, this process will be immediately called and executed. In this process there are three main commands. First, the computer will connect to the database that will be set up. Next, the software will initiate the never ending process of polling the weather data, which will gather and record data from the anemometer and is further explained in the following section. Lastly, the software will start the web server that will be running from the computer. From there, the software will return back to the Main Loop.

5.1.1.1 Get Weather Data


Poll Weather Data
Set windGust to 0
Set min_windDir to 360
Set max_windDir to 0
Get Wind Direction
YES
Is Wind Direction higher than it's been in the last 5 minutes?
NO
Set max_windDir to current Wind Direction
YES
Is Wind Direction lower than it's been in the last 5 minutes?
NO
Set min_windDir to current Wind Direction
YES
Is difference of min_windDir and max_windDir greater than 180?
NO
Swap min_windDir and max_windDir

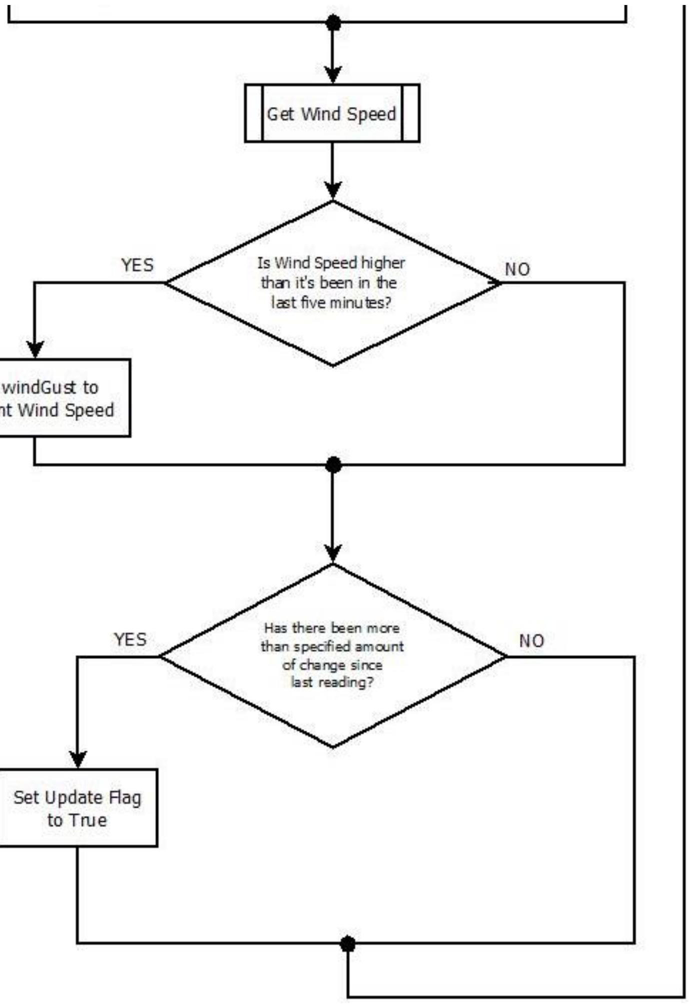

Get Wind Speed
Is Wind Speed higher than it's been in the last five minutes?
YES
NO
Set windGust to current Wind Speed
Has there been more than specified amount of change since last reading?
YES
NO
Set Update Flag to True

Figure 5.3 Get Weather Data

The Get Weather Data process is started by the initialization process. Once this process is started, it never stops, and continuously gets new wind data from the anemometer. At the start of the process, the software will set a wind gust and max wind direction variables to 0 and min wind direction variable to 360, but these variables will later hold the value of the most recent wind gust, min wind direction, and max wind direction. The software will then get the wind direction with a simple command and check if a new min or max wind speed needs to be set. Then the process will get the wind speed with a sub process. Then it will check if the recorded wind speed is higher than it has been in the last 5 minutes (this amount of time can be changed). If so, the wind gust variable will be updated to that recorded value. Next, the software will check if there has been a significant change in conditions since the last reading. If so, the update flag will be set to true for use in the main loop of the software. This process does not return and will continue indefinitely.

5.1.1.2 Start Web Server

This process will be added in the Fall for Senior Design 2

5.1.2 Count Clicks Process

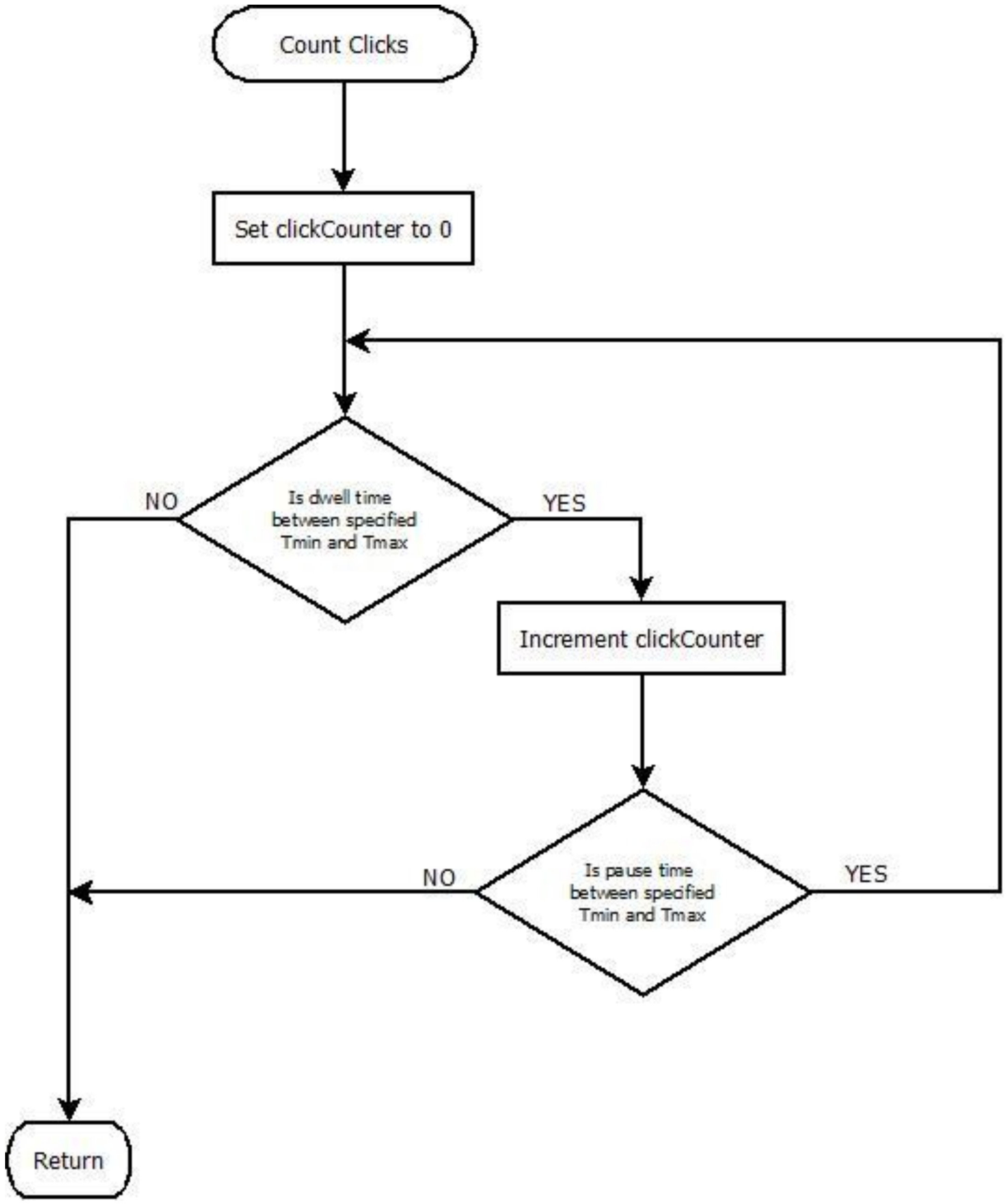
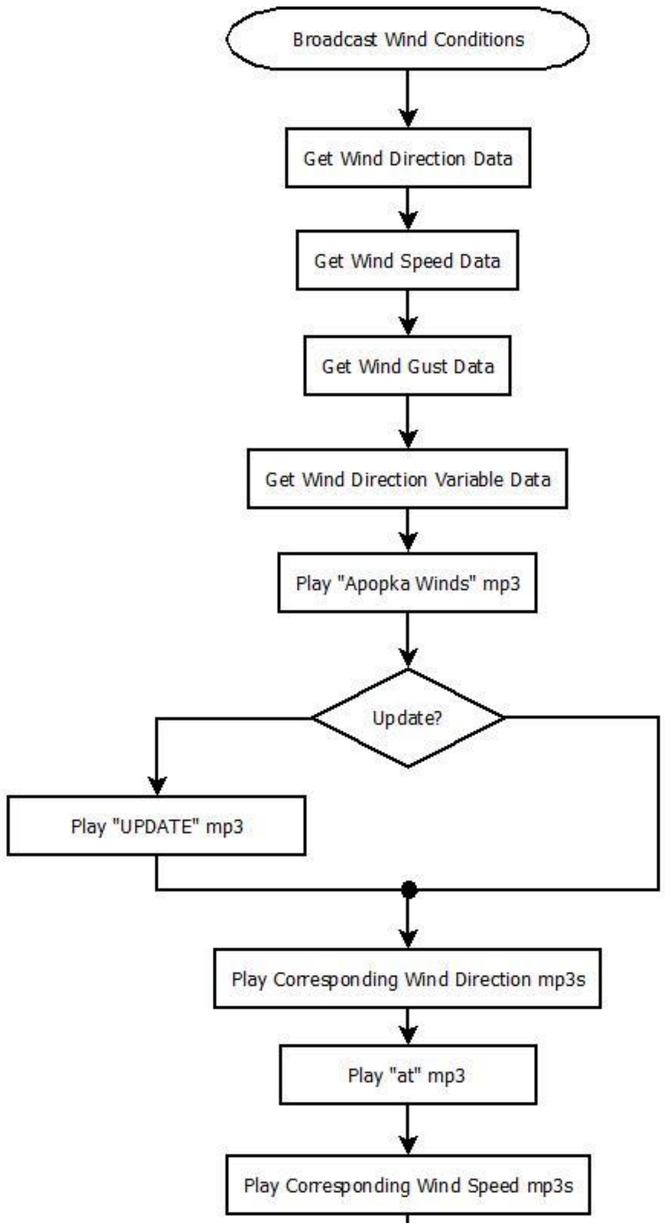

Count Clicks
Set clickCounter to 0
Is dwell time between specified Tmin and Tmax
NO
YES
Increment clickCounter
Is pause time between specified Tmin and Tmax
NO
YES
Return

Figure 5.4 Count Clicks

This process counts the total number of times the pilot keys their mic, or number of clicks. When this process is activated, the software first sets the clickCounter variable to 0. Then it checks that the dwell time, that is the total time that CD is high, is between the user specified min and max times. If it is, the software registers this as 1 click and sets clickCounter to 1. Then it checks that the pause time, that is the total time between clicks or the total time that CD is low, is between the user specified min and max times. If it is, the software goes on to check for another appropriate dwell time and then pause time. It keeps alternating these checks while also incrementing clickCounter. If at any time the dwell time or pause time is not within the user specified min and max times, the process will exit back to the main loop and return the latest value for clickCounter.

5.1.3 Broadcast Wind Conditions Process


Broadcast Wind Conditions
Get Wind Direction Data
Get Wind Speed Data
Get Wind Gust Data
Get Wind Direction Variable Data
Play "Apopka Winds" mp3
Update?
Play "UPDATE" mp3
Play Corresponding Wind Direction mp3s
Play "at" mp3
Play Corresponding Wind Speed mp3s

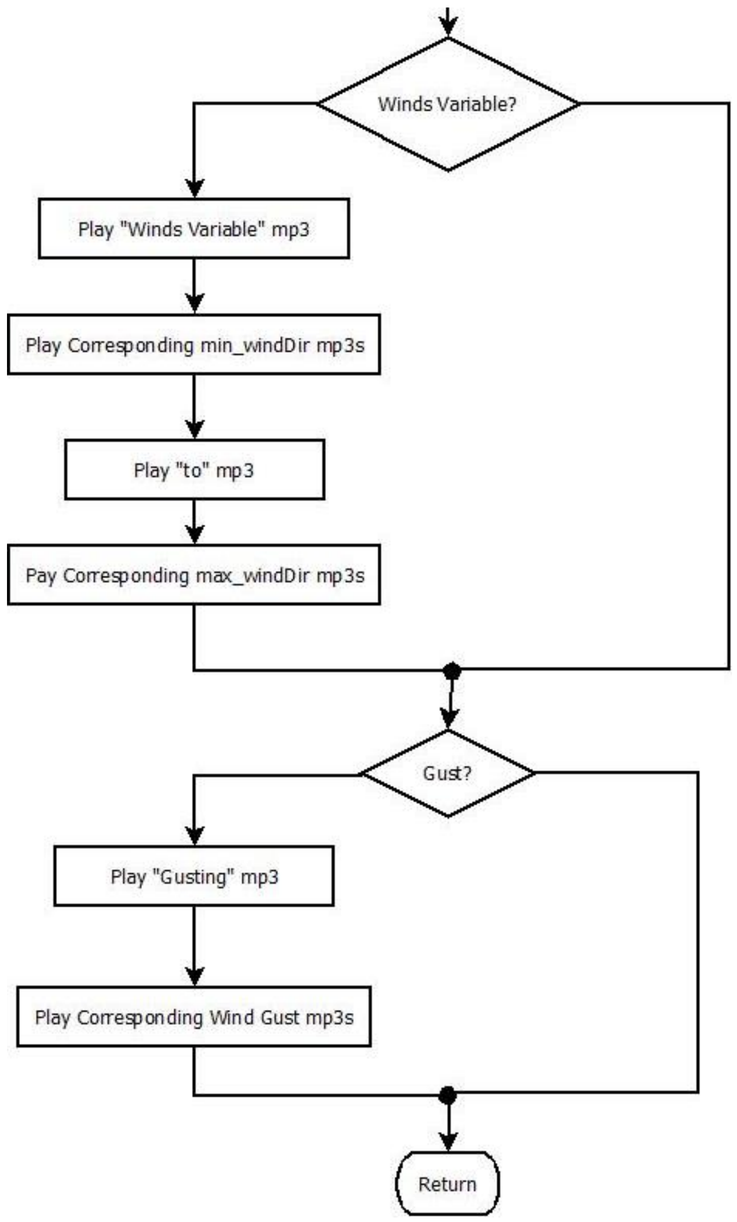

Winds Variable?
Play "Winds Variable" mp3
Play Corresponding min_windDir mp3s
Play "to" mp3
Pay Corresponding max_windDir mp3s
Gust?
Play "Gusting" mp3
Play Corresponding Wind Gust mp3s
Return

Figure 5.5 Broadcast Wind Conditions

This process starts after three clicks have been recognized by the main loop of the software. It starts by grabbing the necessary data from the Redis stores. After grabbing the wind direction data, wind speed data, wind gust data, and wind direction variable data, the process will play an mp3 file that speaks, “Apopka Winds”. Then it will determine the status of the Update flag. If true, the software will play an mp3 file that speaks, “Update”. At this point the software will need to play the corresponding mp3 files to speak the correct values of the wind direction and speed. Wind direction will be separated by digit and spoken in that fashion (e.g. 270 degrees would be spoken “2 – 7 – 0”), then an mp3 file that speaks, “at” is played, and finally the wind speed is spoken in a normal format. This will require the software to determine the ten’s value and the one’s value of the wind speed and play the corresponding mp3 files (e.g. 27 knots will have to be broadcast as “20” and “7” and when followed in direct succession will sound like a normal “27”).

At this point the software will have to do some calculations to determine if additional information needs to be broadcasted. If there are variable winds, meaning the direction winds are blowing changes more than a few degrees, the software will play corresponding mp3 files for the min and max directions in the same way they are played above. Then if the winds have gusted significantly in the past hour, the software will play corresponding mp3 files to broadcast the max gust wind recorded in the allotted amount of recorded time.

5.1.3.1 Save Wind Data to Database

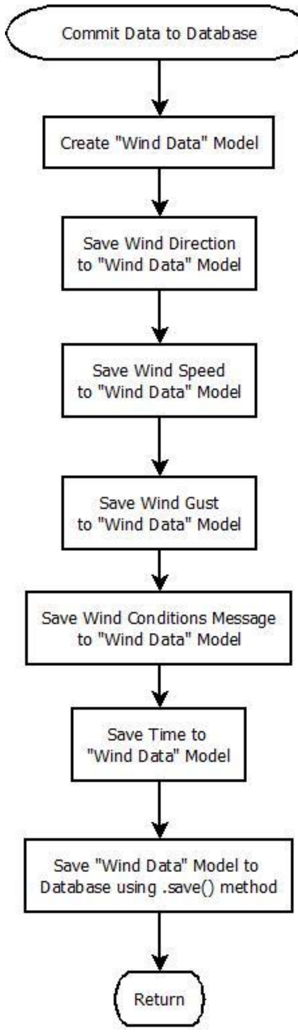

Commit Data to Database
Create "Wind Data" Model
Save Wind Direction to "Wind Data" Model
Save Wind Speed to "Wind Data" Model
Save Wind Gust to "Wind Data" Model
Save Wind Conditions Message to "Wind Data" Model
Save Time to "Wind Data" Model
Save "Wind Data" Model to Database using .save() method
Return

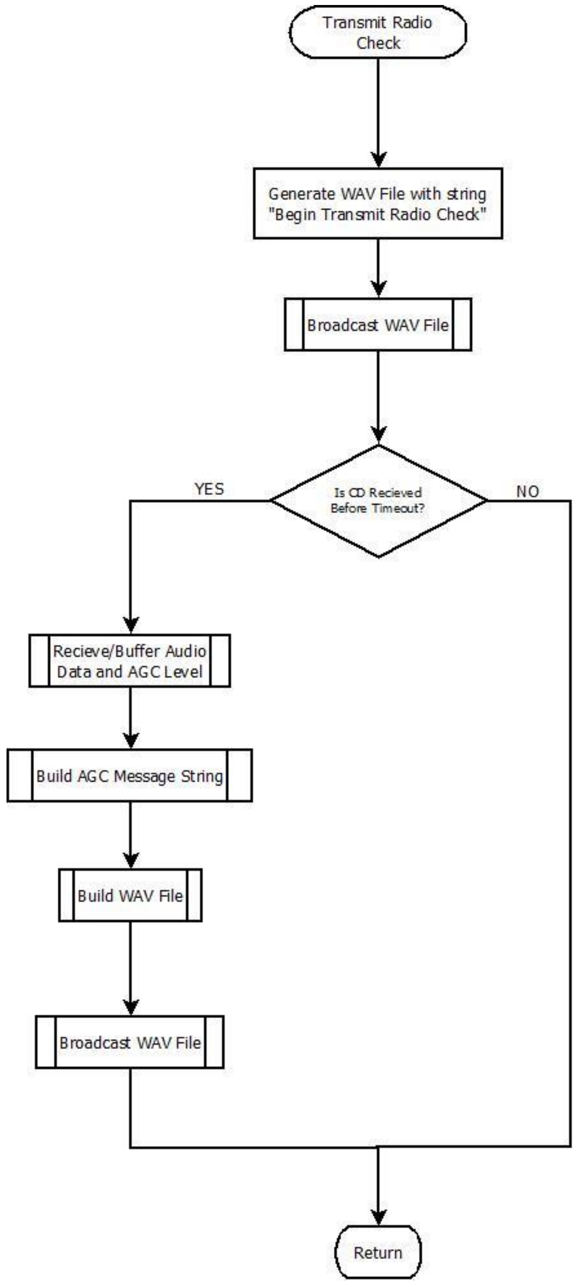
Figure 5.6 Commit Data to Database

This is the process in which data recorded by the anemometer get committed to the database. The first step is to create a model, which is a form of an object for a database in Django. After the model is created all the new data can be saved to it. The wind speed, wind direction, gust, message string, and current time are all saved as parameters in the model. From there, the software will save the entire model to the database using a Django save method. Once all this is completed, the process will return to the process that called it.

5.1.4 Transmit Radio Check

There are two possible Transmit Radio Check processes with one using an Analog to Digital Converter to get incoming audio data and the other using a USB audio Interface to record incoming audio.

5.1.4.1 Transmit Radio Check with Analog to Digital Converter



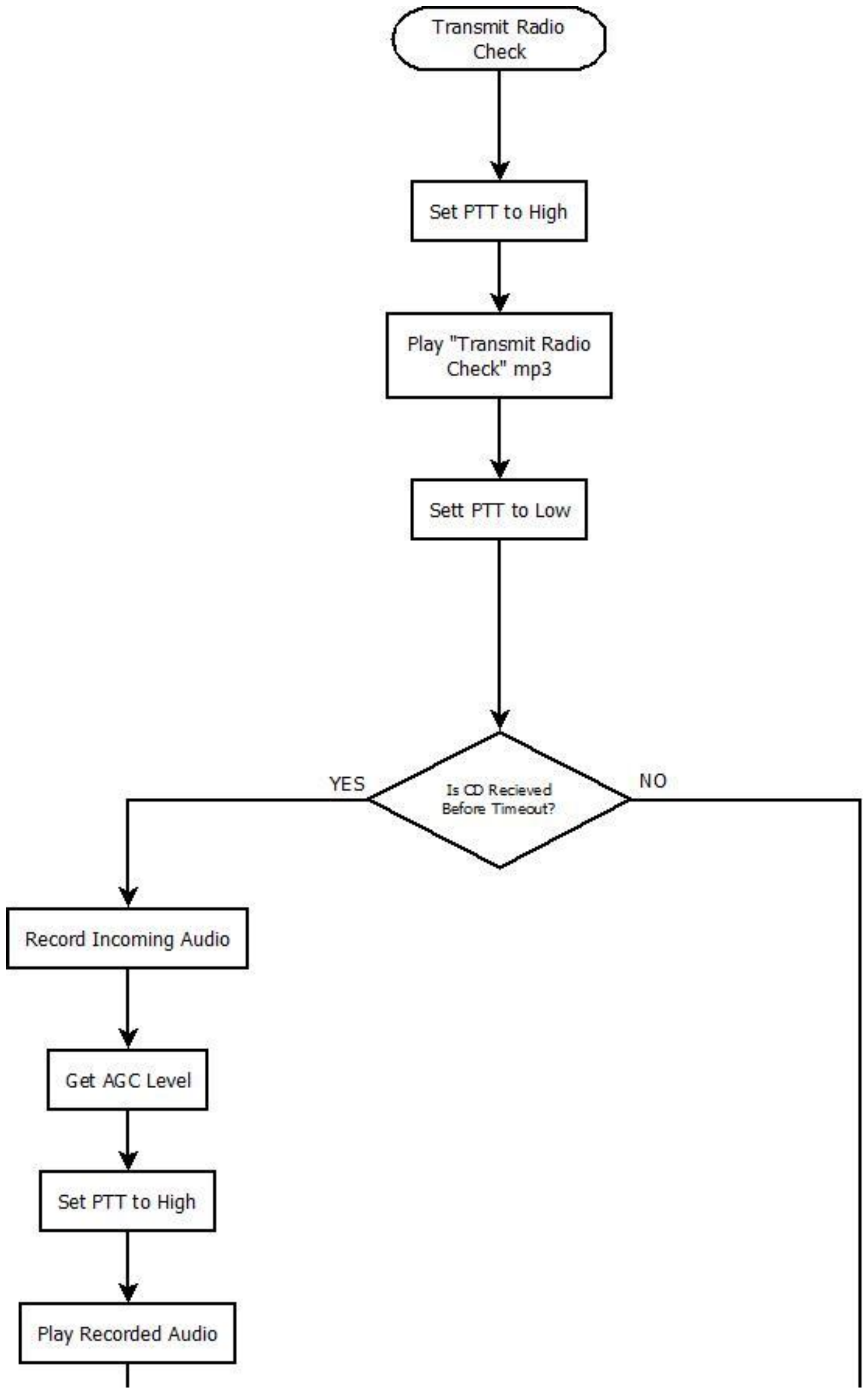
Transmit Radio Check
Generate WAV File with string "Begin Transmit Radio Check"
Broadcast WAV File
Is CD Recieved Before Timeout?
YES
NO
Recieve/Buffer Audio Data and AGC Level
Build AGC Message String
Build WAV File
Broadcast WAV File
Return

Figure 5.7 Transmit Radio Check using ADC

This “Transmit Radio Check” process is started when the software registers four clicks from a user and utilizes the ADC as a way to get incoming audio data. It starts by prompting the user for a transmit radio check. Then the software checks if a CD is received within a timeout window (probably around 5 seconds). If a CD isn’t registered, the software will simply return to the main loop without doing anything. If there is a CD registered, four subprocesses will be executed; receive/buffer audio data and AGC level, build the AGC message string, build the WAV file, and broadcast the WAV file.

These subprocesses use the ADC. When receive/buffer audio data and AGC level is called the process receives all the incoming data from the ADC and at some point gets the AGC level also from the ADC. Then the process builds a string to tag onto the end of the recording when it’s broadcast. Then the WAV file actually gets built and lastly the process broadcasts the WAV files. Once these steps are completed, the software will return back to the main loop.

5.1.5 Transmit Radio Check Process with USB Audio Device



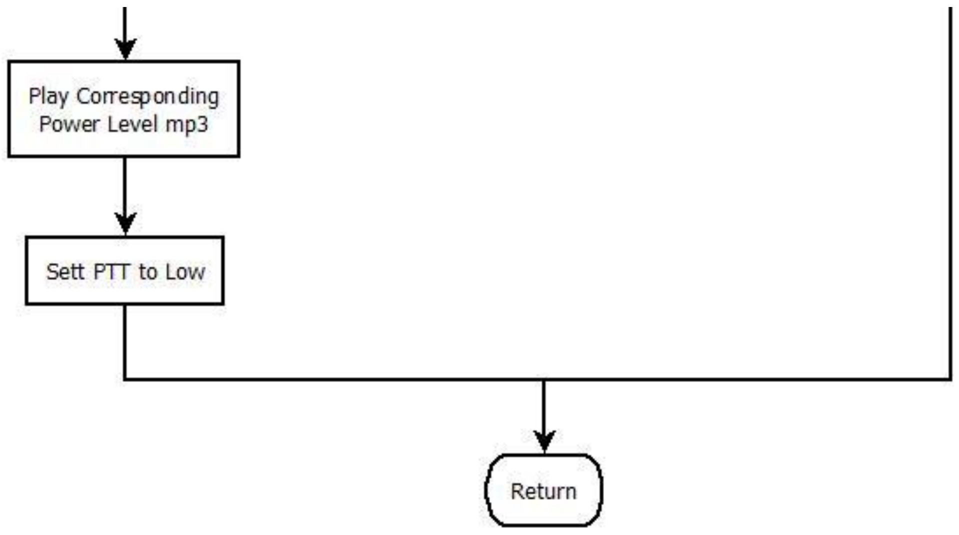
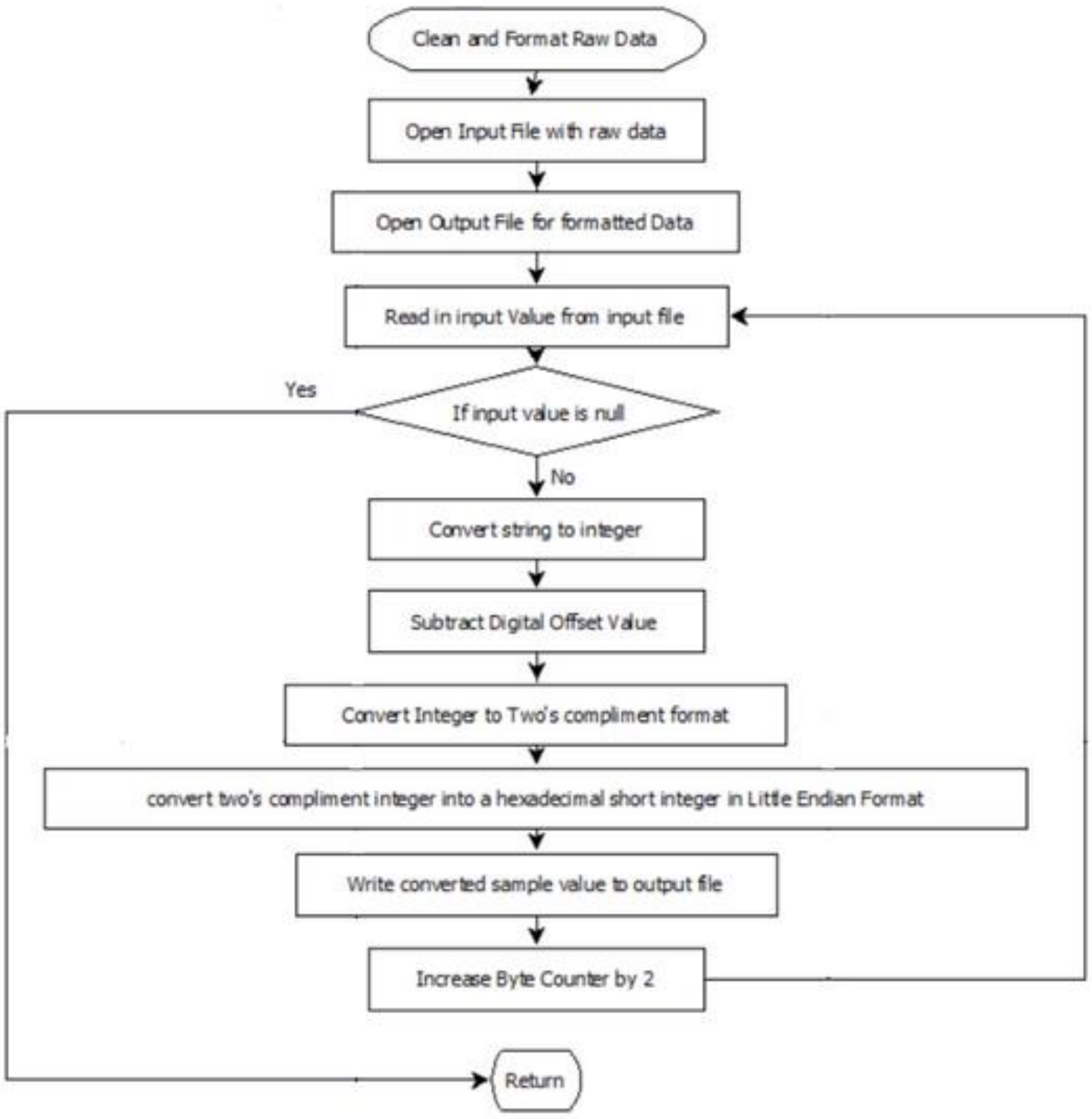


Figure 5.8 Transmit Radio Check

This “Transmit Radio Check” process is started when the software registers four clicks from a user and utilizes a USB audio interface. It starts by prompting the user for a transmit radio check. Then the software checks if a CD is received within a timeout window (probably around 5 seconds). If a CD isn’t registered, the software will simply return to the main loop without doing anything. If there is a CD registered, the process will start recording audio from the USB interface. While it’s recording, the AGC level will be grabbed from the ADC. At this point the radio will be put into transmit mode and the recorded audio as well as the calculated power level will be broadcast to the user, then the radio will be taken out of transmit mode. Once these steps are completed, the software will return back to the main loop.

5.1.5.1 Clean and Format passed in Audio Data


Clean and Format Raw Data
Open Input File with raw data
Open Output File for formatted Data
Read in input Value from input file
If input value is null
Yes
No
Convert string to integer
Subtract Digital Offset Value
Convert Integer to Two's compliment format
convert two's compliment integer into a hexadecimal short integer in Little Endian Format
Write converted sample value to output file
Increase Byte Counter by 2
Return

The "Clean and Format Raw Data" function is the main function responsible for taking in the sampled Audio data, take from the MCP3008 Analog to digital converter, and makes sure that the data is valid. Once assuring that the data is valid, this function will convert the raw data into the required two's compliment and little endian formats. The input and output of this data will be through file input and output. We considered simply using a byte array in memory, however the size of the audio data can very quickly exceed the amount of spaced occupied by a python variable. To avoid this issue, we will simply pull the sampled 10–bit digital reading from the analog to digital converter from a text file and then output the corresponding formatted audio data into an output text file.

The first task of this function is to initialize the file variables that we will need. This includes the input file that will be providing the sample values and then the output file that will store the formatted data. From here we will begin reading in the unformatted audio data and read in the first digital value. This is the first command in a loop with the exit condition being represented by the condition block in the flow chart. To decide whether or not we need to exit the loop we need to determine if we are at the end of a file. In python, this is determined by if we have read in a null byte. If we have received a null byte, we exit this subroutine. If we do not receive a null byte, then the value read represents the current sample we need to format. In order to format the sample, we need to first convert the read in sample from a string to an integer. Here is where we need to briefly revisit how the interface board is manipulating the audio signal before it is sent through the analog to digital converter.

Since the MCP3008 does not read negative values, the interface board is offsetting the voltage by 3V. This means that the base value of all the readings read in are 3V, or have a digital offset value of 522. So the next thing that we need to do is to subtract the read in digital value by an offset of 522. This will effectively allow us to receive in negative readings, which is important because audio signals have both positive and negative readings. Once the offset has been accounted for, we need to convert the new calculated sample and encode it into the two's compliment format. Two's compliment is the specified format used by the WAVE file format standard to account and accurately represent the samples taken in the negative range of the audio signal. Considering that the MCP3008 is a 10-bit analog to digital converter, we will be using 16 bits to represent the sample in the wave files we create. This is because the format chunk of a wave file, needs to have the sample size in bytes represented so that any software playing the wave file know how to effectively read the formatted data. Since 10 bits is more than one byte, we need two bytes, or 16 bits, to represent each sample. We are not using multiple channels on the MCP3008 to read in multiple audio channels, as for the purposes of this design, a Mono channel audio set up is sufficient. This means we only need one channel sampling audio data as opposed to two channels for a left and right stereo audio setup. If we did have left and right channels, we would need a total of four bytes, two bytes for the left and right channel; however, for a single channel Mono audio set-up, two bytes are all that is needed. This two-byte window is equivalent to a short integer in python. While python instinctively determines the type of variable that is stored, which is usually a 32-bit integer, we must tell python to force the variable storing the current two's complimented audio sample to by a short integer that is 16 bits, effectively two bytes in length. Also, because wave files store these samples in Little Endian format, we must tell python to force this short integer to be in this particular byte order standard. Both the tasks of forcing the converted two's complimented audio sample to be a short integer with a length of 2 bytes, and forcing the byte order of that short integer to be in a little endian format in the same line of code. Therefore, these two steps are represented as a single block and are effectively treated as one step in this subroutine. Lastly we need to write the formatted audio sample to the output file we will use to build the wave file in another subroutine as well as increase the bytesRead counter by 2. This bytesRead counter is responsible for keeping track of the total bytes of raw audio data were converted, and will be responsible for returning the amount of audio data needed to be written, in bytes, to the finalized data chunk of the wav file. We are increasing this bytesRead counter by two, because there are two bytes being represented for each digital audio sample we read in from the input file. Since the total size of the data needs to be represented in bytes, we must increase this counter by two bytes for every sample.

5.1.5.2 Build WAV File

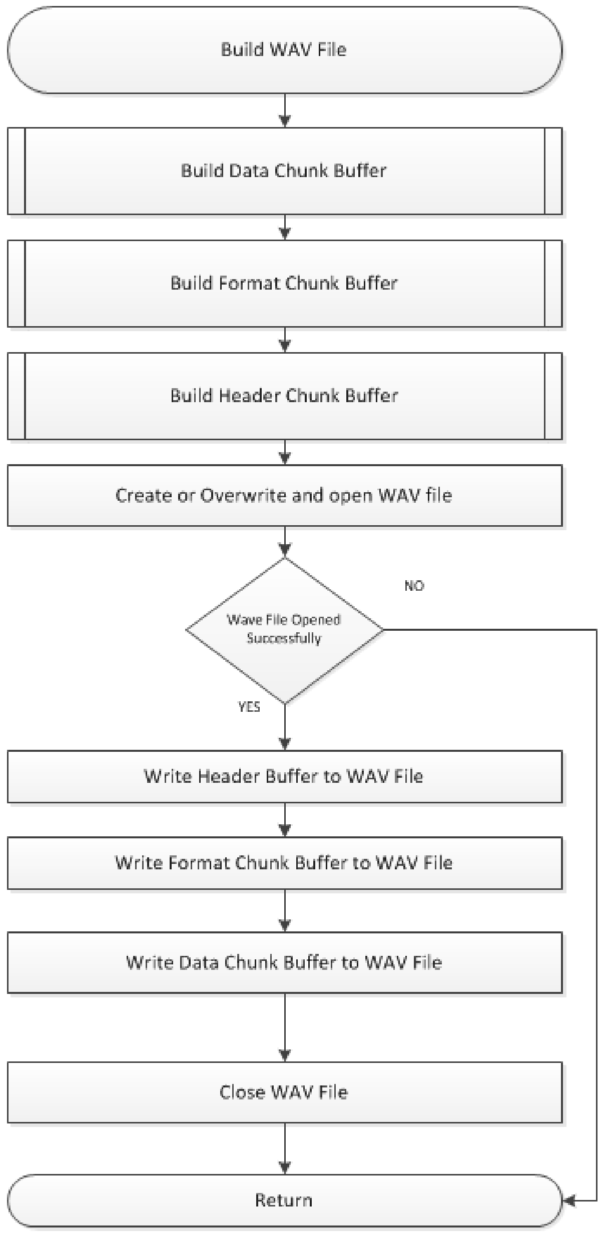

Build WAV File
Build Data Chunk Buffer
Build Format Chunk Buffer
Build Header Chunk Buffer
Create or Overwrite and open WAV file
Wave File Opened Successfully
NO
YES
Write Header Buffer to WAV File
Write Format Chunk Buffer to WAV File
Write Data Chunk Buffer to WAV File
Close WAV File
Return

Figure 5.9 Build WAV File

The Build WAV File sub process is used to take the read in Audio data from the Audio Channel of the Analog to Digital Converter and create a .wav file that can be replayed and broadcasted out back through the radio. To understand this process, you must understand a small piece of how WAV files are structured. A WAV file is a type of file called a Resource Interchange File Format, or “RIFF”, file. RIFF files have a Header Section, depicting what kind of RIFF file (WAV, MP4, etc.) it is followed by multiple sections called “chunks”. The WAV format has 2 chunks which include the “format” chunk and the “data” chunk. The format chunk, sets key fields that tell the program reading the file how to handle the audio for playback. The Data chunk is essentially the pure audio data in whatever recording format was set by the format section. This means that in total the WAV file format has 3 main sections: the Header, the Format Chunk, and the Data Chunk. In this section there will be three sub sections containing the sub processes to create, populate, and format the data for each of these three sections. In order to understand each of the sub processes for each section of the WAV file below, we will need to go over the necessary fields for each section/chunk of the WAV file. These fields are necessary in order to understand the flow of each sub process whose purpose is to format and populate the data and fields for each section. Therefore, it is also necessary to include these field tables with a brief description of the fields which is done below.

Header Section

Field Name	Size (in bytes)	Endian-ness
chunkID	4	Big
chuckSize	4	Little
riffFormat	4	Big

The first section of a WAV file is the Header which consists of 12 bytes of data. The first 4 bytes, and the first field of this section is the Chunk ID. This Chunk ID is unique since it is the identifier indicating what type of file it is. This field always contains a value of “RIFF” to indicate it is a Resource Interchange File Format file. The Second field is the chunkSIze; this field is also unique as it contains the size of the entire file in bytes minus the 8 bytes of data (for the chunkSize and chunkID) in this Header Chunk of the WAV file. Lastly is riffFormat field which tells the program reading the file what type of RIFF file it is reading. Since we are writing to a WAV file, this field will always be set to “WAVE”.

Format Chunk Section

Field Name	Size (in bytes)	Endian-ness
chunkID	4	Big
subChunkSize	4	Little
audioFormat	2	Little
numOfChannels	2	Little
sampleRate	4	Little
byteRate	4	Little
blockAlign	2	Little
bitResolution	2	Little

The Format Chunk contains multiple fields that let the program reading the file know how the actual audio data is formatted. Like the Header section, the Format Chunk contains a chunkID and a chunkSize desribed as subChunkSize. However, these fields refer strictly to the data included in the Format Chunk as opposed to the entire file. The chunkID is always set to “fmt” indicating it is the format chunk. The space character is included in this field to fill out the 4th byte so the file does not include a null byte in the middle of the data. Likewise, the subChunkSize field includes the entire size of the Format Chunk which will always be 16 bytes in length letting the reading program know how many bytes there are before the next section. Next is the audioFormat field which in this case will be set to a value of 1 indicating that the audio data is in a Pulse Code Modulated audio format. Next is the numOfChannels field which, as its name implied, contains the number of audio channels being sampled. The two fields above are only 2 bytes in size so the Endian-ness of these fields are not so important. The sampleRate field and byteRate field are pretty self-explanatory and hold the sample rate, in Hz, in which the audio was sampled by the Analog to Digital Converter, and the calculated byte rate of the audio calculated from the sample rate, number of channels and bit resolution. These two fields above are each 4 bytes in length and are formatted in Little Endian. The blockAlign field stores the number of bytes per sample in all sampled channels. The bitResolution field holds the number of bits per sample.

Data Chunk Section

Field Name	Size (in bytes)	Endian-ness
chunkID	4	Big
subChuckSize	4	Little
rawData	variable	Little

The Data Chunk is the final chunk of the WAV file containing the actual audio data sampled by the Analog to Digital Converter. Like the two other chunks, the Data Chunk contains two fields for the chunkID and the overall size of the rawData. The chunkID is always “data” and the subChunkSize is an integer in Little Endian Format. The last section of the Data Chunk and the WAV file is the rawData which, as implied, contains the raw audio data read by the Analog to Digital Contverter. The main thing to note here is that the rawData field has each sample in Little Endian Format. This means with two channel audio with a 16-bit resolution, each set of 16 bytes will be in Little Endian Format but the order in which each sample was take remains in order. This means the ENTIRE SECTION is NOT in Little Endian format, but each SAMPLE is in Little Endian Format. This holds true for each channel. The two channels as a whole are NOT Little Endian, but each channel is of Little Endian format starting with the left audio channel followed by the right audio channel.

In order to build the WAV file from the read in audio data, we first need to build each section of the WAV file in byte array buffers that can be directly written to the WAV file. We can break these up into three different sub processes: Build Header Buffer, Build Format Chunk Buffer, and Build Data Chunk Buffer. These sub processes will create, populate, format, and return the byte array buffer for their particular sections of the WAV file. However, to populate the header section of the WAV file, we need the size of the Format and Data chunks; this means that the two sub processes in charge of returning these sections need to happen before we create the header buffer that contains the fields for the header section. To make things simple we have arranged the sub processes for creating the sections for the WAV file in reverse order in order to process and clean the data, format the WAV file, and then create the header section. This was every sub process for each section of the WAV file has all the information it needs about the other sections before running.

We first run the “Build Data Chunk Buffer” sub process in order to clean and format the audio data read in by our Analog to Digital Converter as well as populate all of the remaining fields of the Data Chunk (these fields are shown below in the Build Data Chunk Buffer sub process section). This sub process will return the byte buffer containing the formatted bytes for the Data Chunk of the WAV file (which includes the formatted audio data from the ADC). Next we run the “Build Format Chunk Buffer” sub process to build, populate and format the Format Chunk of the WAV file. This sub process will return the byte array buffer containing the Format Chunk for the WAV file. Next we run the “Build Header Buffer” sub process which will build, populate, format and return the byte array buffer containing the Header section for the WAV file. This sub process will need to be given the size of the Data Chunk Buffer and the Format Chunk Buffer to properly calculate the fields for the header section (these fields are shown below in the “Build Header Buffer” sub process section).

After all of the byte array buffers for the three sections of the WAV file have been created, formatted, and returned by their respective sub processes, we need to write them to the WAV file. The first step here is to actually create and open a WAV file. Since each Transmission Check will not need to be saved to the system or database and multiple Transmission Checks cannot happen at the same time, we can simply and safely write over the existing WAV file, if there is any, and write the new Transmission Check audio data to the WAV file. If there is no existing WAV file, we will simply create one. We do not include a decision block to determine if we need to create or overwrite the WAV file here, since the python line to create and overwrite a file are exactly the same. However, we do need to check whether the WAV file opened successfully. If there was error opening the WAV file, we return with an error to the parent process, otherwise we continue to writing the sections to the WAV file. Next we will simply write each Sections byte array buffer to the file in order. First we write the Header Buffer to the WAV file; then we write the Format Chunk Buffer to the WAV file, and lastly we write the Data Chunk Buffer to the WAV file. We do not have to create sub processes for writing each buffer to the WAV file because the write method in Python's File object is polymorphic and can take a single variable or any type of array (including a byte array) as its argument. This means the parsing of the buffer to write each byte is abstracted away by Python's File Input/output API and we do not have to worry about creating this method ourselves.   After writing the buffers to the Wav File, we will close the WAV file and return from the Build WAV File sub process.

5.1.5.3 Build Data Chunk Buffer

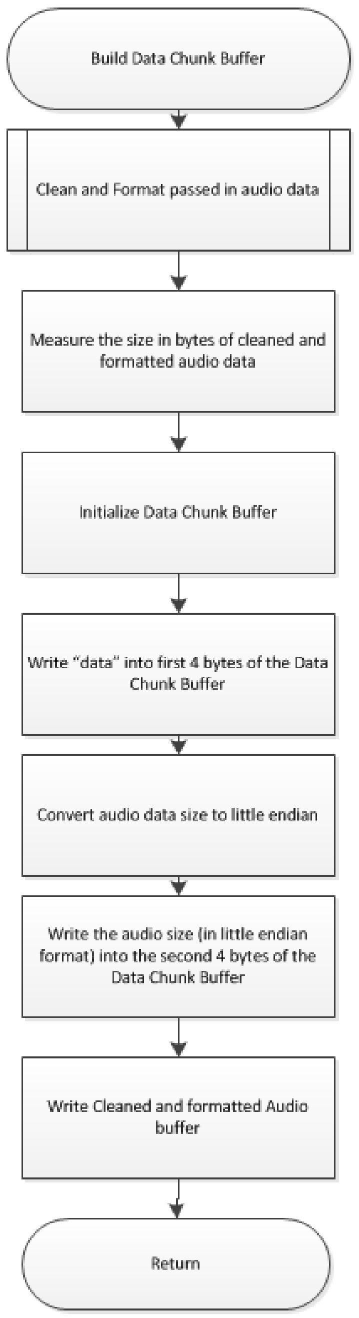

Build Data Chunk Buffer
Clean and Format passed in audio data
Measure the size in bytes of cleaned and formatted audio data
Initialize Data Chunk Buffer
Write "data" into first 4 bytes of the Data Chunk Buffer
Convert audio data size to little endian
Write the audio size (in little endian format) into the second 4 bytes of the Data Chunk Buffer
Write Cleaned and formatted Audio buffer
Return

Figure 5.10 Build Data Chunk Buffer

The Build Data Chunk Buffer sub process takes in the raw audio data and builds the byte array buffer for the Data Chunk of the wav file to be returned to the parent "Build WAV File" process. The process starts by running a sub process that will clean and format the raw audio data. This sub process will be responsible for making sure the raw data is of the correct format and Endian-ness so that I can simply be written into the Data Chunk Buffer as is. This process is too long to be included here so it is included as the sub process “Clean and Format passed in audio data”. Once we have the returned audio data that has been cleansed and formatted, the next step is to measure the size of the formatted audio data returned from the “Clean and Format passed in audio data” sub process. This measurement will have to be formatted in Little Endian Format but we will do so later. Next using the calculated size of the raw audio data, we know how large our Data Chunk Buffer will be. The size of this byte array will be the measured size of the formatted audio data plus 8 bytes for the chunkID and subChunkSize fields.

Now that we have all of the required fields for the Data Chunk of the WAV file, we can start writing those values into the Data Chunk Buffer. First we will write the chunkID, whose value will always be “data”, to the first 4 bytes of the Data Chunk Buffer. This field does not have to be formatted since it is already in the required Big Endian Format. Next we take the calculated size of the formatted audio data and convert it into a Little Endian formatted integer so that it can be written into the Data Chunk Buffer byte array. After this conversion, we then write the converted subChunkSize to the next 4 bytes of the Data Chunk Buffer. Lastly, we can write the formatted audio data we got from the “Clean and Format passed in audio data” sub process into the remaining byte space of the Data Chunk Buffer. We do not need to format here before writing since the sub process formatted the raw audio data for us. At this point the Data Chunk Buffer byte array contains the complete and correctly formatted Data Chunk and can be returned to the “Build WAV File” parent process.

5.1.5.4Build Format Chunk Buffer

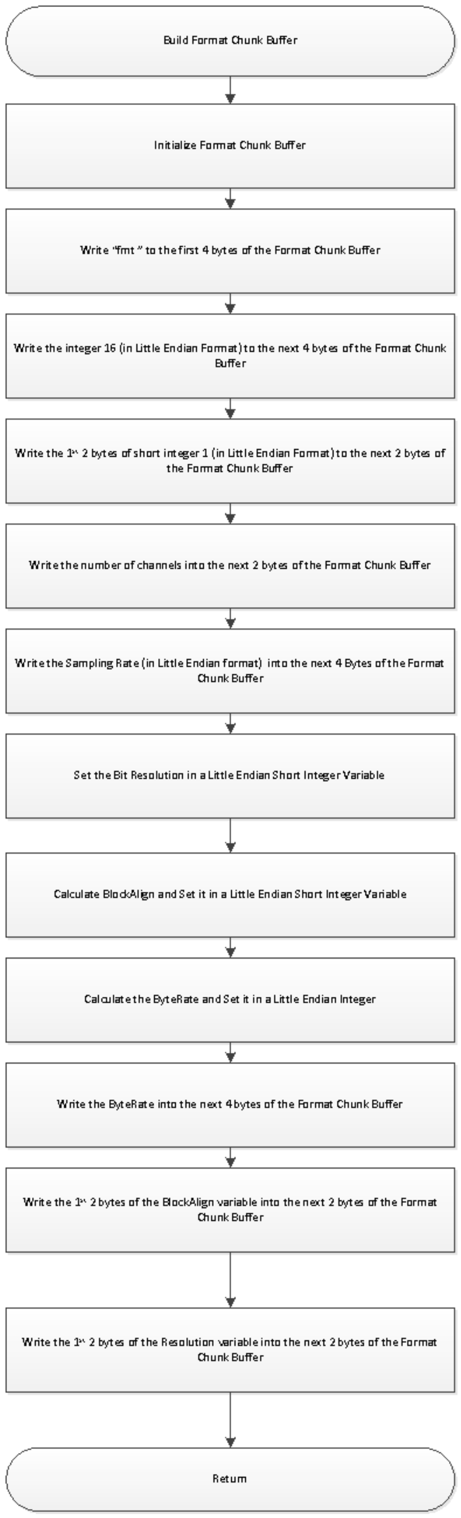

Build Format Chunk Buffer
Initialize Format Chunk Buffer
Write "fmt " to the first 4 bytes of the Format Chunk Buffer
Write the integer 16 (in Little Endian Format) to the next 4 bytes of the Format Chunk Buffer
Write the 1st 2 bytes of short integer 1 (in Little Endian Format) to the next 2 bytes of the Format Chunk Buffer
Write the number of channels into the next 2 bytes of the Format Chunk Buffer
Write the Sampling Rate (in Little Endian format) into the next 4 Bytes of the Format Chunk Buffer
Set the Bit Resolution in a Little Endian Short Integer Variable
Calculate BlockAlign and Set it in a Little Endian Short Integer Variable
Calculate the ByteRate and Set it in a Little Endian Integer
Write the ByteRate into the next 4bytes of the Format Chunk Buffer
Write the 1st 2 bytes of the BlockAlign variable into the next 2 bytes of the Format Chunk Buffer
Write the 1st 2 bytes of the Resolution variable into the next 2 bytes of the Format Chunk Buffer
Return

Figure 5.11 Build Format Chunk Buffer

The "Build Format Chunk Buffer" sub process takes in no arguments and builds the byte array buffer for the Format Chunk of the wav file to be returned to the parent “Build WAV File” process. The first task is to initialize the byte array that will be the Format Chunk Buffer. The Format Chunk of a WAV file is always the same size regardless of how large the raw data is since the fields are all of fixed length. With all of the field sizes added together the total size of the Format Chunk is 24 bytes long. Therefore, we can initialize our Format Chunk Buffer byte array to be 24 bytes in length. From here we will simply go through and write each field of the Format Chunk to the buffer making any calculations we need on the way. The first field is the chunkID which is always set to “fmt”. Keep in mind that the space character must be included to avoid any NULL bytes from being included in the middle of the WAV file.

Next we can set the subChunkSize field to be the integer value of 16 in Little Endian format. This value represents 16 bytes as opposed to the total size of the Format Chunk Buffer (24 bytes) because this field is simply there to let the program reading the file know the amount of space, in bytes, that it has to read before it gets to the Data Chunk section of the WAV file. The next field in the Format Chunk is the audioFormat field which indicates how the raw audio was sampled. Almost all Analog to Digital Converters sample using some form of Pulse Code Modulation. We indicate this by placing the decimal value of 1 in this field to select the PCM method. However, it is important to note that this field is of two bytes in length. This means since we stored this integer in a short integer in Little Endian format, we need specifically grab the first 2 bytes of the variable to write to the Format Chunk Buffer. If we write the full variable, we will exceed the byte limit of the field and corrupt the WAV file.

Next we will write the number of channels that we are using into the Format Chunk Buffer. Like the audioFormat field, the numOfChannels field is two bytes, so we must only grab the first 2 bytes of the short integer (Little Endian formatted) variable. This field’s value can either be a decimal value of 1 or 2 depending if you have a mono or stereo audio set up. Since we are grabbing the audio for the Transmission Check from the speaker pins of the radio, we will be using 2 channels and will set this field to a value of 2. Next we will set the following 4 bytes of the Format Chunk Buffer to be the sampling rate at which we sampled the audio in Hertz. This means that we will be setting this value to be 44100 represented by a Little Endian Formatted Integer as we want to sample double the frequency of any sound audible to the human ear per the Nyquest Sampling Theorem. To write the following fields, we will need to calculate them in reverse order to avoid making the same calculations multiple times to improve efficiency. We can make all of the calculations using the Little Endian Format for integers to avoid having to format each field before writing the field values. This way all of the field values are already in the correct format. We know the bitResolution by default and will set this value to bit resolution of the Analog to Digital Converter we are using. Using the bit resolution, we are able calculate the blockAlign by dividing the bitResolution by 8 and then multiplying by the number of channels we are sampling. Then lastly we can calculate the byteRate field by multiplying the sample rate number of channels and block align fields together. Now we can simply write each field to the Format

Chunk Buffer byte array in order. First we write the byteRate field to the Format Chunk Buffer as is since the field is 4 bytes in length. Then we write the first 2 bytes of both the blockAlign and bitResolution fields, respectively, since these fields are only 2 bytes in length from the Format Chunk's prospective in the WAV file. From here we have the complete byte array for the Format Chunk Buffer and can return it to the "Build WAV File" parent process.

5.1.5.5 Build Header Buffer

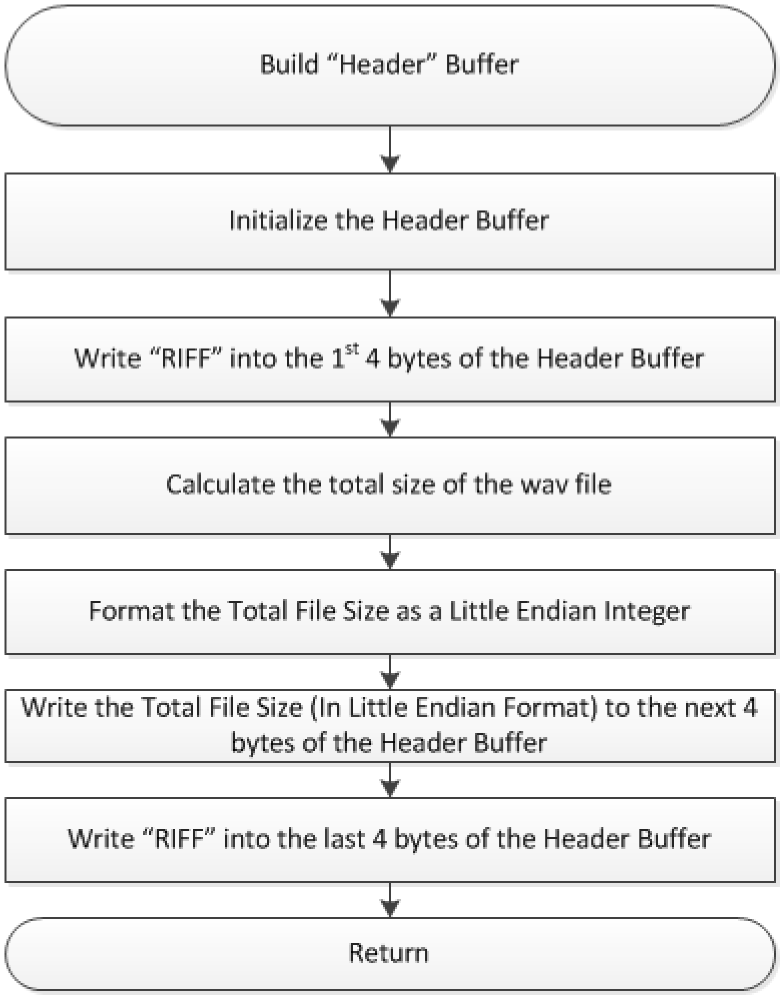

Build "Header" Buffer
Initialize the Header Buffer
Write "RIFF" into the 1st 4 bytes of the Header Buffer
Calculate the total size of the wav file
Format the Total File Size as a Little Endian Integer
Write the Total File Size (In Little Endian Format) to the next 4 bytes of the Header Buffer
Write "RIFF" into the last 4 bytes of the Header Buffer
Return

Figure 5.12 Build Header Buffer

The "Build Header Buffer" sub process takes in the Data Chunk Buffer Size and the Format Chunk Buffer size as arguments to build the byte array buffer for the Header Chunk of the WAV file to be returned to the parent "Build WAV File" process. The first task of this sub process is to initialize the Header Buffer. Since the Header Chunk of a WAV file is always the same size, we can initialize the

Header Buffer byte array to be 12 bytes in length for the three 4 byte long fields. We can start by writing the chunkID field to the buffer. As stated above, this value tells the program reading the WAV file what type of file it is reading. This field is always the string “RIFF” and can be written to the Header Buffer in Big Endian Format.

Next we calculate the total size of the WAV file which can be done by adding the passed in sizes of the Format Chunk Buffer and the Data Chunk Buffer plus 4 for the remaining riffFormat field in the Header Chunk. We do not need to include the 8 bytes from the first two fields since the purpose of the chunkSize is to tell the program reading the WAV file how many bytes it needs to read before it reaches the end of the file from the point it’s at (which is already past the first 2 fields of the Header Chunk). From here we can write the resulting chunkSize field to the second 4 bytes of the Header Buffer in Little Endian format. Lastly, we need to write the riffFormat field to the Header Buffer. This value simply tells the program reading the WAV file what type of RIFF file it is reading. For WAV files this value is always the string “WAVE” and can be written to the last 4 bytes of the Header Buffer in Big Endian format. From here the Header Buffer is contains the complete formatted contents of the Header Chunk and this sub process can return the Header Buffer byte array to the “Build WAV File” parent Process Communication with Interface Board.

5.1.6 Pin Layout and SPI

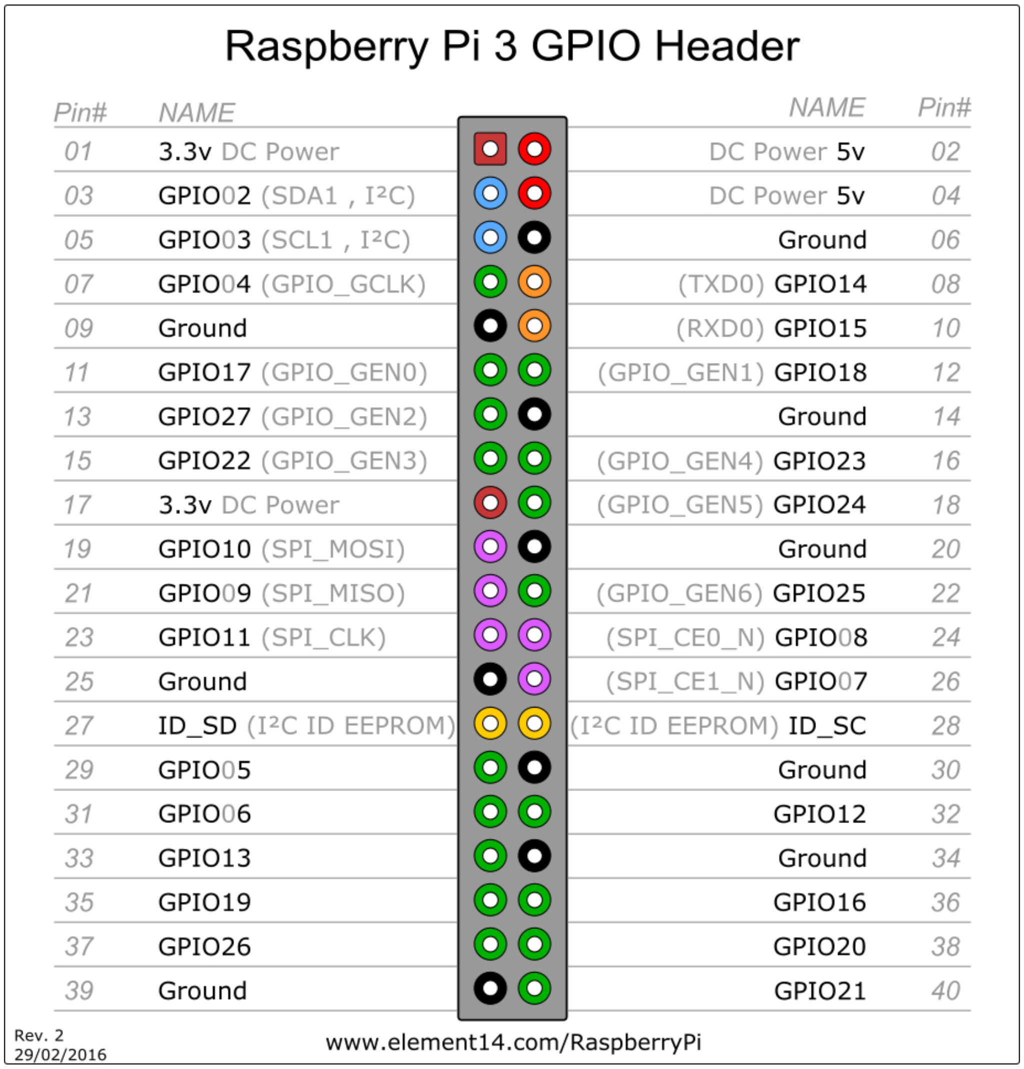

Raspberry Pi 3 GPIO Header
Pin# NAME NAME Pin#
01 3.3v DC Power | DC Power 5v 02
03 GPIO02 (SDA1 , I²C) | DC Power 5v 04
05 GPIO03 (SCL1 , I²C) | Ground 06
07 GPIO04 (GPIO_GCLK) | (TXD0) GPIO14 08
09 Ground | (RXD0) GPIO15 10
11 GPIO17 (GPIO_GEN0) | (GPIO_GEN1) GPIO18 12
13 GPIO27 (GPIO_GEN2) | Ground 14
15 GPIO22 (GPIO_GEN3) | (GPIO_GEN4) GPIO23 16
17 3.3v DC Power | (GPIO_GEN5) GPIO24 18
19 GPIO10 (SPI_MOSI) | Ground 20
21 GPIO09 (SPI_MISO) | (GPIO_GEN6) GPIO25 22
23 GPIO11 (SPI_CLK) | (SPI_CE0_N) GPIO08 24
25 Ground | (SPI_CE1_N) GPIO07 26
27 ID_SD (I²C ID EEPROM) | (I²C ID EEPROM) ID_SC 28
29 GPIO05 | Ground 30
31 GPIO06 | GPIO12 32
33 GPIO13 | Ground 34
35 GPIO19 | GPIO16 36
37 GPIO26 | GPIO20 38
39 Ground | GPIO21 40
Rev. 2
29/02/2016
www.element14.com/RaspberryPi

Figure 5.13 Raspberry Pi 3 GPIO Header Layout

Interface Board	Physical Pin #	GPIO Logical Pin #
TTLPTT	32	12
TTLCD	36	16
SPI (MISO)	21	9
SPI (MOSI)	19	10
SPI (SCLK)	23	11
SPI (CE)	24	8
Pulse Counter	38	20
TX Audio	3.5mm Audio Jack	-

Above is an image of all 40 of the GPIO pins available on the Raspberry Pi 3 showing their physical pin number as well as their software available GPIO pin number. Above you will also find a table showing which connection from the interface board is going to what pin on the Raspberry Pi. It is really important to note that not all of these available pins are general purpose pins and have specific purposes and all GPIO pins both receive and transmit digital signals. This means that these particular pins cannot be used; a good example of this includes the various ground pins located on the GPIO pinout. Concerning all of the inputs and outputs between the Raspberry Pi's GPIO pins and the Interface board, there 5 connections to mention: TTL PTT (GPIO pin 12), TTL CD (GPIO pin 16), SPI Bus (hardware spi), Pulse counter (GPIO pin 20), and TX Audio Out (Onboard Audio Jack).

The TTL PTT signal will be connected from the Raspberry Pi 3's GPIO pin number 12 to the interface board. This connection will be active high, meaning that when the software wishes to broadcast a signal, it will set this pin high letting the interface board know to send the appropriate signals to the radio. Secondly is the TTL CD which will connect the Interface board to Logic GPIO pin 16 on the Raspberry Pi's external pins. This pin will be active high, and upon the pin going into a high state, the software will be notified that there is a signal being detected by the radio. This pin will be utilized to select certain function options, such as WX transmit and TX Check, as well as let the software know when to start reading in audio data from the SPI bus. The SPI bus, or Serial Peripheral Interface bus, is a communication method implemented on many proprietary chips like the MCP3008 Analog to Digital Converter we are using. This communication protocol involves 4 main connections between the two components that are transferring and receiving data. While most of the connections run straight from the MCP3008 ADC on the interface board to the Raspberry Pi, the main data line coming from the MCP3008 to the Raspberry Pi must run through a voltage divider since the MCP3008 is running on a 5V logic digital level and the Raspberry Pi is running on a 3.3V digital logic level; so from here on out I will refer to the connections as being in between the interface board and the Raspberry Pi's hardware SPI pins.

The 4 connections for the SPI bus include Master in Slave Out (MISO), Master Out Slave In (MOSI), The Clock Line (SCLK), and the Chip Enable Line (CE). All of these connections are digital and active high with the exception of the CE signal which is a digital active low signal. In this architecture we will refer to the Raspberry Pi as the "Master" and the Interface Board as the "Slave", due to the fact that the Raspberry Pi will be driving the actions of reading and writing data while the Interface Board will simply be providing and receiving data on command. All SPI connections will be connected to the hardware SPI pins as shown in the table above. The MISO line will be for reading in data from the Interface board that has been Analog to Digitally converted. The MOSI line will be for the Raspberry Pi to send data to the Interface board essentially telling the ADC which channel to read and what data to send back. The SCLK line is to keep the clocks of the Raspberry Pi and the ADC located on the Interface board synchronized. Lastly the CE line will be set to low whenever the Raspberry Pi wants to read from that chip. Looking at the Pulse Counter connection, this connection is coming from the Interface board and will be set to High in pulses as the wind direction moves. The Raspberry

Pi will monitor this pi and keep track of how many pulses it gets, helping it determine the wind speed coming from the sensor. Lastly we have the TX Audio which will be coming from the Raspberry Pi to the Interface Board from the Audio Jack mounted onto the Raspberry Pi. As the name suggests this will be the signal, that when TTL PTT is set to a digital High, will be sent to the Interface board to be broadcast out over the radio channel.

5.1.7 MCP3008

5.1.7.1 Background Information

For the Analog to Digital Converter we chose to use the MCP3008. This particular ADC as chosen due to the fact that the software libraries and interfaces for this chip were already open-sourced by Adafruit Industries. This made reading from the chip easier and allowed us to focus more on what we were going to do with the information provided to us as opposed to spending time writing code to get the information we needed. Secondly we chose this chip from a software standpoint because we can use MCP3008 to read all of our analog signals as opposed to just a group of them. This versatility provides for a lot of code reuse as instead of having to read data from multiple sources we can read all of our sensor data from one chip.

It is important to note that the PCM1862 was a viable option for reading the Audio Data specifically as we were not sure whether or not that reading our audio data from the same ADC as other data would affect the quality of the audio read in. The PCM1862 is a dedicated audio ADC that has built in filters and Automatic Gain Control. However, we concluded that having a separate, dedicated Audio ADC was not needed as we were already using the interface board to filter and add gain to the signal. Also the other analog signals that the MCP3008 would be reading are sampled at extremely low sampling rates. This means that we do not have to worry about the other components lowing the MCP3008's total sample rate to a low enough rate in which audio could not be affectively read. Our final conclusion was that the MCP3008 would be accurate and fast enough to sample and convert all of the analog signals coming from our external sensors as well as audio data.

5.1.7.2 Analog to Digital Conversion

Since we will be get values from the MCP3008 that have been converted from an audio signal that has been amplified and offset by 3V, for the software it will be important to take note of the relationship between the input analog voltage and the outputted digital signal. Taking a look at the MCP3008's datasheet, we are provided an estimated equation taking in the voltage level of the received analog input and outputting the expected digital value from the chip. For our system we are operating the chip at a V_{ref} of 5V. The equation is provided below. Using this equation from the datasheet, it was important to compare the values expected to the actual values we were getting from the system. Below is a table that provides the sampled voltage level from an inputted analog wave and outputs the 10-bit Value read from the software sampling the ADC at 8 KHz. The third column is the output of the equation provided by the MCP3008's datasheet using the 5V Vref and the Voltage level from column one as inputs.

$$Digital\ Output\ Code = \frac{1024 \times V_{IN}}{V_{REF}}$$

Where:

V_{IN} = analog input voltage
V_{REF} = reference voltage

Voltage Level	10-bit Value	Expected Value
4.99	1023	1022
4.97	1019	1018
4.96	1017	1016
4.95	1015	1014
4.94	1013	1012
4.93	1011	1010
4.92	1010	1008
4.91	1008	1006
4.89	1003	1002
4.84	990	992
4.8	978	984
4.73	954	969
4.71	945	965
4.64	919	951
4.59	902	941
4.54	886	930
4.5	875	922
4.44	856	910
4.4	842	902
4.34	824	889
4.29	808	879
4.2	784	861
4.15	768	850
4.1	756	840
4	731	820
3.96	720	812
3.9	705	799
3.84	692	787
3.81	686	781
3.74	671	766
3.7	661	758
3.66	654	750
3.6	639	738
3.55	629	728
3.5	619	717

Voltage Level	10-bit Value	Expected Value
3.44	608	705
3.4	600	697
3.34	586	685
3.29	578	674
3.25	569	666
3.19	559	654
3.14	548	644
3.1	541	635
3.06	533	627
3	522	615
2.94	511	603
2.9	504	594
2.85	494	584
2.8	486	574
2.75	478	564
2.69	468	551
2.65	460	543
2.6	452	533
2.55	442	523
2.5	433	513
2.45	425	502
2.39	415	490
2.35	407	482
2.29	399	469
2.25	391	461
2.2	382	451
2.15	375	441
2.09	364	429
2.05	359	420
1.99	347	408
1.95	341	400
1.9	331	390
1.85	323	379
1.8	315	369
1.75	308	359

1.7	300	349
1.64	291	336
1.6	282	328
1.53	272	314
1.5	266	308
1.45	259	297
1.39	248	285
1.34	240	275
1.29	231	265
1.25	225	257
1.19	216	244
1.14	207	234
1.1	200	226
1.05	192	216
1	183	205
0.95	176	195
0.9	168	185
0.85	158	175
0.79	149	162
0.74	141	152
0.7	133	144
0.64	124	132
0.6	116	123
0.55	108	113
0.5	98	103
0.45	90	93
0.4	81	82
0.35	72	72
0.29	61	60
0.25	53	52
0.2	43	41
0.15	33	31
0.09	20	19
0.05	12	11
0.045	10	10
0.039	8	8
0.029	6	6
0.021	4	5
0.016	3	4
0.003	1	1
0	0	0

To better visualize the relationship between the expected values from the equation and the actual values that we are reading, we graphed the data from the table above.

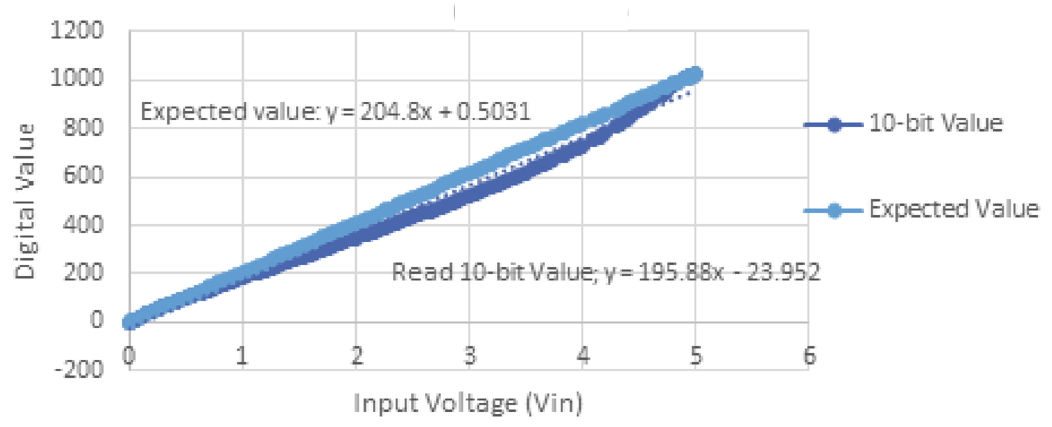


Figure 5.14 Relationship between Input Voltage and 10-bit Value with Expected

After graphing the relationships of both the Voltage Level to Read 10-bit Digital value and the Voltage level to the Expected digital value from the equation provided by the datasheet, it is important to note that there is some slight deviation from the linear behavior the equation provided by the datasheet expects. Taking a look at the graphed chart above (generated from the table above), you can see that the read 10 bit digital values are relatively linear, but express some non–linear behavior between 1V and 4.5V. In this range the reading we took began to deviate slightly from the expected values generated by the equation from the datasheet. The non-linearity here may be due to the operation system driving the sampling of the chip. Most ADCs are run on microcontrollers that run precompiled and fast code, where in this case the operating system is in control of almost every part of the system. That being said, the non-linearity is within an acceptable range for the data we are sampling and we will proceed with using the MCP3008.

5.1.7.3 MCP3008 Communication Logic

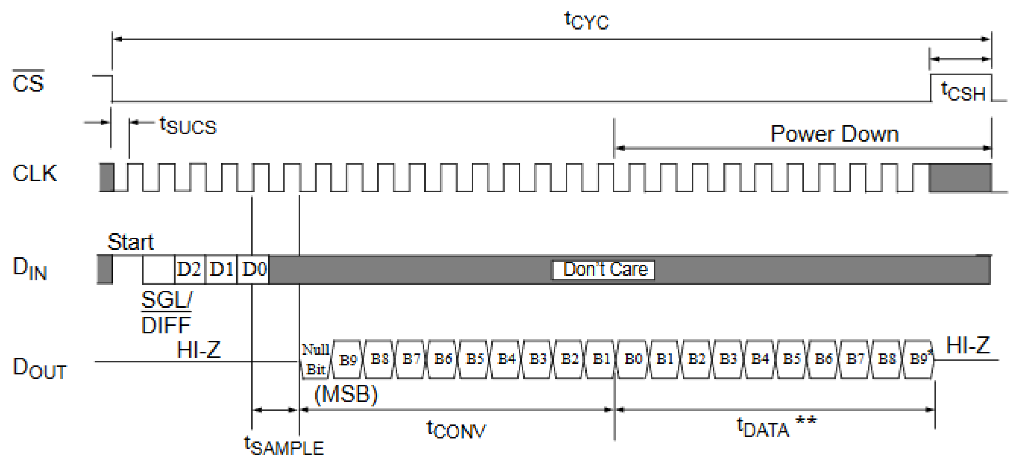
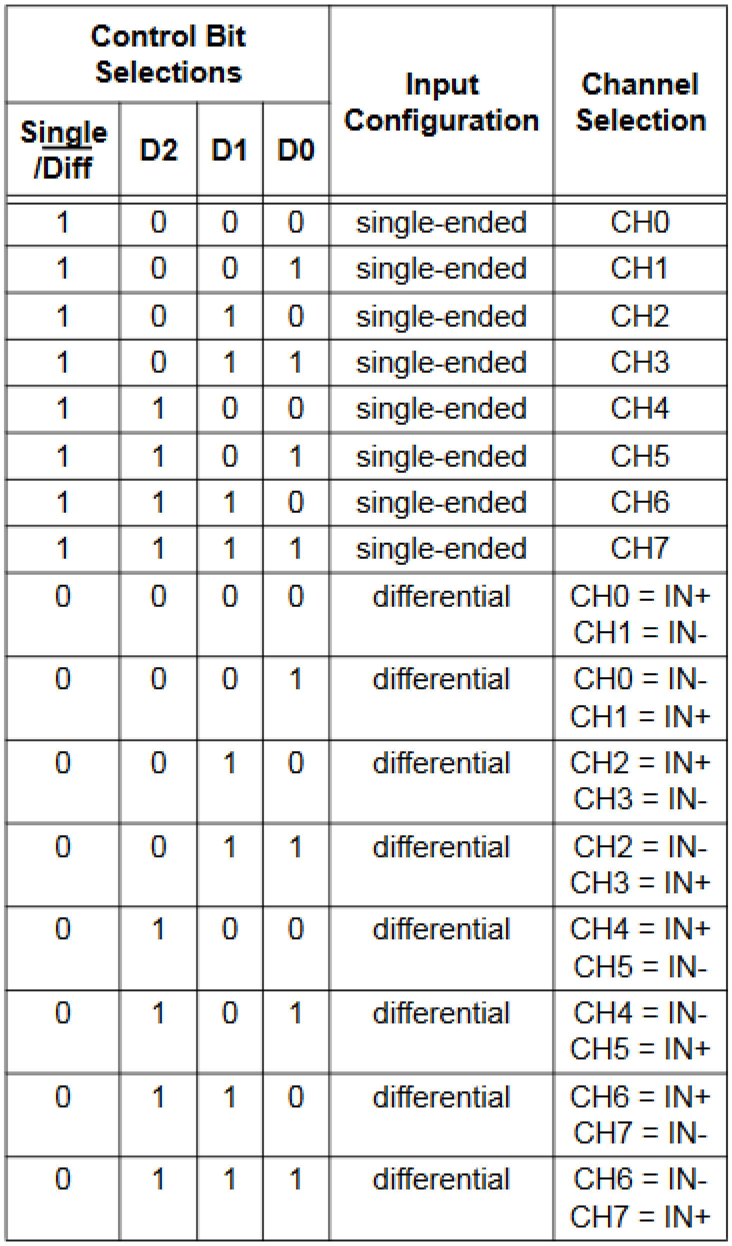


Figure 5.15 MCP3008 Logic

Communication between the Raspberry Pi and the MCP3008 Analog to Digital converter happens over the hardware SPI interface between the two. However, for the Raspberry Pi to read from the MCP3008 there is a certain protocol in which to communicate that read action. While this communication process will be handled by the software using the Adafruit_MCP_3008 python library, the process in which this is depicted by the image above taken from MCP3008's datasheet. The first step is for the Raspberry Pi to let the MCP3008 know it will be selecting a function from that chip; it does this by setting the CE pin low. From here the Raspberry Pi will send a series of configuration bits over the SPI_MOSI line. These configuration bits and their functions are shown in the following table.



Control Bit Selections				Input Configuration	Channel Selection
Single /Diff	D2	D1	D0		
1	0	0	0	single-ended	CH0
1	0	0	1	single-ended	CH1
1	0	1	0	single-ended	CH2
1	0	1	1	single-ended	CH3
1	1	0	0	single-ended	CH4
1	1	0	1	single-ended	CH5
1	1	1	0	single-ended	CH6
1	1	1	1	single-ended	CH7
0	0	0	0	differential	CH0 = IN+ CH1 = IN-
0	0	0	1	differential	CH0 = IN- CH1 = IN+
0	0	1	0	differential	CH2 = IN+ CH3 = IN-
0	0	1	1	differential	CH2 = IN- CH3 = IN+
0	1	0	0	differential	CH4 = IN+ CH5 = IN-
0	1	0	1	differential	CH4 = IN- CH5 = IN+
0	1	1	0	differential	CH6 = IN+ CH7 = IN-
0	1	1	1	differential	CH6 = IN- CH7 = IN+

Figure 5.16 MCP3008 Channels

When the MCP3008 receives these configuration bits, it will know what function to perform and returns the necessary data back over the SPI_MISO line prefaced by a Null bit. This Null bit lets the Raspberry Pi know it should begin receiving data. For example, if the Raspberry Pi sent over the bit sequence 1000 out over the SPI_MOSI line, the MCP3008 would read the Analog signal from channel 0, convert it to a digital value, and send that digital number back over the SPI_MISO line prefaced by a Null bit. After the data has finished transmitting, the Raspberry Pi will reset the SPI_CE pin back to High so that the chip will not receive any unintended signals from the Raspberry Pi.

5.2 Configuration Screen

This section will be added in fall for Senior Design II

5.3 Web Server

This section will be added in fall for Senior Design II

6 Integration and Prototype

6.1 Interface Board Breadboarding

For breadboarding and testing purposes the operational amplifiers used were TL084s due to ease of access. But for the practical application of this circuits the operational amplifiers used will be LM324s. This is because in the final PCB the interface board will have no negative DC power sources. The behaviors of both operational amplifiers under the current application is similar enough to the point where their performance is interchangeable if power sources were not to be considered.

6.1.1PTT Circuit

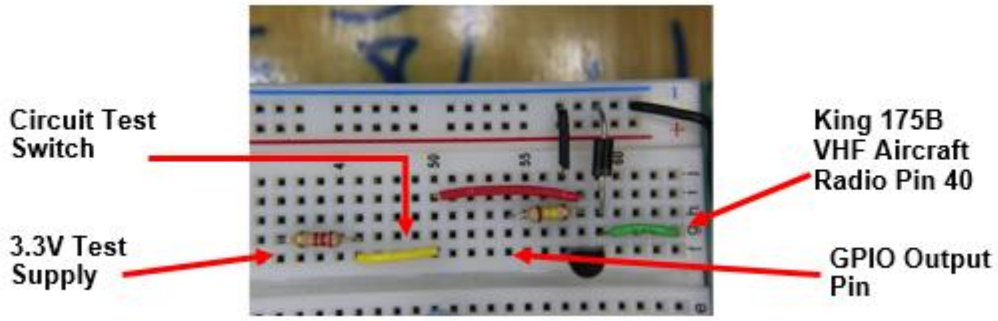


Figure 6.1 PTT Circuit Breadboard

6.1.2RX Audio Circuit

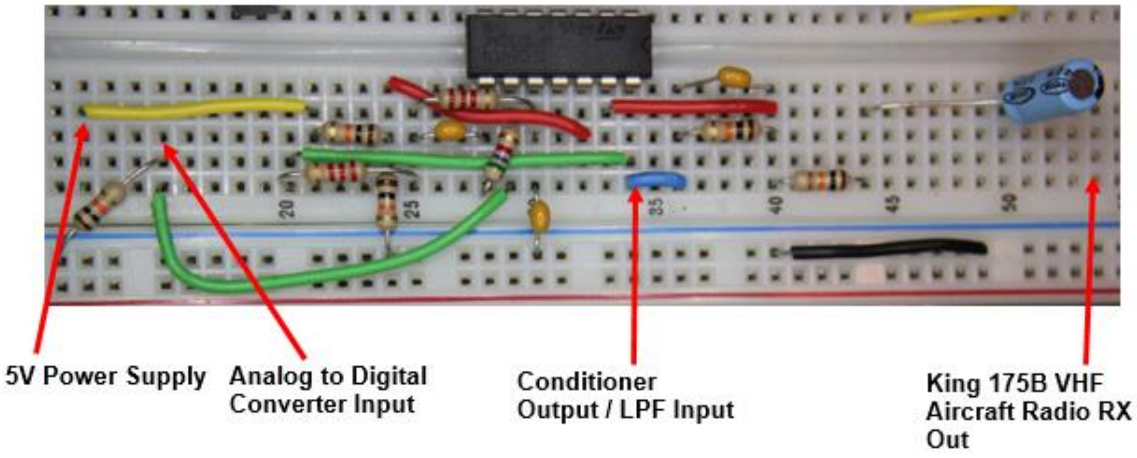

5V Power Supply
Analog to Digital Converter Input
Conditioner Output / LPF Input
King 175B VHF Aircraft Radio RX Out

Figure 6.2 RX Audio Circuit Breadboard

6.1.3TX Audio Circuit

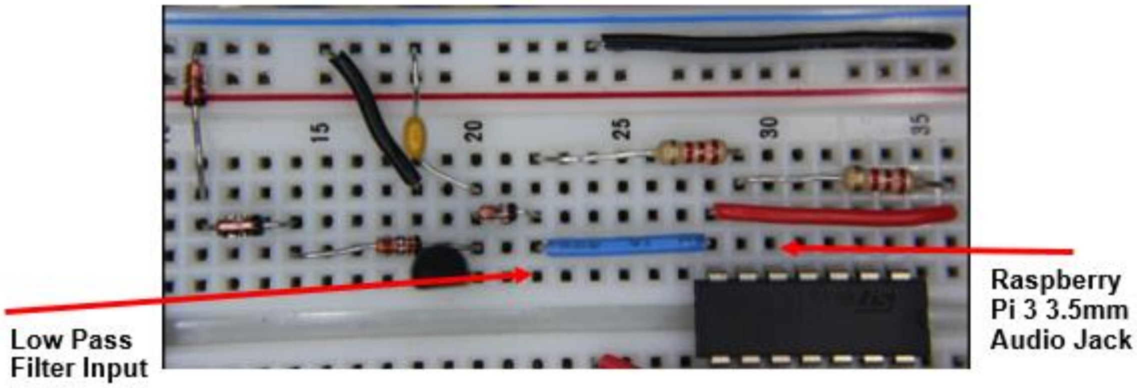


Figure 6.3 TX Audio Compressor Circuit Breadboard

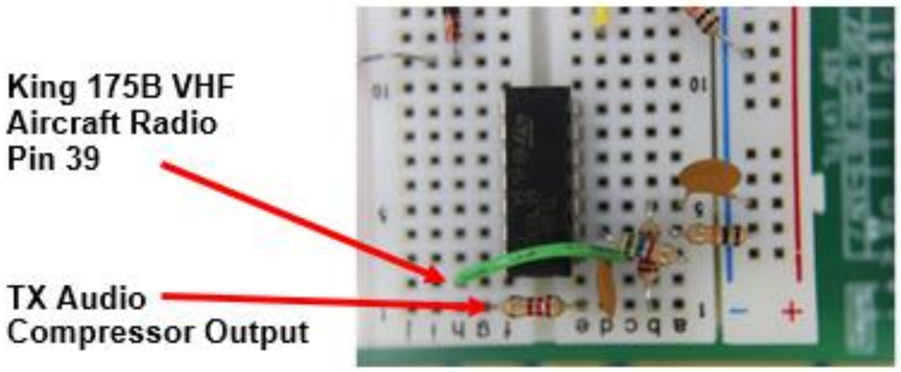

King 175B VHF Aircraft Radio Pin 39
TX Audio Compressor Output

Figure 6.4 TX Audio Low Pass Filter Circuit Breadboard

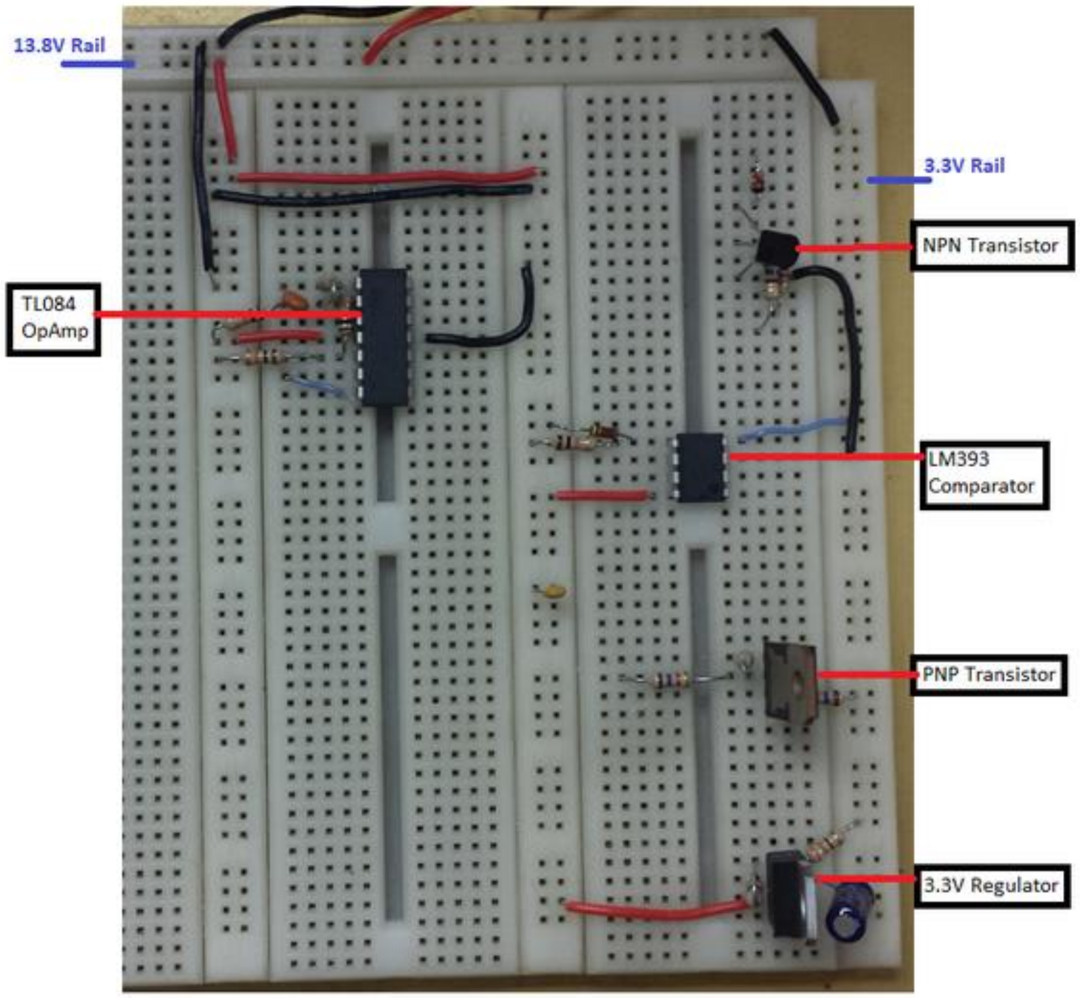


Figure 6.5 Section of Overall Breadboard

For prototyping, several 3.3V regulators were tested. The first one to be tested was the LM317. The input to the regulator is 13.8V, this is also connected to a .1uF decoupling capacitor. This capacitor adds fast charge storage, commonly used when connected to a power supply which are relatively slow. The second pin on the regulator is connected to 300ohm resistor and a 240ohm resistor. The value of 300ohm was found by connecting a potentiometer to the second pin and adjusting it until 3.3V were reached. Once the output was set and constant, the potentiometer was removed and its set value was measured so it could be substituted with a single value resistor. The output of the regulator is also connected to a bypass or decoupling 1uF capacitor that serves the same purpose of adding fast charge storage. Below is the schematic for the 3.3V regulator setup.

6.1.4 3.3V Regulator Circuit

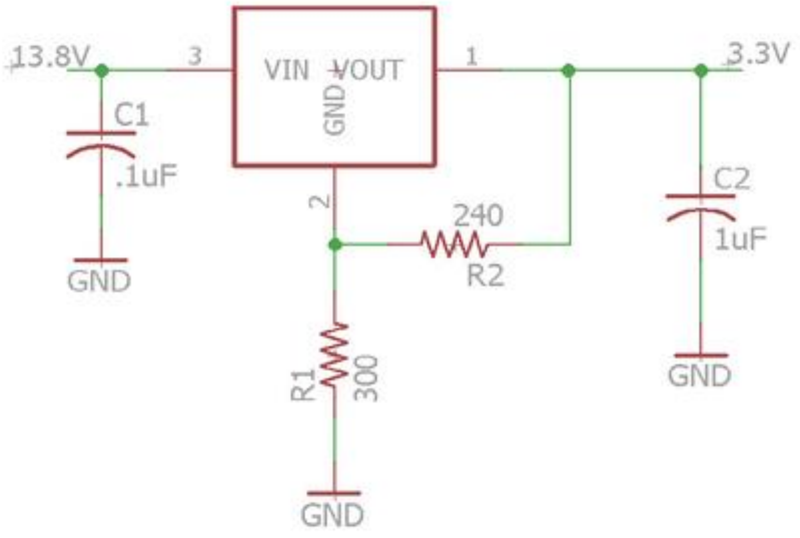


Figure 6.6

As mentioned this circuit was used for prototyping. The 3.3V output was used for the logical output of the comparator. Although there is a second use of 3.3V on our system, to the raspberry pi, this regulator is not able to provide the necessary current to power the microcontroller. The picture below shows the regulator connected to the resistors and capacitor.

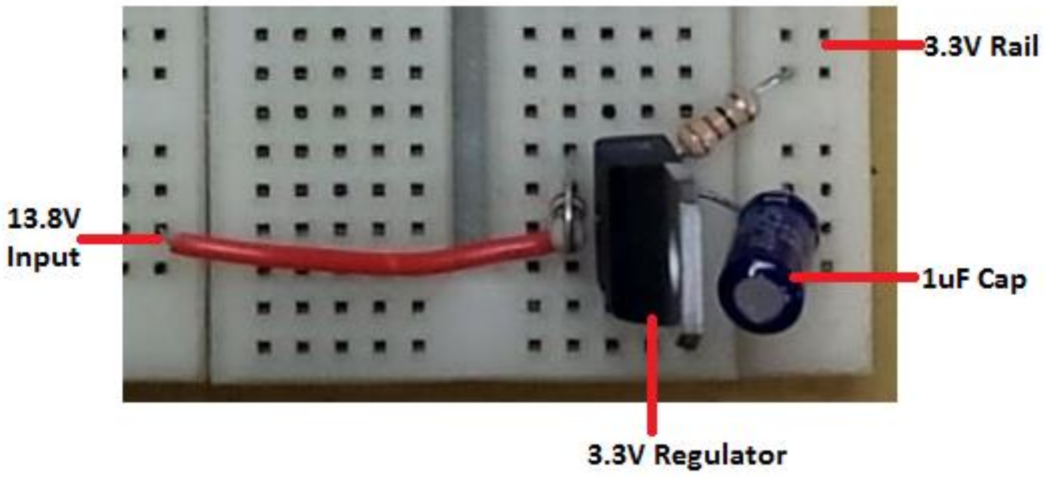


Figure 6.7 3.3V Regulator Circuit Breadboard

The image below shows the output of the 3.3V regulator along with the input connected to the oscilloscope. The input is 13.8V and out 3.3V.

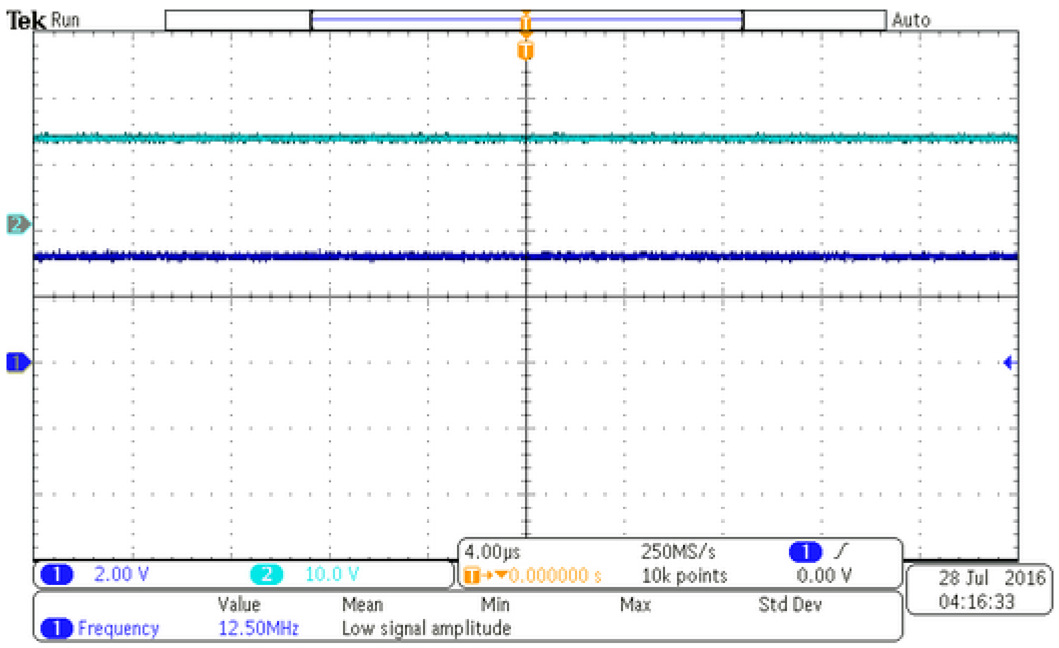


Figure 6.8

6.1.5 Carrier Detect Comparator Circuit

Another segment of the prototype is the comparator circuit. The 8.5k ohm and 10k ohm resistors can be seen at the left of the picture where the 13.8V input comes from the supply and it's then divided. Note that on the image, pin 1 of the comparator is at the bottom right.

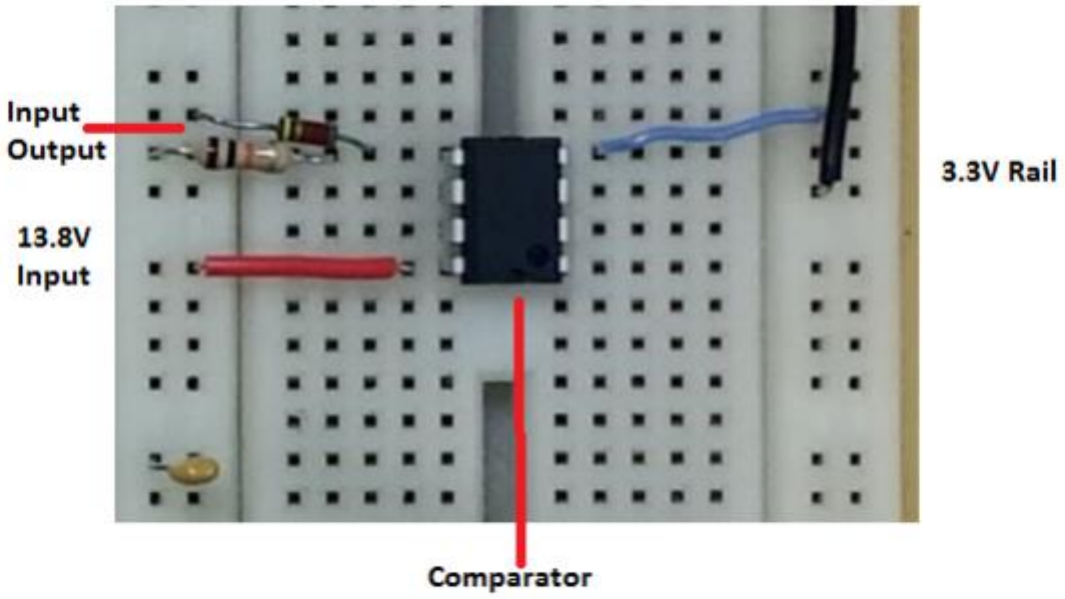


Figure 6.9 Comparator Circuit Breadboard

6.1.6 AGC Conditioner Circuit

Below is the TL084 op amp used for the AGC. The input power rails can be seen on the left and right of the image and can also be seen on the main image of the breadboard. The input during the prototype testing was 13.8V. The voltage for operation that will be given to the op amp in the final design will be 12V from the 12V regulator. The op amp can provide with up to 4 stages to use. The AGC circuit, as previously discussed, uses the first two stages which can be observed on the left side of the op amp.

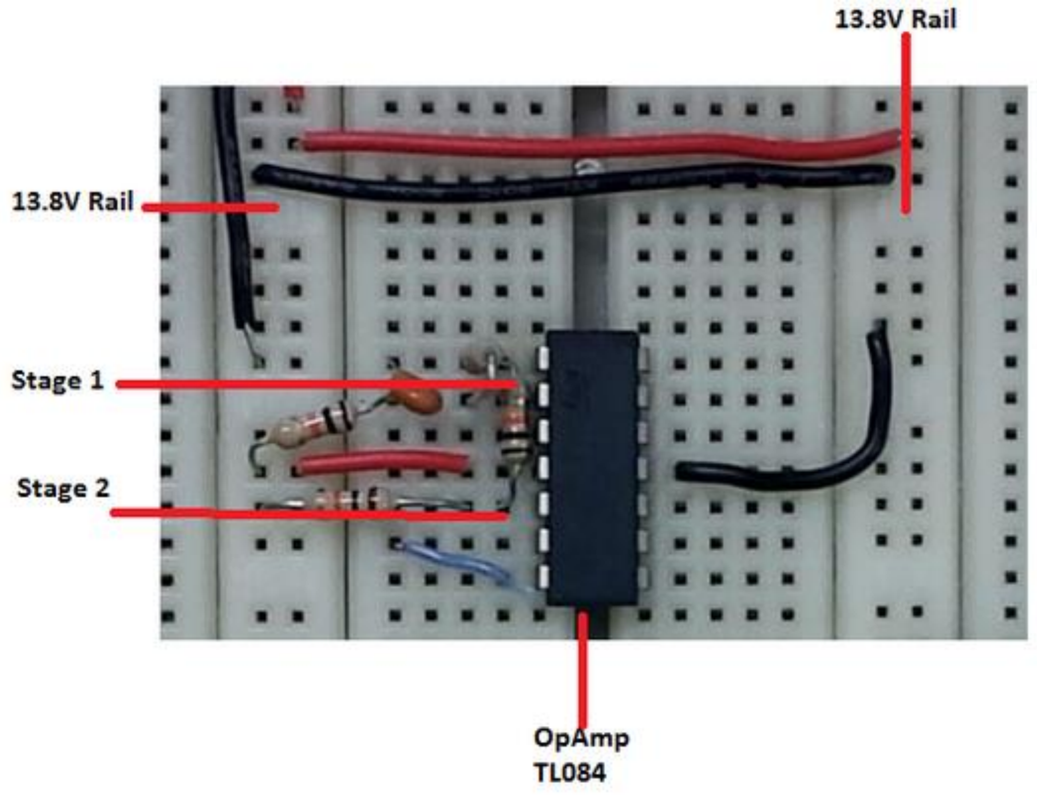


Figure 6.10 AGC Conditioner Circuit Breadboard

The PNP transistor used for prototyping was the TIP42C. This is a power transistor, this will suffice our use for testing and seeing the change in wind speed but it will not be quick enough for the final system. A switching transistor will be used in the final system. This transistor was used for testing only like the 3.3V regulator. Below is the setup on the breadboard as seen on the schematic of the overall system. The 3.3V rail is given by the 3.3V regulator and used when the wind speed input (left) is connected to ground. Overall acting as a switch for the 3.3V source according to the wind speed. This 3.3V output is then sent to the raspberry pi through a general input/output pin.

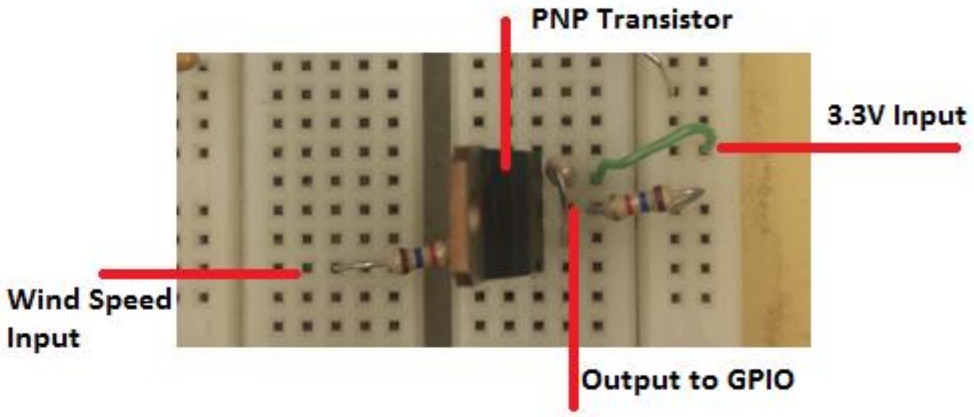


Figure 6.11

The image below continues to show the output and input of and to the PNP transistor to measure the wind speed. The input and output were connected to the oscilloscope and displayed below. Channel 1 output and channel 2 input. It's important to note that when there is wind speed being measured the output of the transistor becomes the square wave output with a magnitude of 3.3V as seen on the image.

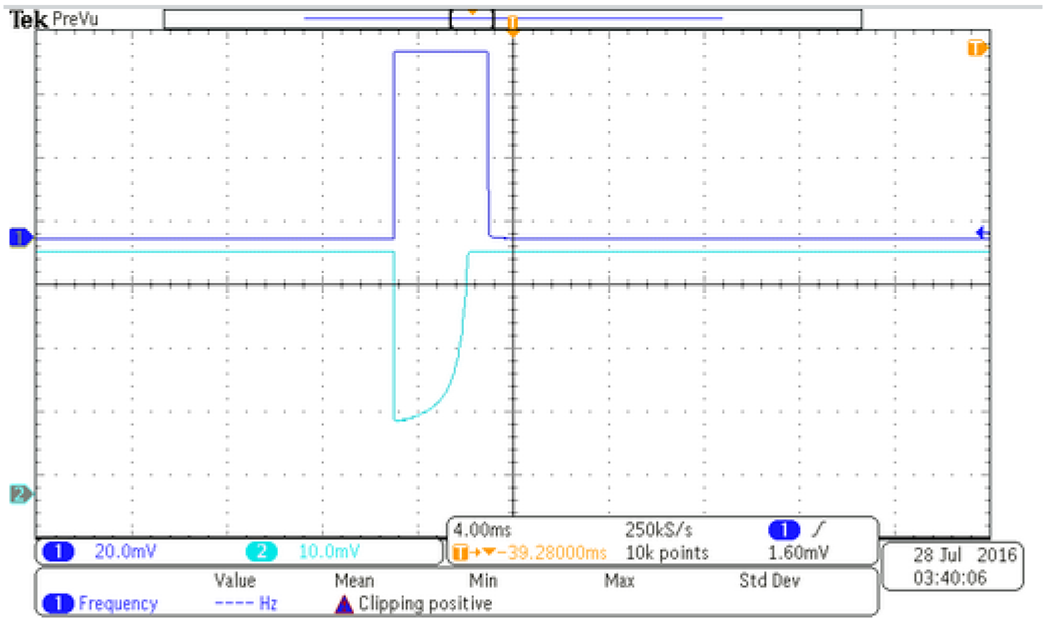


Figure 6.12

Added to the list of components tested is the 5V regulator. This is the regulator used when measuring the wind direction. This 5V input is connected to the wind vane and ground, the output of the wind vane is sent to the op amp and then to the analog and digital converter. The breadboard circuit below shows the input and output along with the setup of the regulator. A .1pF capacitor was added at the output to ground to prevent oscillations.

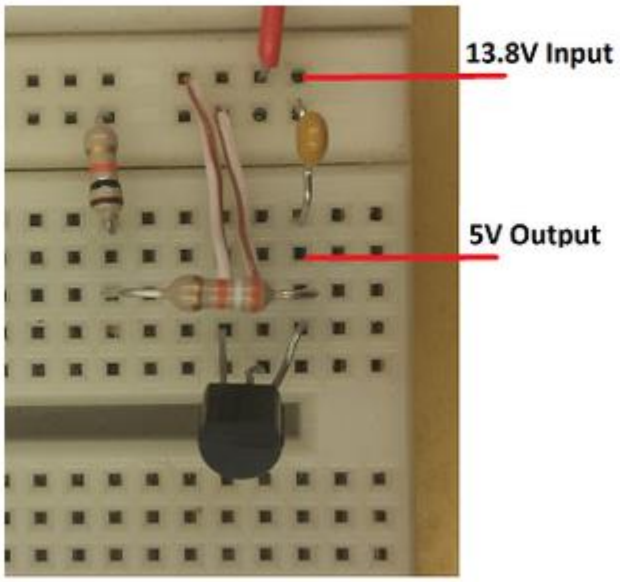


Figure 6.13

The image below shows the output of the 5V regulator along with the input connected to the oscilloscope. Two channels were used and can be seen in this image. The input of 13.8V and the of output 5V.

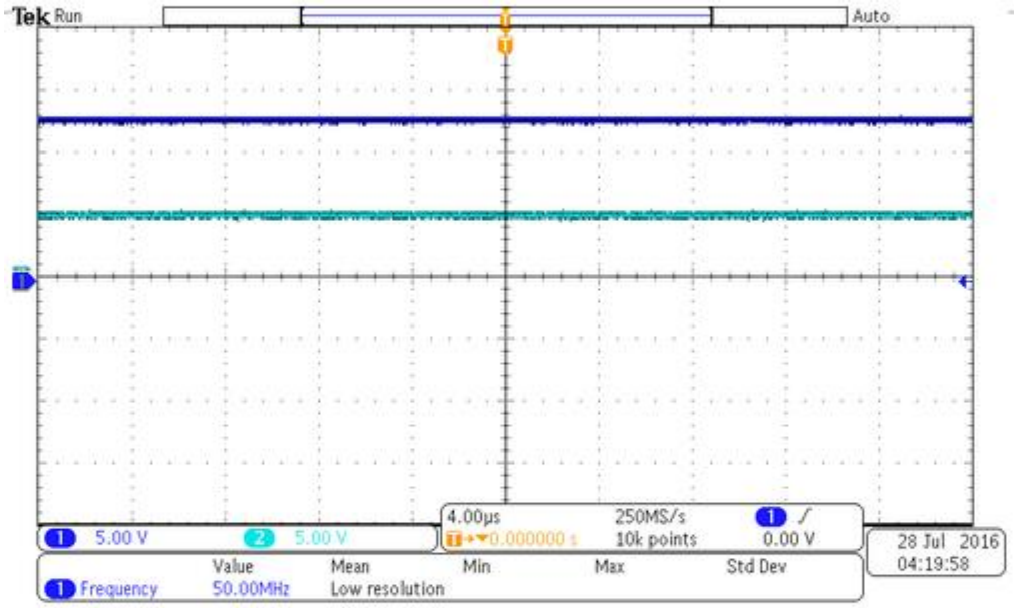


Figure 6.14

The last voltage regulator used in our system is the 12V regulator. The output of this regulator is used to power the positive input of the TL084 operational amplifier. The image below shows the setup used on the breadboard for testing purposes.

Input to device of 13.8V and output of 12V. A capacitor was added to the output and ground to prevent oscillations.

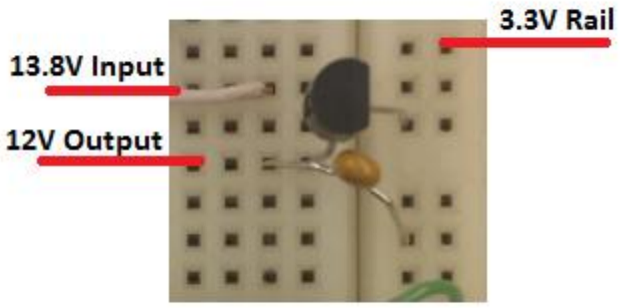


Figure 6.15

The image below shows the output of the 12V regulator along with the input connected to the oscilloscope. Two channels were used and can be seen in this image. The input of 13.8V and the of output 12V. If compared to the 5V and 3.3V regulators it is clear to see the constant but different outputs of each one of them sharing a constant input.

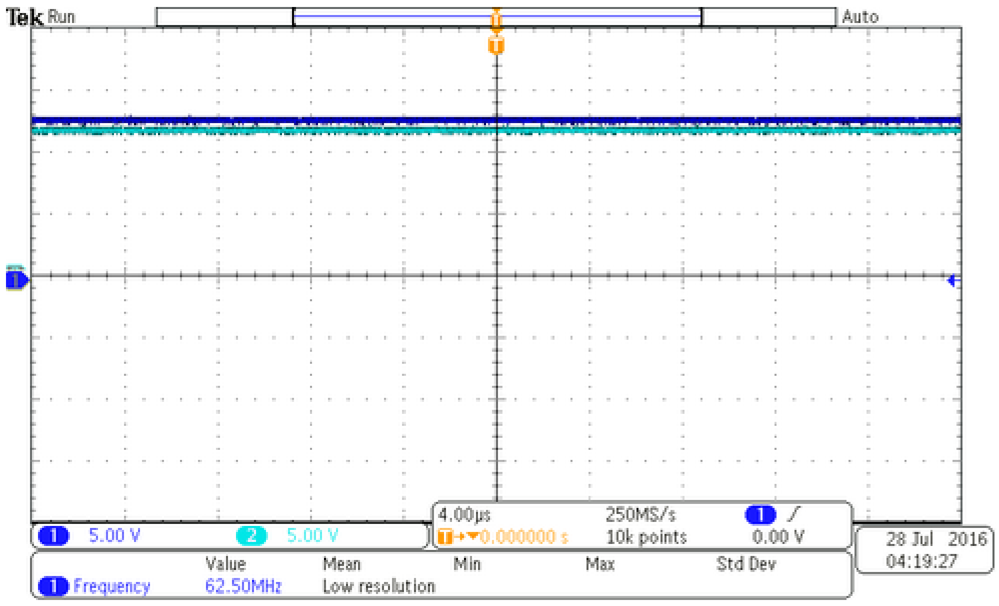
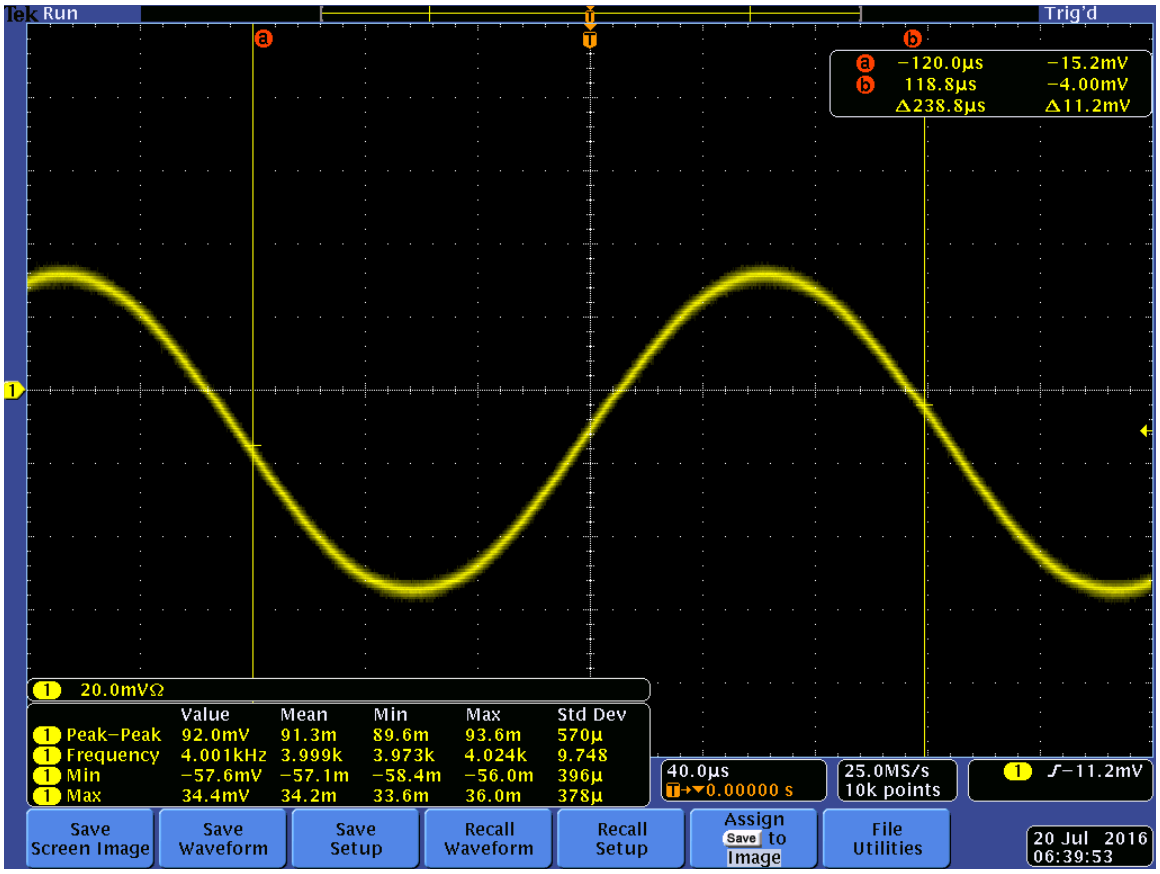


Figure 6.16

6.2 Signal Conditioning Testing

6.2.1RX Audio Circuit

Testing the RX Audio signal circuit, the input to it was a simulation of the output signal of the radio. After extensive testing with the KX-170B VHF Aircraft radio and various input made into pin 39. The received signal by our oscilloscope was a maximum signal of 320mV peak to peak with some offset. This signal was then pushed through the circuit and measured at various points to ensure the circuit was behaving was designed

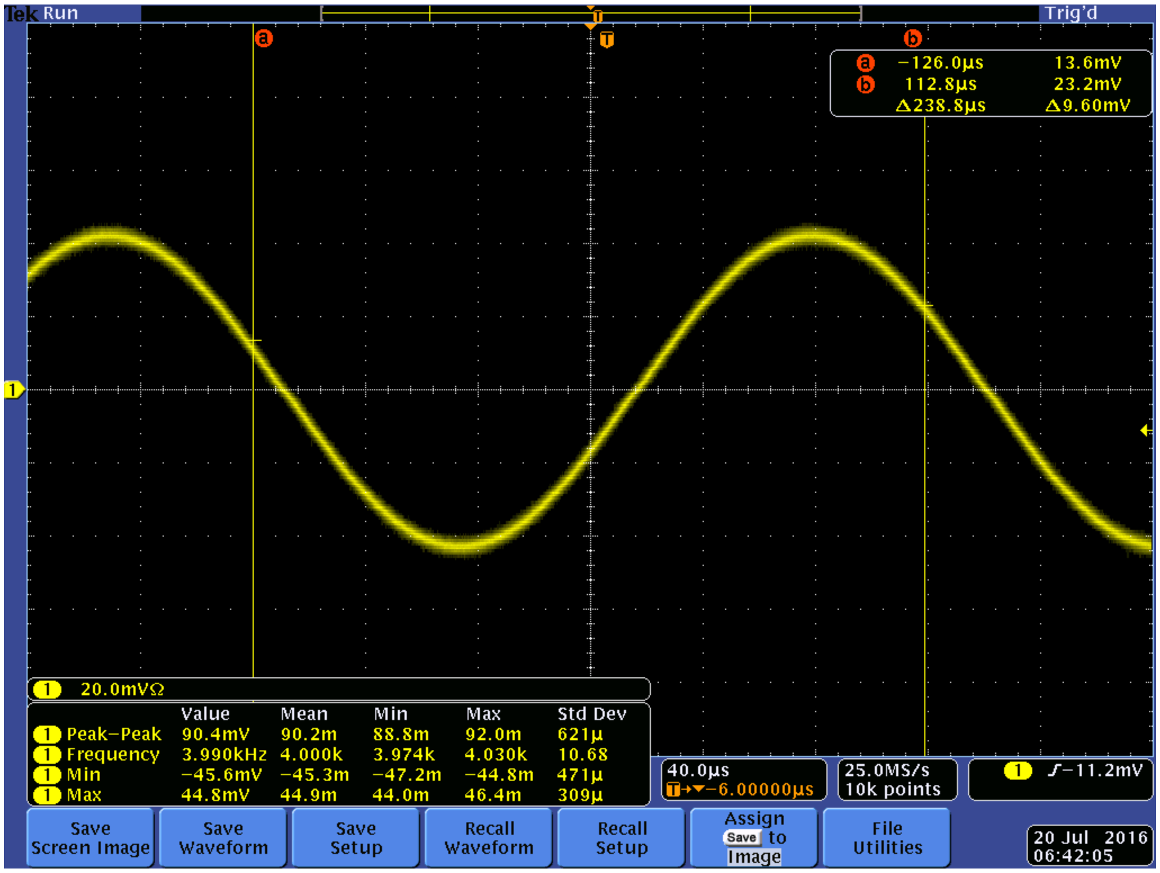


a	−120.0µs	−15.2mV
b	118.8µs	−4.00mV
	Δ238.8µs	Δ11.2mV

	Value	Mean	Min	Max	Std Dev
1 Peak−Peak	92.0mV	91.3m	89.6m	93.6m	570µ
1 Frequency	4.001kHz	3.999k	3.973k	4.024k	9.748
1 Min	−57.6mV	−57.1m	−58.4m	−56.0m	396µ
1 Max	34.4mV	34.2m	33.6m	36.0m	378µ

Figure 6.17 RX Audio Input

With the 320mV peak to peak signal being passed through a polarized 1µF capacitor the 80mV DC offset was removed.



	a	b	Δ
Time	−126.0µs	112.8µs	Δ238.8µs
Voltage	13.6mV	23.2mV	Δ9.60mV

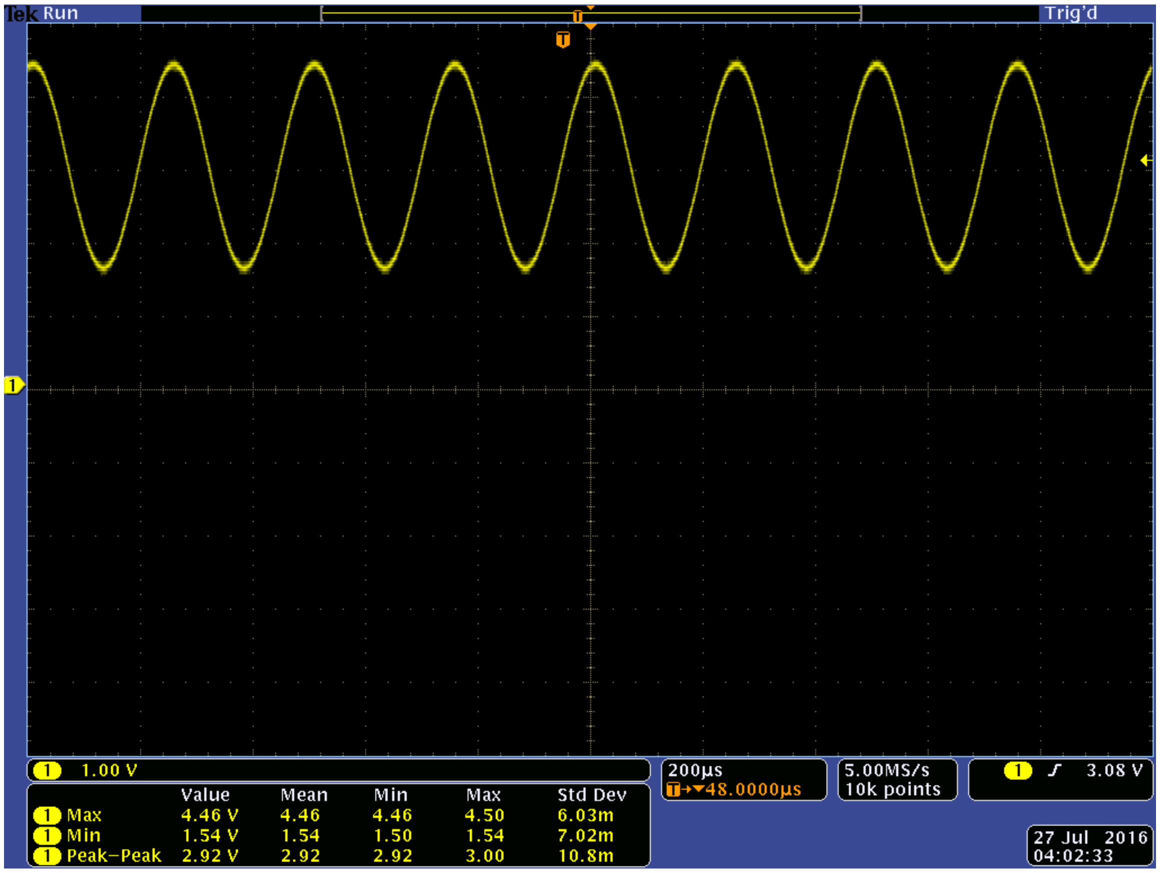
1 20.0mVΩ

	Value	Mean	Min	Max	Std Dev
1 Peak−Peak	90.4mV	90.2m	88.8m	92.0m	621µ
1 Frequency	3.990kHz	4.000k	3.974k	4.030k	10.68
1 Min	−45.6mV	−45.3m	−47.2m	−44.8m	471µ
1 Max	44.8mV	44.9m	44.0m	46.4m	309µ

40.0µs
T→▼−6.00000µs
25.0MS/s
10k points
1 ʃ−11.2mV
20 Jul 2016
06:42:05

Figure 6.18 RX Audio Input with No Offset

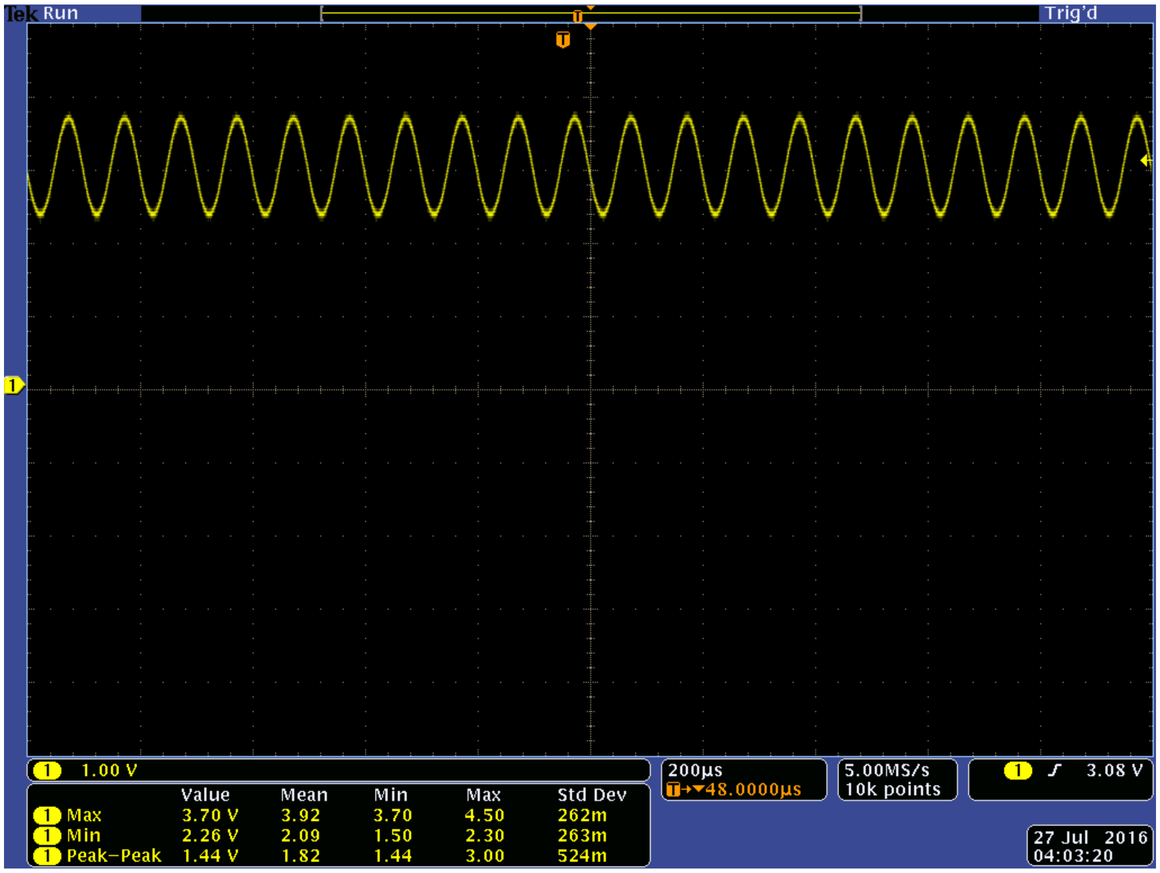
Passing the signal through a conditioning circuit for the analog to digital converter, the signal is again offset with a 2.5V reference voltage provided by the 5V power supply in the interface board.



	Value	Mean	Min	Max	Std Dev
1 Max	4.46 V	4.46	4.46	4.50	6.03m
1 Min	1.54 V	1.54	1.50	1.54	7.02m
1 Peak−Peak	2.92 V	2.92	2.92	3.00	10.8m

Figure 6.19 Filtered RX Audio at 4KHz

Finally, after the signal is offset by the conditioning circuit it is passed through a final low pass filter to ensure that the audio is not affected by any outside noise. It should be noted that due to the equipment being used to measure the signal while also having a DC offset made it so that the scale used to measure the signal be kept relatively high. Leading to the equipment having trouble reading the input frequency. But because the signal being generated was through a function generator and previous measurements demonstrated the frequency of the signal was accurate at 4kHz.



	Value	Mean	Min	Max	Std Dev
1 Max	3.70 V	3.92	3.70	4.50	262m
1 Min	2.26 V	2.09	1.50	2.30	263m
1 Peak−Peak	1.44 V	1.82	1.44	3.00	524m

27 Jul 2016
04:03:20

Figure 6.20 Filtered RX Audio at 9KHz

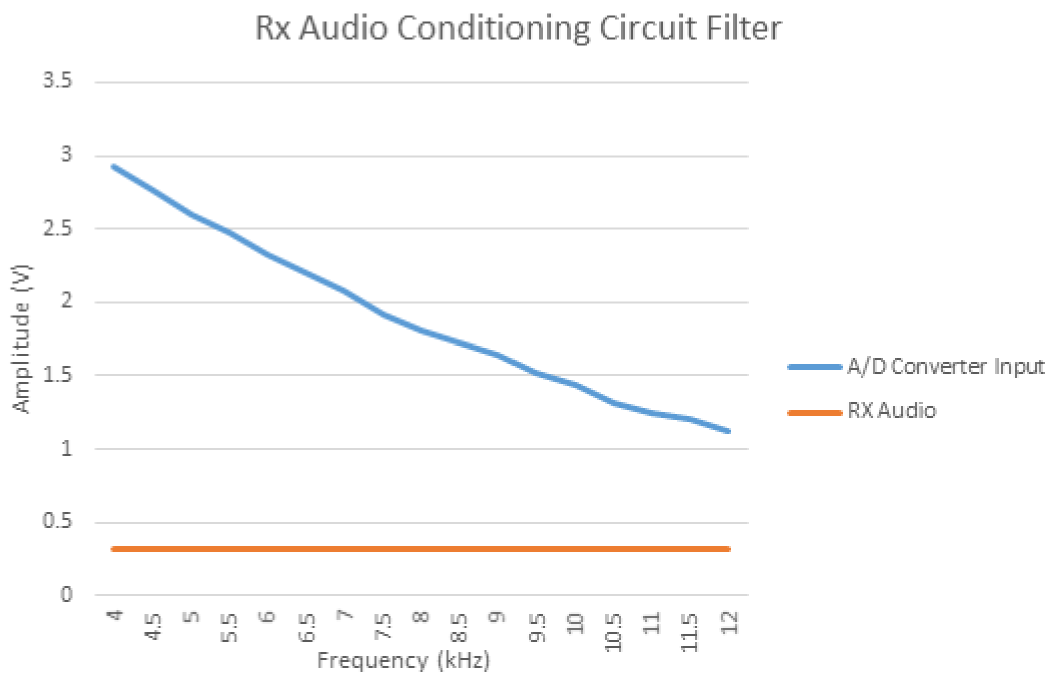


Figure 6.21 RX Audio Circuit Through Low Pass Filter

As it can be seen the amplitude of the signal when at higher frequencies is considerably lower than that of the 4kHz used in radio communication. Ensuring that a 2nd order Butterworth low pass filter will be enough to properly filter out any noise in the RX Audio prior to digital conversion. Though a higher order Butterworth would be possible it would not serve much purpose since noise is going to be in the MHz range. This higher order Butterworth would not only require more components, this increasing costs it would also serve little to no purpose. With the current gain drop at higher frequencies once MHz are reached the signal would be almost nonexistent.

6.2.2 AGC Signal

Since the AGC signal coming from the King KX-170B is a DC signal connecting the circuit input into a DC power source, and using that DC voltage as the AGC values was used for circuit testing. This provided information on how the circuit would behave once finally interfaced with the King KX-170B VHF Aircraft Radio.

AGC In (V)	AGC Stage I Out (V)	AGC Stage II Out (V)
2.1	3.09	1.54
2.34	3.44	1.73
2.5	3.68	1.84
2.7	3.97	1.98
2.9	4.26	2.14
3.1	4.56	2.28
3.35	4.93	2.67
4.06	5.97	2.99
4.3	6.32	3.16
4.56	6.7	3.36
4.79	7.04	3.53
5.01	7.37	3.69
5.2	7.65	3.83
5.45	8.02	4.01
5.7	8.38	4.2
6.01	8.83	4.42
6.33	9.31	4.66
6.45	9.49	4.75

As it can be observed the AGC signal being output into the analog to digital converter in the interface board prior to the Raspberry Pi 3 is properly within the range of the converter. Keeping into consideration the range the signal is able to achieve out of the first stage. This is because the operational amplifier would start to approach rail voltage and start giving inaccurate gain if the output came too close to rail. Thus a conservative maximum bellow 10V is appropriate for a 12V Vcc. This also translated into the input for the analog to digital converter voltage divider. Values for the resistors were selected to provide an input close to the 1V minimum input, for accuracy purposes, and the 5V maximum of the analog to digital converter input.

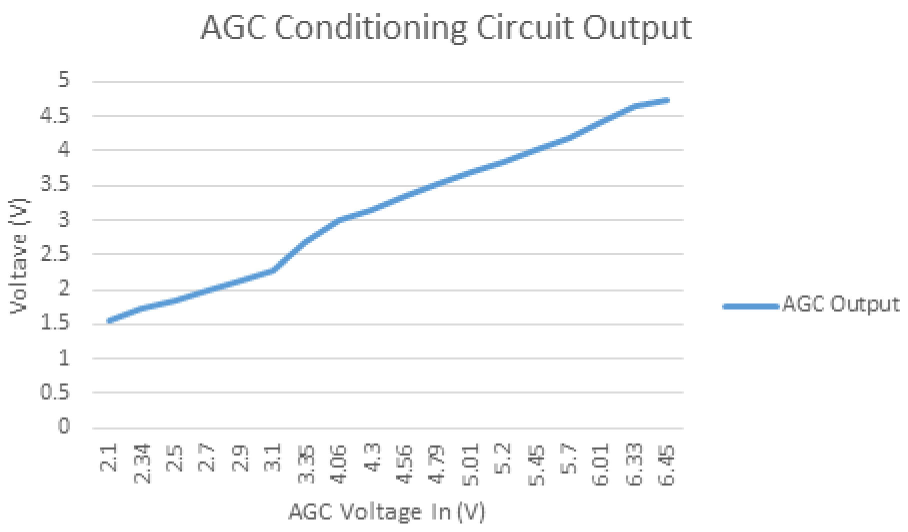


Figure 6.22

6.2.3 PTT Signal

When testing the PTT signal the idea behind it was to check that when the test power supply or the GPIO pin input from the Raspberry Pi 3 would result in a voltage in the King KX-170B VHF Aircraft radio input. This was done by running a 3.3V signal through the intended input whilst having the acting ‘switch’ opened to simulate the Raspberry Pi 3 having an input. Then the test circuit was tested. This circuit had the acting switch closed and the Raspberry Pi 3 input set to ground, as the Raspberry Pi 3 would be acting as ground in this specific case.

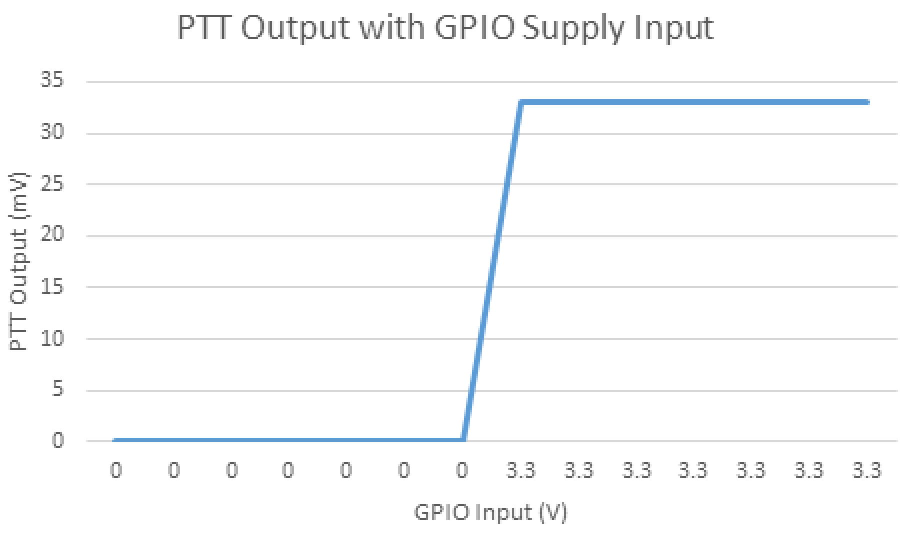


Figure 6.23

When the input is simulated as the GPIO pin the output of the PTT to the King KX-170B radio is a voltage of 33mV. This is enough to ground the relay inside the radio and set it into transmit mode.

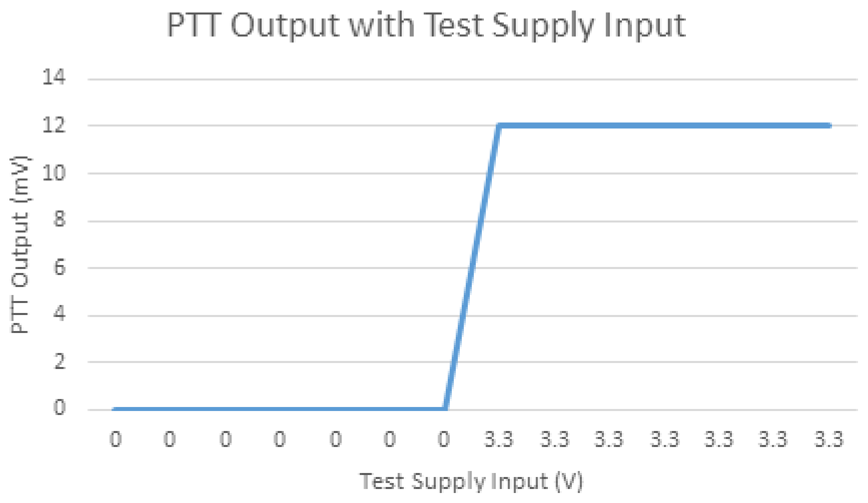


Figure 6.24

As it can be noted the voltage coming out when using the test voltage supply is slightly lower than that of the GPIO pin. This is due to the voltage divide caused by the resistors. This is to ensure that the current going into the Raspberry Pi 3 remains bellow 16mA at all time.

7 Testing

7.1 3.3V Voltage Regulator Testing Procedures

To test the Interface Board 3.3V Voltage Regulator, follow the testing procedure below.

Process	Expected Outcome
Set up the 3.3V regulator with required components for correct output. Capacitors and resistors are the only components needed. Capacitors connected at the input and output to prevent oscillation.	No output yet as there is no input connected
Connect 13.8V input along with connections to voltage meter or oscilloscope for measurement purposes.	Once input is ON, output should be 3.3V
Drop input voltage to 5.1V	Voltage output should still be 3.3V
Drop input voltage to less than 5V	Output will not be a constant 3.3V

7.2 5V Voltage Regulator Testing Procedures

To test the Interface Board 5V Voltage Regulator, follow the testing procedure below.

Process	Expected Outcome
Set up the 5V regulator with required components for correct output. Capacitors are the only components needed. Connected at the input and output to prevent oscillation.	No output yet as there is no input connected
Connect 13.8V input along with connections to voltage meter or oscilloscope for measurement purposes.	Once input is ON, output should be 5V
Drop input voltage to 7V	Voltage output should still be 5V
Drop input voltage to less than 6V	Output will not be a constant 5V

7.3 12V Voltage Regulator Testing Procedures

To test the Interface Board 12V Voltage Regulator, follow the testing procedure below.

Process	Expected Outcome
Set up the 12V regulator with required components for correct output. Capacitors are the only components needed. Connected at the input and output to prevent oscillation.	No output yet as there is no input connected
Connect 13.8V input along with connections to voltage meter or oscilloscope for measurement purposes.	Once input is ON, output should be 12V
Drop input voltage to 13.1V	Voltage output should still be 12V
Drop input voltage to less than 13V	Output will not be a constant 12V

7.4 -12V Voltage Regulator Testing Procedures

To test the Interface Board -12V Voltage Regulator, follow the testing procedure below.

Process	Expected Outcome
Set up the interface board by connecting it to the main 13.8V power supply	LED's in interface board should be lit up to indicate power is running through the system.
Using a multimeter connect the positive end to the input and the negative to ground. Test the input voltage.	5V should be displayed on the multimeter.
Using a multimeter connect the positive end to the output and the negative to ground. Test the input voltage.	-12V should be displayed on the multimeter.

7.5 PTT Testing Procedures

To test the Interface Board PTT Circuit, follow the testing procedure below.

Process	Expected Outcome
Turn off the Raspberry Pi 3 microcomputer.	All signals going to the radio should be silent and the Raspberry Pi 3 microcomputer should be off.
Measure the voltage going into the KX-170B Aircraft Radio pin 40	The voltage going in should be 0V
Press the button on the interface board labeled "PTT Test"	The LED labeled "PTT" should light up when the button is pressed
Measure the voltage going into the KX-170B Aircraft Radio pin 40	The voltage going in should be ##V (still need to test for actual value). There should be an audible 'click' as the PTT voltage in the radio gets pulled to ground.
Measure the voltage running through the resistor to the Raspberry Pi 3 GPIO Pin	It should be no larger than 1.2V

7.6 TX Audio Testing Procedures

To test the Interface Board TX Audio Circuit, follow the testing procedure below.

Process	Expected Outcome
Set up the KX-170B VHF Aircraft radio with the interface board and signal generator or the Raspberry Pi 3 as well	All systems but the anemometer should be present for testing procedure
Set the KX-170B VHF Aircraft radio to a frequency of 123.925MHz	The dial on the KX-170B VHF Aircraft radio front plate should read 123.92
Farther away in another room or outside have a receiving radio set to 123.925MHz as well	The other radio should be tuned to the same frequency as the KX-170B Radio
Connect the interface board to the tone generator, or the Raspberry Pi 3 and transmit a tone through it at 4kHz	The audio should go through the interface board and transmitted out. The user with the receiving radio should hear the synthesized audio
Increase the TX signal to 6kHz	The audio should still be just as strong as when initially transmitted
Increment the TX signal by 1kHz until the signal is no longer picked up by the user with the receiving radio	The audio tone should decrease in signal strength until it becomes in audible or noise. The low pass filter should filter anything above 7.2kHz

7.7 RX Audio Testing Procedures

To test the Interface Board RX Audio Circuit, follow the testing procedure below.

Process	Expected Outcome
Set up the King KX-170B VHF Aircraft radio with the interface board, the Raspberry Pi 3 and an oscilloscope as well.	All systems but the anemometer should be present for testing procedure.
Set the King KX-170B VHF Aircraft radio to a frequency of 123.925MHz	The dial on the King KX-170B VHF Aircraft radio front plate should read 123.92
Farther away in another room or outside have a transmitting radio set to 123.925MHz as well.	The other radio should be tuned to the same frequency as the King KX-170B Radio
Connect the interface board to the Raspberry Pi 3 and the oscilloscope to the analog to digital RX audio in. Then transmit a signal using the transmission radio.	The audio should go through the interface board and pushed into the raspberry pi with a clearly visible 3V DC offset. This should be visible through the Oscilloscope

7.8 Anemometer Testing Procedures

To test the Davis 7911 Anemometer, follow the testing procedure below. Note the revised wiring diagram for correct setup.

Process	Expected Outcome
Setup the Davis 7911 Anemometer in a controlled environment with no wind	The Davis 7911 Anemometer should be upright and components should be stationary
Set up transistor with resistors at base, collector and emitter to base.	No output yet as there is no input from anemometer connected yet
Connect 3.3V input at emitter, ground at collector resistor. See schematic.	No output yet as there is no input from anemometer connected yet
Connect input from anemometer at base and output at collector to different channels from oscilloscope.	No output yet as there is no input from anemometer connected yet
Make the following connections: Black – 3.3V Input Red – Not Used Yellow – Wind Speed Green – Ground	No output yet as there is no input from anemometer connected yet
Turn 3.3V source on. Spin wind measuring cups on anemometer about once every 5 seconds to see output	3.3V square wave output should be seen. Base is grounded through reed switch allowing the 3.3V at emitter flow through transistor.
Increase the rate of the wind cups spinning	The faster the cups spin and close the reed switch every time the more square wave outputs will be seen in a shorter amount of time.

7.9 Wind Vane Testing Procedures

To test the Davis 7911 Wind Vane, follow the testing procedure below. Note the revised wiring diagram for correct setup.

Process	Expected Outcome
Setup the Davis 7911 Wind Vane in a controlled environment with no wind and point Wind Vane away from the Anemometer arm	The Davis 7911 Anemometer should be upright and components should be stationary
Set up operational amplifier for wind direction input.	No output yet as there is no input from wind vane connected yet

Power to op amp should be 12V+ and 12V-	No output yet as there is no input from wind vane connected yet
Make the following connections: Black – 3.3V Input Red – Wind Direction Yellow – Not Used Green – Ground	No output yet as there is no input from wind vane connected yet
Connect input and output to oscilloscope	No output yet as there is no input from wind vane connected yet
Turn 5V source on. Slowly move wind vane to adjust direction	Input and output should display as a change on direction is seen

7.10 Testing Modules on PCB

After testing the components on the breadboard, it was decided to add other features for future debugging. Multiple LEDs and test points are used are on the device to be able to have a more accessible and user friendly debugging. The following table shows the LED and/or test point number to be found on the PCB and their corresponding device and expected outcome.

Test Point / LED	Device	Expected Outcome	Color / Name
LED1	Comparator	LED will be ON if: Comparator output is 3.3V meaning Carrier Detect is present. LED will be OFF if: No Carrier Detect is present.	Green
LED2	AGC	LED will be ON if: Voltage from AGC is being amplified and sent to analog and digital converter. LED will be OFF if: Misconnection or no input voltage from AGC.	Red
LED3	RX Audio	LED will be ON if: Received audio from user is being processed and then sent to ADC. LED will be OFF if: Misconnection or no audio is being received from user.	White
LED4	Wind Speed	LED will be ON if: Wind speed is being updated and sent to be processed.	Yellow

		LED will be OFF if: Anemometer is either not properly connected or there is no change in the wind speed.	
LED5	Wind Direction	LED will be ON if: Wind direction is being updated and sent to be processed. LED will be OFF if: Wind vane is either not connected properly or there is no change in wind direction.	Yellow
LED6	PTT Transistor	LED will be ON if: PTT transistor connect to 3.3V at base. Note this 3.3V is not collector. LED will be OFF if: Transistor is open; no input is being received from PTT.	Blue
LED7	Audio Compressor	LED will be ON if: There is audio present as output LED will be OFF if: No audio being processed or output.	Green
LED8	ADC Channel 1	LED will be ON if: AGC voltage is being received from TL084 op amp. LED will be OFF if: No AGC voltage is being received from TL084 op amp.	Red
LED9	ADC Channel 2	LED will be ON if: RX Audio is being received from TL084 op amp. LED will be OFF if: No RX Audio is being received from TL084 op amp.	Red
LED10	ADC Channel 3	LED will be ON if: Wind direction is being received from TL084 op amp. LED will be OFF if: No wind direction is being received from TL084 op amp.	Red

Test Point 1	3.3V Regulator	Accessible TP to check if 3.3V+ regulator output is correct.	TP1
Test Point 2	5V Regulator	Accessible TP to check if 5V+ regulator output is correct.	TP2
Test Point 3	12V Regulator	Accessible TP to check if 12V+ regulator output is correct.	TP3
Test Point 4	3.3V Inverter	Accessible TP to check if 12V- output is correct from voltage inverter.	TP4

7.11 Raspberry Pi Testing Procedures

To test the Raspberry Pi's connection with the interface board, follow the testing procedure below.

Process	Expected Outcome
Power on the Raspberry Pi from the Interface Board	Red LED on Pi should turn on. Verify no signal LEDs on the Interface Board are on
Press the CD Test Button on the Interface Board three times	Verify that after a few seconds the PTT LED and the TX Audio LED on the Interface Board turn on
Press the CD Test Button on the Interface Board four times, then hold once more, to simulate a transmit radio check	Verify that after a few seconds the PTT LED and the TX Audio LED on the Interface Board turn on
Connect headphones or speakers to the Raspberry Pi and once again press the CD Test Button on the Interface Board three times	Verify that the Raspberry Pi plays a wind conditions broadcast with reasonable values.

7.12 Unit Tests

Because it will be hard to tests most of the "Build WAV File" process on a micro level, we will have to do most of the testing via message logging and writing separate testing scripts across the various sub functions the "Build WAV File" function has. In order to properly explain our testing plan for the sub functions of this function, you will find that below we have separated the testing plans for the various sub functions of the system into sections. Each section will include the testing scripts to be used for that particular sub function, as well as go over any messages that will be logged to keep track of errors.

Before we go into each sub functions and how they will be tested, it is important to describe the way that all of the various testing scripts and messages will be linked back to the main “Build WAV File” function and organized. As for logging and messaging, there will be a master text file by the name of BuildWavFile.log whose purpose will be to keep track of any error messages that happen across the system that are related to the “Build WAV File” function and its sub functions. This will be the acting message hub for all of the testing scripts and sub functions to post error messages that relate to the “Build WAV File” function.

7.12.1 Clean and Format Passed in Audio Data Testing Procedures

Testing Scripts

Testing Script	Success Condition
Data.test.py	The resulting digital values from the input file are equal to their respective digital unconverted values from the output file
ByteLength.test.py	The number of lines in the input and out files are the same

Data.test.py

The main goal for testing this sub function is to make sure that the converted and formatted data that was returned from this function matches up with the input data. Since the main goal of this sub function is essentially just to make some conversions, and does not majorly alter the data at all, we can use the opposite process to convert the formatted data back to see if we get the correct digital value. The only potential issue here is that this function subtracts 255 from the initial digital value to compensate for the voltage offset from the interface board used to push the entire signal into the positive range. Considering both options of adding the 255 digital value back to the number and then comparing the input value to the output number, or subtracting 255 from the input number and comparing that to the un converted output value, there really isn’t any major differences between them. We ended up choosing to subtract the 255 digital value, to compensate for the 3V digital offset, from the initial value and comparing that to the output data because this allows us to check and validate the conversion steps essentially halfway between the raw data form and the outputted data. While not totally different from the first option, this does provide a bit more credibility to the test due to the fact that we aren’t simple just reversing the logic of the initial function we are testing. If we went with the first option and based the test on the reversal of the initial function being tested and the logic of the function was wrong, simply reversing this process would lead to a test function with equally wrong logic. This first test will be to validate that the conversion from the raw data to the format we need for the WAVE file did not corrupt the value of the data even if it is represented in a different form.

The second test we will have for this sub process, will be to check the byte sizes of both the input file, containing the raw audio samples, and the output file, containing the converted and formatted audio data for the WAVE file. Even though the input data file contains 10 bit digital values while the output file contains the little endian 2’s complimented representation of the sample, which is 2 bytes in length, we can essentially make sure that the number of lines in the input file match the number of input lines in the output file. This is because each line, no matter which file you are reading from, represents one sample, so the line numbers effectively represent the number of samples contained in that file. While this is a simple test, it is important because we need to make sure that no samples we missed or left out of the conversions for any reason. Losing a sample could have drastic effects on the resulting output audio and can corrupt the resulting WAV file if the WAV file expects more or less bytes than is actually store in the data chunk of the file. This test will help us avoid file corruption due to miss calculated byte sizes in either the input or output audio data.

As for any error messages this sub process may need to pass on to the BuildWavFile.log, we mainly need to focus on file input and output errors as well as any variable typing errors. Because file input and output is how we are receiving our input audio data as well as outputting our formatted audio data from this function, any file input or output errors need to be classified as critical errors to the system. Thus, any such file input or output errors need to be passed to the BuildWavFile.log file. Since the input and output files will always be named the same thing, and overwritten for each iteration of the “Build WAV File” function, the error messages we send to the log file can be the same for any file operational errors that are of the same type. For example, and errors pertaining to the opening of a file may be formatted as “ERROR – unable to open file: <filename>”, where <filename> will be replaced with the file name of the file that failed to open. These error messages can be seen in the complete error message table below. Likewise, since this sub processes uses so many data conversion and formatting methods, we need to make sure that we did not encounter any errors when executing these critical functions. Upon encountering one of these errors we will send a message to the error log. Upon the failure of this sub routine to open the error messaging log, the error will be send to the standard output of the raspberry pi.

Error Messages

Error Type	Error Message
Cannot Open Input Audio File	“ERROR – unable to open file: audioIn.txt”
Cannot Open Output Audio File	“ERROR – unable to open file: audioout.txt”
Cannot Read From Input Audio File	“ERROR – unable to read from file: audioIn.txt”

Cannot Write To Output Audio File	“ERROR – unable to write to file: audioIn.txt”
Error When Converting To 2’s Compliment	“ERROR – Unable to convert sample to 2’s Compliment”
Error When Converting to Short Integer and Little Endian Format	“ERROR – Unable to convert sample to Short Integer and Little Endian Format”

7.12.2 Build Data Chunk, Build Format Chunk, and Build Header Chunk Testing

Testing Scripts

Testing Script	Success Condition
Feilds.test.py	All fields are of the required byte length and any static/predefined fields values are correct
ChunkSizes.test.py	The number of bytes in each chunk is equal to the pre/defined static chunk size or the chunkSize field of the corresponding chunk
DataLength.test.py	The number of bytes after the first 8 bytes of the data chunk is equal to the number in the chunkSize field of the data buffer

Feilds.test.py

The goal of this test is to verify that the all chunks in the resulting WAV file contain the correct fields and that any field with a pre-determined value has been set to that value. This test is important because it will determine if the resulting WAV file from the “Build WAV File” process is a valid wav file and that all fields in each chunk are populated. This will make sure no fields are missing. If this test fails, then it will pass a message to the BuildWavFile.log file with more details on what error was experienced. These messages are accounted for below.

ChunkSizes.test.py

There are a couple of main goals to this test script, however it can be said that this test script validates whether each chunk, as well as the file itself, is of the correct size recorded by the chunkSize in each chunk. While for the data and format chunks, this just means reading the value in the chunkSize fields and validating the chunks are in fact that length in bytes, the header chunk requires that the entire file length be check and will be kept track of as the other two size checks are happening. This actually works out in our favor as we have to essentially read the file byte by byte to get to the fields we need to get to. Lastly it is important to note that the Header chunk is always 12 bytes in length, so we will validate this first in this test script.

<u>DataLength.test.py</u>

While the data size is validated in the testing for the “Clean and Format Raw Audio Data” sub process, we need to again validate that the number of bytes occupied by the data section of the wave file indeed match up to the number store in the data chunk’s subChukSize field. This is a check also being tested in the ChunkSizes.test.py testing script, but here we will make sure that the number of bytes per sample is the same as the number in the format chunks BlockAlign field, which tells any program reading the wave file how many bytes are in each “block” or sample to be read. This is especially important when reading multiple channels of audio since the number of bytes per sample is doubled to cover both the right and left channel. However, in this case we just need to make sure that the byte size here matches the number of bytes for each sample in the data section.

<u>Error Messages</u>

Error Type	Error Message
Incorrect Chunk ID	“ERROR – incorrect chunk id in the: <chunk> Chunk”
Missing Field	“ERROR – Missing the <chunk field> in the: <chunk> Chunk”
Invalid Field Value	“ERROR – incorrect field value in the: <chunk> Chunk. Expected <expected value>. Received <received value>”
Chunk Size does not match value in chunkSize field	“ERROR – Size of: <chunk> Chunk does not equal read chunk size of <chunkSize field value>”

7.13 System Testing Procedures

To test the overall system once each individual component has been tested, follow the testing procedure below.

Process	Expected Outcome
Connect the Davis 7911 Anemometer to the Interface Board	They will be connected
Connect the Radio to the Interface Board	They will be connected
Connect the Monitor and Keyboard to the Raspberry Pi	They will be connected

Connect the Raspberry Pi to the Interface Board	They will be connected
Connect the Power Supply to the Interface Board	The system should power on. All power LEDs, including the Raspberry Pi’s red LED, should turn on
Set the Radio to a channel that is not used in the immediate vicinity	Radio should be set to a new channel
Set the wind vane to its north position and verify the wind cups are stationary	N/A
Key the handheld radio mic three times	Verify wind conditions are broadcast with values of zero degrees and zero knots
After 60 seconds, move the wind vane to its south position, 180 degrees from before	Verify that a wind conditions update is broadcast with the values of 180 degrees and still zero knots
After 60 seconds, rotate the wind cups as fast as possible for at least five seconds	Verify that a wind conditions update is broadcast with the values of 180 degrees and greater than zero knots
After 60 seconds, key the handheld radio mic four times	Verify a transmit radio check prompt is broadcasted
After 60 seconds, key the handheld radio mic four times	Verify nothing happens for the 60 seconds, and after the four keys a transmit radio check prompt is broadcasted
Key the handheld radio mic and speak into the radio, once done speaking, unkey the mic	Verify the message just spoken into the handheld radio is broadcasted and followed by a power level broadcast
After 60 seconds, key the handheld radio mic three times	Verify wind conditions are broadcast
After 60 seconds, key the handheld radio mic one time	Verify nothing happens
After 60 seconds, key the handheld radio mic two times	Verify nothing happens
After 60 seconds, key the handheld radio mic five times	Verify nothing happens

8 Management

8.1 Task List and Milestones

TASK	Estimated Completion Date	Person Responsible	Assistant	Status
Documents				
Task List	6/16/2016	Jordan	Team	IP
Hardware Block Diagram		Jordan	Team	C
Software State Diagram		Jordan	Team	C
Timeline		Jordan	Team	C
Table of Contents	6/29/2016	Jordan	Team	C
User Control Manual	8/2/2016	Jordan	Team	IP
Bill of Parts	7/26/2016	Jordan	Team	IP
Final Document 120pg	7/29/2016	Jordan	Team	IP
Weather Station				
Research weather station	6/15/2016	Mason	Jordan	C
Determine weather station	6/21/2016	Mason	Jordan	C
Buy weather station	6/23/2016	Mason	Jordan	C
Connect to MicroProcessor	7/22/2016	Mason	Jordan	NS
Microporocessor Hardware				
Determine microprocessor	6/17/2016	Jordan	Mason	C
Buy microprocesor	6/23/2016	Jordan	Mason	C
Breadboard and test ADC with Pi	6/28/2016	Mason	Jordan	C
Research inputs from interface board	6/23/2016	Mason	Jordan	C
Research audio output through headphone jack	6/30/2016	Jordan	Mason	C
Connect inputs and outputs	7/22/2016	Jordan	Mason	NS
Software				
Determine Linux distro to use	6/23/2016	Mason	Jordan	C
Determine coding language to use	6/28/2016	Jordan	Mason	C
Write high level logic/state diagram	6/23/2016	Jordan	Mason	C
Code WX input data	7/15/2016	Mason	Jordan	C
Have Demo of Python multi-threading/pipes	7/23/2016	Jordan	Mason	C
Code WX module	7/23/2016	Jordan	Mason	C
Code TX module	7/29/2016	Mason	Jordan	IP
Code Input to Wav file module	7/28/2016	Mason	Jordan	IP
Code entire program	7/30/2016	Jordan	Mason	IP
Interface Board Overall				
Draft schematics for ALL circuits in the updated block diagram	6/28/2016	Francisco	Carmen	C
Make an integrated schematic that shows all components of Interface Board	7/21/2016	Carmen	Francisco	C
Build breadboard from new schematic and test with anemometer	7/22/2016	Francisco	Carmen	C
Take breadboard to Prof. Young's office and test with radio	7/25/2016	Carmen	Francisco	C
Demo all testing for Prof. Young	7/25/2016	Francisco	Carmen	C

Demo MCU Audio recording and playback	7/26/2016	Mason	Jordan	IP
Finalize Interface Board Schematic	7/26/2016	Carmen	Francisco	IP
Once previous steps are completed, Build the Vector Board	7/29/2016	Francisco	Carmen	NS
Weather Station				
Add two new inputs from weather station to block diagram	6/28/2016	Carmen	Francisco	C
PTT to TTL PTT				
Research transistor and resistor setup				C
Design schematic				C
Choose parts				C
Prototype and test	7/22/2016	Francisco	Carmen	IP
Carrier Detect to TTL Carrier Dectect				
Research comparator				C
Study schematic and squelch				C
Design schematic				C
Choose parts				C
Prototype and test	7/22/2016	Francisco	Carmen	IP
RX Audio to SPI bus		Francisco	Carmen	
Conditioner				C
Analog to Digital Converter	6/26/2016			C
Design Schematic				C
Choose parts				C
Prototype and test	7/22/2016	Francisco	Carmen	IP
AGC voltage to SPI bus		Carmen	Francisco	
Conditioner	6/21/2016			C
Analog to Digital Converter	6/21/2016			C
Design schematic	6/28/2016			C
Choose parts				C
Prototype and test	7/22/2016	Francisco	Carmen	IP
TX Audio		Francisco	Carmen	
Audio conditioning	6/26/2016			C
Compressor				C
Design schematic				C
Prototype and test	7/22/2016	Francisco	Carmen	IP
Power		Carmen	Francisco	
Supply to device (12V)	6/21/2016	Carmen		C
Inverted 12V for opamps	6/21/2016	Carmen		C
5V design	6/21/2016	Carmen		C
3V Design	6/21/2016	Carmen		C
Design schematic for power distribution	6/28/2016	Carmen		C
Schematic Review	6/28/2016			C
Choose parts				C
Prototype and test	7/22/2016	Francisco	Carmen	IP
Radio		Carmen/Francisco	Carmen/ Francisco	
Fix 175B radio				IP
Determine output pins				C

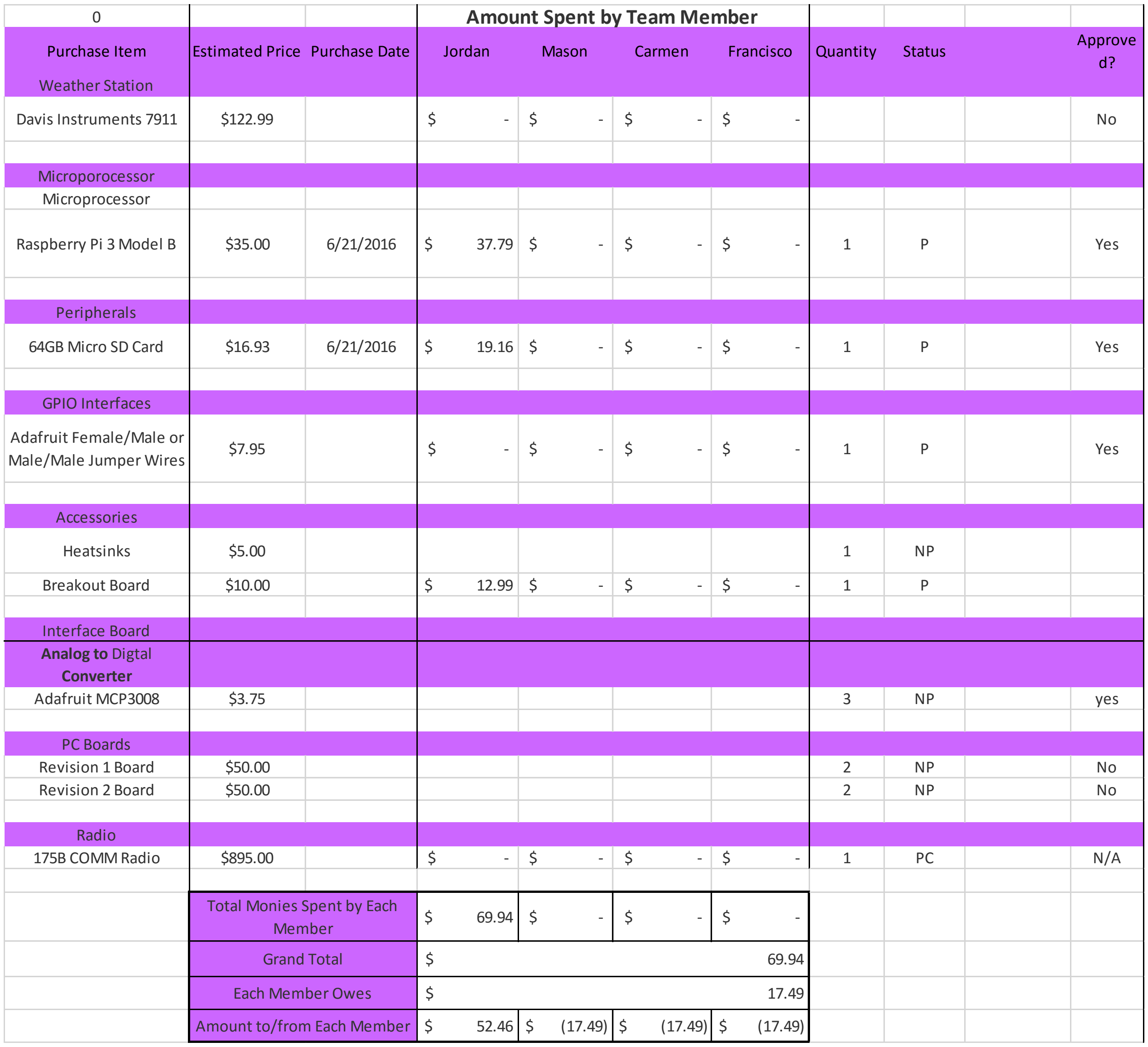
Figure 8.1 Task list as of 7/27/2016

Figure 8.1 shows our task list for this project. It includes any task that could take more than 15 minutes and generally is used as a guide to tell members what to do next. Each task has a name, a due date, a member responsible, an assistant, and a status. Competed tasks are also marked in green to easily show what is done, and also what isn't, to easily show members what still needs to be done.

Our goal for this semester in Senior Design I was to create a working prototype of our system. Therefore, this task list does not include anything Senior Design II will need past the prototyping stage. Also, multiple tasks are marked as In Progress to show that we are still working on them.

8.2 Budget and Finances

Our sponsor has provided us with a $250 budget for approved parts. Below is a list of items we purchased and their cost as well as which member bought them. The cost of these items, if they should exceed our $250 budget will be absorbed evenly by the team members.



0			Amount Spent by Team Member							
Purchase Item	Estimated Price	Purchase Date	Jordan	Mason	Carmen	Francisco	Quantity	Status		Approved?
Weather Station										
Davis Instruments 7911	$122.99		$ -	$ -	$ -	$ -				No
Microporocessor										
Microprocessor										
Raspberry Pi 3 Model B	$35.00	6/21/2016	$ 37.79	$ -	$ -	$ -	1	P		Yes
Peripherals										
64GB Micro SD Card	$16.93	6/21/2016	$ 19.16	$ -	$ -	$ -	1	P		Yes
GPIO Interfaces										
Adafruit Female/Male or Male/Male Jumper Wires	$7.95		$ -	$ -	$ -	$ -	1	P		Yes
Accessories										
Heatsinks	$5.00						1	NP		
Breakout Board	$10.00		$ 12.99	$ -	$ -	$ -	1	P		
Interface Board										
Analog to Digtal **Converter**										
Adafruit MCP3008	$3.75						3	NP		yes
PC Boards										
Revision 1 Board	$50.00						2	NP		No
Revision 2 Board	$50.00						2	NP		No
Radio										
175B COMM Radio	$895.00		$ -	$ -	$ -	$ -	1	PC		N/A
	Total Monies Spent by Each Member		$ 69.94	$ -	$ -	$ -				
	Grand Total		$ 69.94							
	Each Member Owes		$ 17.49							
	Amount to/from Each Member		$ 52.46	$ (17.49)	$ (17.49)	$ (17.49)				

Figure 8.2 Team Budget as of 7/27/2016

8.3 Team Member Time Management

Week	Week Ending Date	Jordan		Carmen		Mason		Francisco	
		Meetings / Learning / Research	Productive Effort / Report Writing	Meetings / Learning / Research	Productive Effort / Report Writing	Meetings / Learning / Research	Productive Effort / Report Writing	Meetings / Learning / Research	Productive Effort / Report Writing
1	5/29/2016	1	0	1	0	1	0	1	0
2	6/5/2016	4	2	4	2	4	2	4	2
3	6/12/2016	3	3	3	2	8	1	5	3.75
4	6/19/2016	5.5	2.5	12	4	4	4	5.5	7
7	6/26/2016	1	1	10	5	2	6	6	6
8	7/3/2016	4.5	10	3	3.5	3	3	10	1
9	7/10/2016	5	8	5	4	1	5	5	8
10	7/17/2016	4	9.5	2.5	6	3	2	1	5.5
11	7/24/2016	3	3	6	8	5	12	2	5
12	7/31/2016	5	12	3	8	4	5	5	6
13	8/7/2016	2.5	4	2.5	4	2.5	4	2.5	4
Time Spent		38.5	55	52	46.5	37.5	44	47	48.25
Total Time		93.5		98.5		81.5		95.25	

9 Appendix

9.1 Datasheets

9.1.1 TPS 63700 Voltage Inverter

http://www.ti.com.cn/cn/lit/ds/symlink/tps63700.pdf

9.1.2 TL084 Op-Amp

http://www.ti.com.cn/cn/lit/ds/symlink/tl084.pdf

9.1.3 LM234 Op-Amp

http://www.ti.com/lit/ds/symlink/lm134.pdf

9.1.4 2N4403 PNP Transistor

https://www.fairchildsemi.com/datasheets/MM/MMBT4403.pdf

9.1.5 2N4401 NPN Transistor

http://www.tme.eu/en/Document/f8d055de9406c1d3890391b8977313b3/2N4401BU.pdf

9.1.6 2N3820 FET

http://www.radiotechnika.hu/images/2N3820.pdf

9.1.7 BC546 NPN Transistor

http://www.onsemi.com/pub_link/Collateral/BC546-D.PDF

9.1.8 LM393 Comparator

http://www.ti.com/lit/ds/symlink/lm2903-n.pdf

9.1.9 LM124W Op-Amp

http://www.st.com/content/ccc/resource/technical/document/datasheet/60/2a/ed/7f/ec/7e/4e/5c/CD00005087.pdf/files/CD00005087.pdf/jcr:content/translations/en.CD00005087.pdf

9.1.10 MCP3008 Analog to Digital Converter

https://cdn-shop.adafruit.com/datasheets/MCP3008.pdf

9.1.11 2N5458 JFET

http://www.onsemi.com/pub_link/Collateral/2N5457-D.PDF

9.1.12 78L12 Voltage Regulators

http://www.ti.com.cn/cn/lit/ds/symlink/lm78l05.pdf

9.1.13 AC90-50D

http://www.faa.gov/documentLibrary/media/Advisory_Circular/AC90-50D.pdf

9.1.14 Raspberry Pi 3 Model B

https://www.inet.se/files/pdf/1974044_0.pdf