# Emergency, Need Backup!

## Final Report

Team Number: SD May21-43

Client: Collins Aerospace

Adviser: Dr. Andrew Bolstad

**Team Members (Role):**

Marcelo Abrantes (Chief Engineer: Power Systems)

James Curtis (Meeting Scribe)

Caroline Easley (Meeting Facilitator)

Michael Kuehn (Communications Director)

Benjamin Welte (Project Documentation)

Abbey Wilder (Test Engineer)

Stepan Zelenin (Chief Engineer: Communications)

Team Email: sdmay21-43@iastate.edu

Team Website: https://sdmay21-43.sd.ece.iastate.edu

## Acknowledgement

The design team would like to specially thank Dr. Andrew Bolstad for his technical expertise and advice as well as Collins Aerospace for the generous funding it provided for this project.

# Table of Contents


Table of Contents ... 3
Table of Figures ... 5
1 Introduction: Project Background & Overview ... 6
2 Project Design ... 6
2.1 Requirements, Constraints, and Standards ... 6
2.1.1 Functional Requirements ... 6
2.1.2 Constraints ... 7
2.1.3 Standards ... 7
2.2 Design Evolution ... 8
2.3 Final Revised Design ... 9
2.3.1 Transmitter ... 9
2.3.2 Receiver ... 11
2.3.3 SNMP Control ... 12
2.4 Component Details ... 13
2.4.1 Low-Pass Filter ... 13
2.4.2 Filter Bank ... 14
2.4.3 Mixer ... 14
2.4.4 Local Oscillator ... 15
2.4.5 Push-To-Talk ... 15
2.4.6 Voltage Regulators ... 15
2.4.7 Level Shifters ... 16
2.5 PCB Design ... 16
2.6 Security Concerns & Countermeasures ... 17
2.6.1 Physical Security ... 17
2.6.2 Cybersecurity ... 17
3 Implementation ... 18
3.1 PCB Assembly ... 18
3.2 Local Oscillator Programming ... 19
3.3 SNMP User Interface ... 21
3.4 Integration ... 22
4 Testing Process & Results ... 22

4.1 General Testing Plan ........ 22
4.2 Component Testing & Results ........ 22
4.2.1 High Frequency Filters ........ 23
4.2.2 Mixer ........ 23
4.2.3 Local Oscillator ........ 24
4.2.4 MIB Browser and Agentuino ........ 24
4.2.5 Miscellaneous Parts ........ 25
4.3 System Testing & Results ........ 25
5 Context: Related Products & Literature ........ 27
Appendix I: Operation Manual ........ 28
Before Starting ........ 28
Controlling the Radio ........ 28
Setup ........ 28
Reading Configuration Information ........ 29
Writing Configuration Information ........ 30
Push To Talk ........ 30
Appendix II: Alternative Designs ........ 30
Appendix III: Other Considerations ........ 30
Appendix IV: Code ........ 31
SNMP_LO.ino ........ 31
DHCP.ino ........ 36
LO.ino ........ 39
Appendix V: Abbreviations & Acronyms ........ 53
Appendix VI: Works Cited ........ 54

# Table of Figures

Figure 1: Radio Block Diagram ........ 9
Figure 2: Transmitter Block Diagram ........ 10
Figure 3: Receiver Block Diagram ........ 11
Figure 4: SNMP Control Circuit ........ 12
Figure 5: Low-Pass Filter Schematic ........ 13
Figure 6: Simulated LFP Frequency Response ........ 13
Figure 7: Filter Bank Schematic ........ 14
Figure 8: Mixer & Supporting Circuitry ........ 15
Figure 9: Final PCB Layout ........ 16
Figure 10: Final PCB ........ 19
Figure 11: LO Test Circuit ........ 20
Figure 12: LO Oscilloscope Output ........ 20
Figure 13: MIB Browser User Interface ........ 21
Figure 14: Modulated Transmitter Output ........ 26
Figure 15: Modulated Music From a Desktop PC ........ 27
Figure 16: MIB Browser Interface ........ 29
Figure 17: MIB Browser Interface ........ 29

# 1 Introduction: Project Background & Overview

Airplane pilots rely on a radio for communication, so if it breaks, they can no longer communicate with air traffic control (ATC) or other aircraft. This causes many problems since ATC is no longer able to communicate with the aircraft to coordinate airspace deconfliction. It also means that the airplane pilots cannot communicate any ongoing problems to ATC.

To solve the myriad complications that can result from a radio malfunction, this project aims to create a backup radio for use as an alternate communication device if an airplane's main radio fails. Designing a backup radio in its entirety is outside the scope of a two-semester class, so this project's goal is to design and implement a prototype capable of basic amplitude modulation (AM) signal transmission and reception as well as sending status updates and receiving commands via Ethernet. This prototype will then be used by future engineering teams as the foundation for a full design capable of passing civil certification.

# 2 Project Design

## 2.1 Requirements, Constraints, and Standards

This specification establishes performance requirements for the Receiver/Transmitter (hereby referred to as the RT) Airborne Emergency Back Up communication system.

### 2.1.1 Functional Requirements

1. The RT shall provide unencrypted voice communications.
2. The RT shall transmit and receive over the following frequency ranges:
   - 117.975 to 137.000 MHz
   - 225 to 400 MHz
3. The RT shall provide the following tuning increments over the defined frequency ranges shown in Table 1:

| Frequency Range | Tuning Resolution |
|---|---|
| 117.975 to 137.000 MHz | 8.33 kHz, 25 kHz |
| 225 to 400 MHz | 25 kHz |

**Table 1: Operating Frequency Range**

4. The RT shall support Amplitude Modulation (AM) only.
5. The RT shall provide an Ethernet port in accordance with IEEE 802.3 (10/100 Base-T) for control and status operations.

6. The RT shall provide an Ethernet Simple Network Management Protocol (SNMP) interface for the control and status operations shown in Table 2.

| Command | Options |
|---|---|
| Frequency | Tunable frequency from 117.975 to 400 MHz |
| | |
| **Status** | **Options** |
| Frequency | Report currently tuned frequency from 117.975 to 400 MHz |
| Tx/Rx Mode | Report current operational mode |

**Table 2: SNMP Operations**

7. The RT shall provide a discrete input, Push-To-Talk (PTT) that is used to enable/disable transmit operations.
8. The RT shall provide an input capable of accepting baseband analog voice signals.
9. The RT shall provide a balanced 150 ohm +/-10% narrowband audio input interface.
10. The RT shall provide a balanced 600 ohm +/-10% narrowband audio output interface.

### 2.1.2 Constraints

1. The RT electronics piece part cost shall not exceed $1500.
2. The RT shall not exceed 6.0 pounds in total weight.
3. The RT shall not exceed dimensions of 6.0 inches x 6.0 inches x 6.0 inches.

### 2.1.3 Standards

1. ED-23C – Minimum Operational Performance Specification for Airborne VHF Receivers
    - The RT shall adhere to these standards' specifications for a Class H2 receiver and dual class transmitter
2. RTCA/DO-160G – Environmental Conditions and Test Procedures for Airborne Equipment
    - The RT should provide a path to compliance with these standards' operational environment specifications.

3. MIL-STD-188-243A – Tactical Single Channel Ultra High Frequency Radio Communications

    - The RT should comply with these standards' requirements for operation within the Ultra High Frequency (UHF) band

4. RTCA/DO-254 – Design Assurance Guidance for Airborne Electronic Hardware

    - The RT should provide a path to compliance with these standards' operational environment specifications.

5. IEEE 802.3 – Ethernet Working Group Standards

    - The Ethernet port on the RT should comply with the specifications established in this standard.

6. Assorted Simple Networking Management Protocol (SNMP) standards: RFC 3410, RFC 3411, RFC 3412, RFC 3413, RFC 3414, RFC 3415, RFC 3417, and RFC 3418

    - The RT's Ethernet communications should comply with the specifications for the Simple Networking Management Protocol established in these standards.

## 2.2 Design Evolution

As the design team tested the initial radio design, several parts were replaced to better meet the project's functional requirements. Most of these revised part selections resulted from inadequacies in the original parts that were either omitted from their datasheets or not noticed during initial part selection. Specifically, a new high pass filter was selected with a higher corner frequency, a new mixer was picked for its capability to both upconvert and downconvert, and a new local oscillator was chosen because the previous oscillator could not generate frequencies across the entire range specified in the project's requirements. These new part selections changed the layout of the PCB because they had different footprints than the parts they replaced.

Additionally, the circuit schematic was updated to include impedance matching so as to maximize signal integrity and power delivery.

### 2.3 Final Revised Design

The radio design that this project implements consists of three major functional blocks: a receiver to demodulate an audio signal, a transmitter to modulate an audio signal, and an SNMP control circuit to receive and carry out commands from a computer. The interaction between these blocks is shown below in Figure 1 and is elaborated in the following sections.

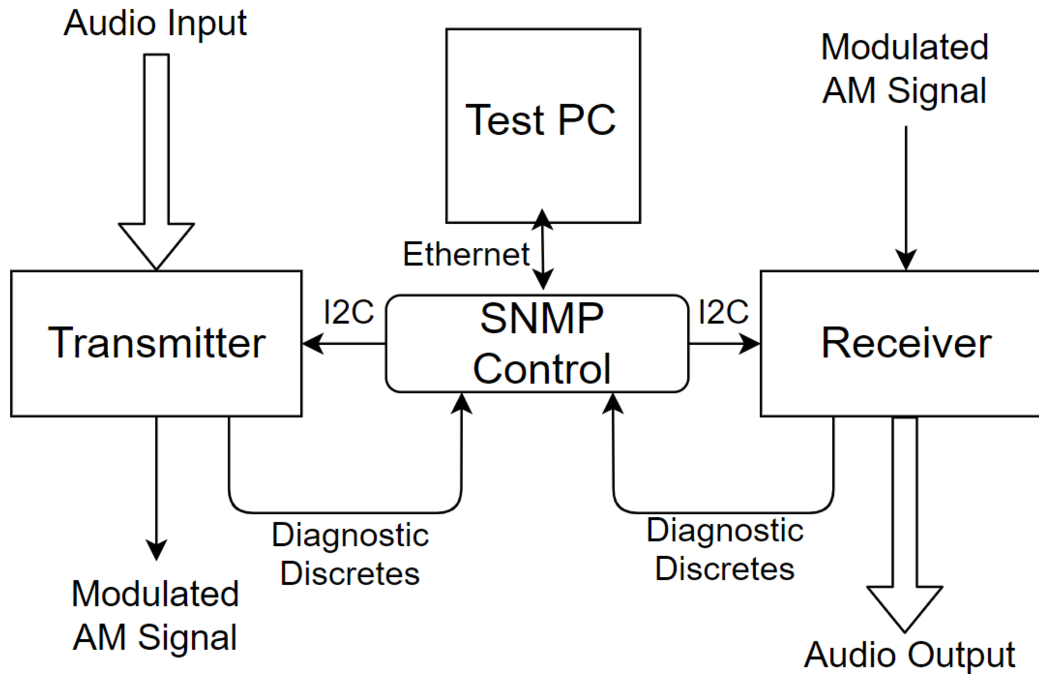


**Figure 1: Radio Block Diagram**

#### 2.3.1 Transmitter

The transmitter includes a microphone, a filter bank, a local oscillator (LO), a mixer, an analog multiplexer, and an analog demultiplexer.

First, the microphone (which serves as a balanced 150 ohm +/- 10% narrowband audio input interface) captures the external sound whenever the push to talk button is pressed, and this signal is then sent to a filter bank which isolates frequencies under 3kHz, the range of the human voice. More details on the custom filter bank are presented in section 2.4.

The resulting filtered signal is then modulated using two components, an LO and a mixer IC. The LO receives a command via I2C from the microcontroller to adjust its output signal to a specific frequency. This output is then mixed with the audio signal coming from the low-pass filter (LPF). This process modulates the amplitude of the LO signal containing the audio information. Similar to the filter bank, more information is presented on the LO and mixer in section 2.4.

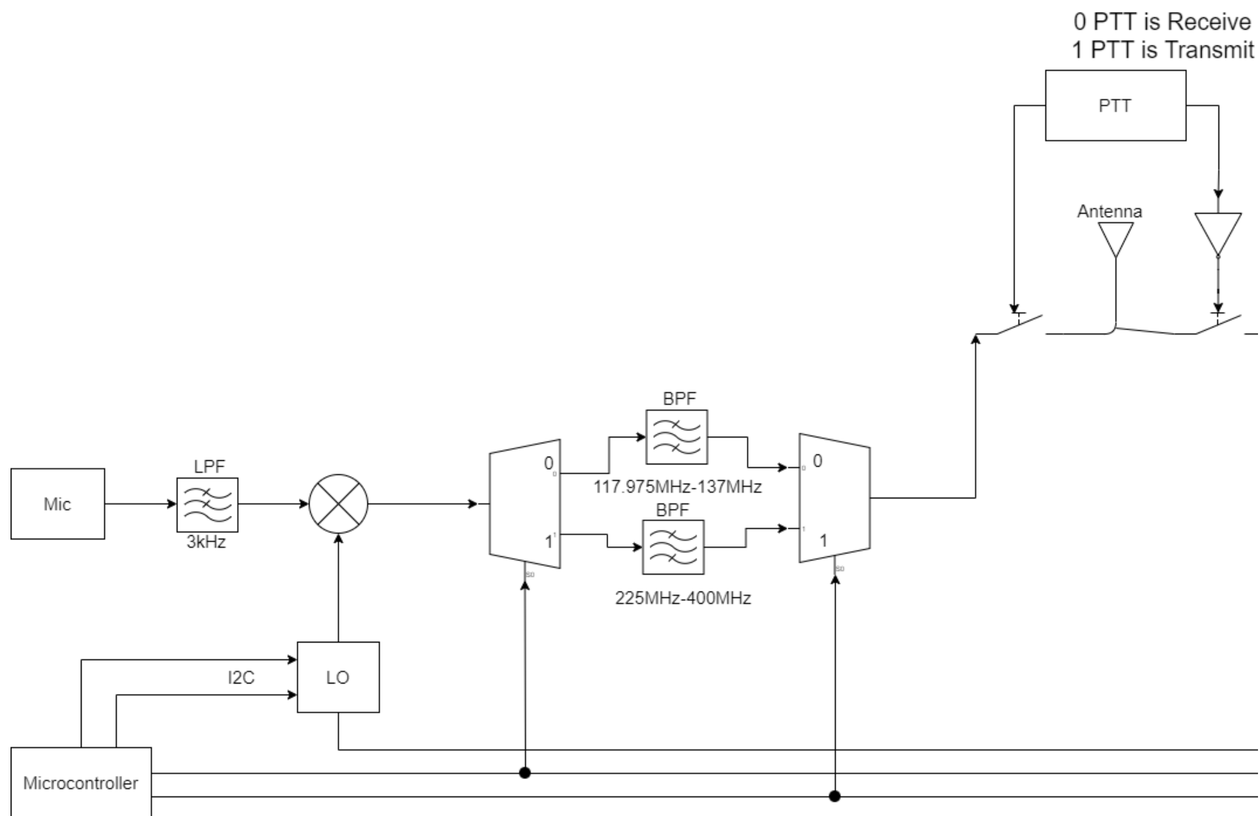


**Figure 2: Transmitter Block Diagram**

For the modulated signal from the mixer to be transmittable, its frequency must fall within the desired band. A custom filter bank including a 1:2 analog multiplexer, one 117.975MHz to 137MHz band-pass filter (BPF), one 225MHz to 400MHz BPF, and a 2:1 analog demultiplexer ensures that this is the case. More details on this custom filter bank are included in section 2.4.

This transmitter design fulfills the requirements to (1) provide unencrypted voice communications, (2) transmit signals over the specified frequency range, (3) transmit signals on the specified frequency channels, (4) support only AM modulation, (8) provide an input capable of accepting baseband analog voice signals, and (9) provide a balanced 150 ohm +/- 10% narrowband audio input interface.

### 2.3.2 Receiver

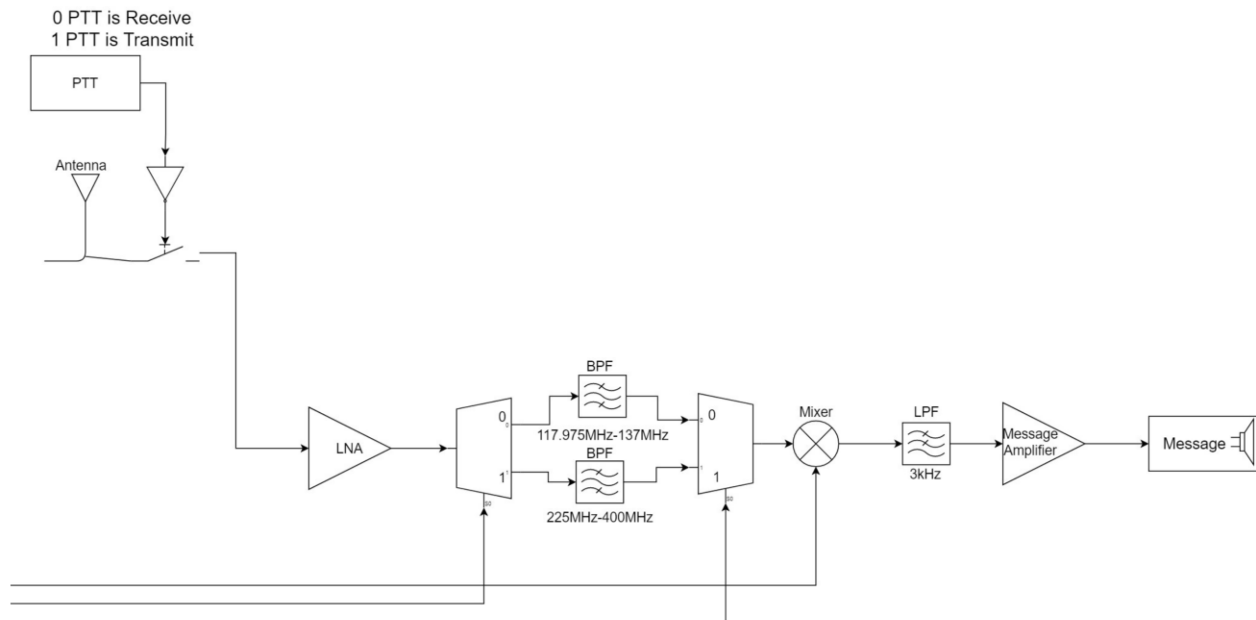


**Figure 3: Receiver Block Diagram**

The receiver sends an input signal through a series of filters and amplifiers to produce an audible message.

First, the receiver gets an input from an antenna that was modeled during testing with a cabled connection. Then, to eliminate signals outside the specified frequency range, this initial input then passes through a BPF bank corresponding the current frequency band after passing through a low-noise amplifier (LNA). The LNA combats the inevitable attenuation of the message signal during transmission.

After being amplified and filtered, the signal is then frequency-shifted by a mixer using the output from a local oscillator and filtered again, this time by a low-pass filter (LPF). The mixer shifts the amplified signal so that it is centered around a frequency of zero Hertz as opposed to the frequency that it was broadcasted on. More details on the mixer are presented in section 2.4. The LPF then ensures that the final signal is restricted to the bandwidth of the human voice. Similar to the mixer, more details are presented on the low pass filter in section 2.4.

Finally, the message signal is amplified again before being outputted to a speaker or a similar device to be made audible. The output port is a balanced 600 ohm +/- 10% audio output interface.

This receiver design fulfills the requirements to (1) provide unencrypted voice communications, (2) receive over the specified frequency range, (3) receive on the specified frequency bandwidths, (4) support amplitude modulation only, and (10) provide a balanced 600 ohm +/- 10% audio output interface.

### 2.3.3 SNMP Control

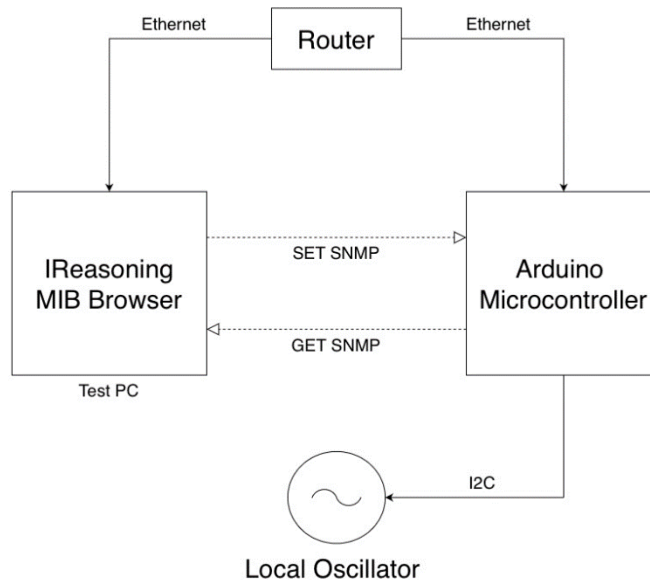


**Figure 4: SNMP Control Circuit**

The SNMP control circuit receives SNMP commands and requests from test software running on a computer. An Ethernet shield enables an Arduino to decode the SNMP messages before accordingly sending control signals to the rest of the radio.

Open-source software known as iReasoning MIB Browser sends simple SNMP commands and requests to the Arduino to change and query the radio's operating frequency. The SNMP messages are sent over a physical Ethernet cable.

The Arduino receives SNMP messages from the Ethernet-based local area network (LAN). It then decodes the SNMP message and performs the appropriate action such as setting the oscillator frequency with an I2C command or reporting back the frequency channel currently being transmitted or received on.

This SNMP control circuit fulfils the design requirements to (5) provide a standard ethernet port and (6) provide an SNMP interface capable of supporting the specified commands.

Additional design constraints such as (2) the radio shall not exceed 6.0 pounds in total weight and (3) the radio's size shall not exceed 6.0 inches by 6.0 inches by 6.0 inches were observed during part selection. This iteration of the project need not conform to the ED-23C standard, but future versions will need to pass ED-23C certification, so the ED-23C standard served as a guideline for the design process. Specifically, the radio will need to be a class H2 receiver and a dual class transmitter, as specified by section 2.6.2 of the ED-23C standard.

## 2.4 Component Details

### 2.4.1 Low-Pass Filter

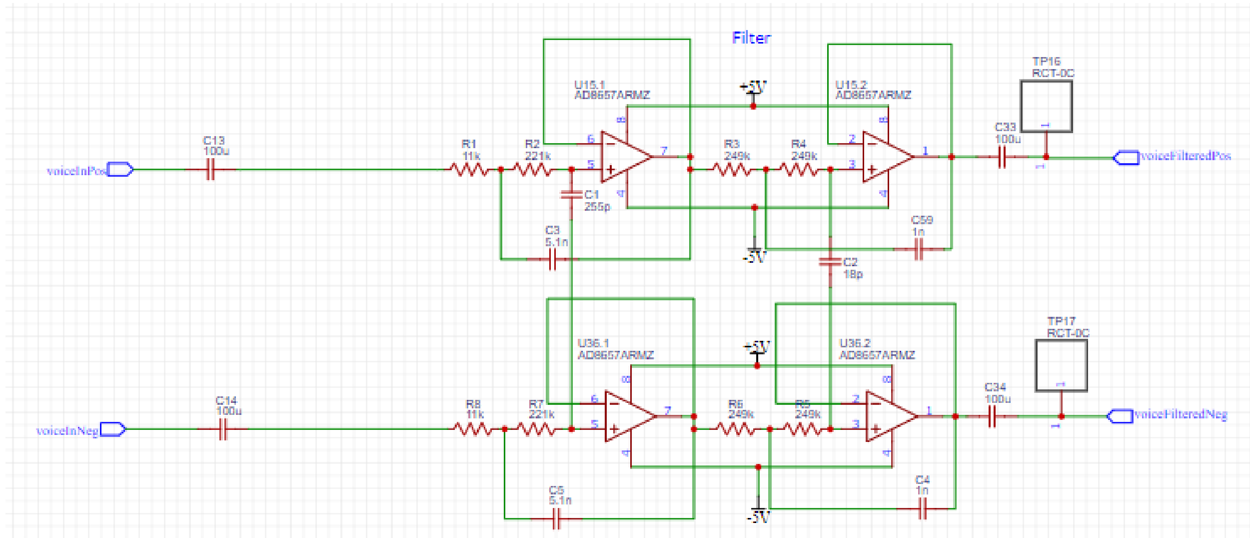


**Figure 5: Low-Pass Filter Schematic**

The design presented in section 2.3 required a custom differential 3kHz low pass filter that is pictured above in Figure 5. It filters out environmental noise outside the frequency range of the human voice at the beginning of the transmitter's signal chain. It is also utilized at the end of the receiver's signal chain to similarly filter out noise outside the range of the human voice. The circuit was modeled in LTspice and its simulated frequency response is shown in Figure 6.

**Figure 6: Simulated LFP Frequency Response**

### 2.4.2 Filter Bank

The design proposed in section 2.3 also contains a high frequency filter bank to filter out unwanted harmonics in signals from both the mixer and the environment. As seen in Figure 7 below, this filter bank consists of a 1:2 analog multiplexer, one 117.975MHz to 137MHz BPF, and one 225MHz to 400MHz BPF. An onboard Arduino controls the multiplexers. DC blocking capacitors ensure that no DC offset voltages are introduced into the signal chain.

The filter bank's input connects to the input of the demultiplexer, whose outputs connect to each BPF. Similarly, the output of each BPF connects to each input of the multiplexer, and the multiplexer's output is the final output of the filter bank. A control signal from the Arduino determines if the multiplexer and demultiplexer route the filter bank's input through filters corresponding to the radio's UHF and VHF bands to eliminate harmonics from the audio signal. A detailed schematic of the filter bank is presented below in Figure 7.

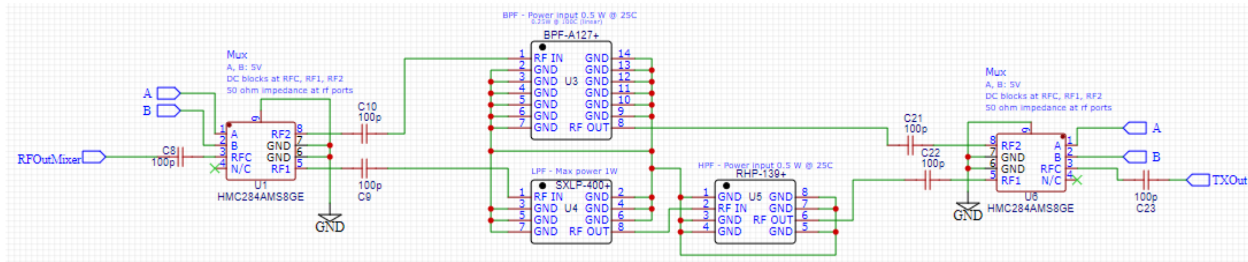


**Figure 7: Filter Bank Schematic**

### 2.4.3 Mixer

Another critical component in the design presented in section 2.3 is the AD8343 mixer. The mixer converts an input audio signal into an AM output signal, or vice versa. It does this by multiplying the input signal by a carrier wave supplied by the local oscillator. The frequency of the carrier wave is the modulation frequency for the resulting AM wave.

The mixer's supporting circuitry can be seen in Figure 8 below. Points of note include the primary inputs and outputs. The differential mixer input is on the left marked as voiceFilteredPos and voiceFilteredNeg. This signal mixes with the differential oscillator input marked as LO_P and LO_N at the top of the figure. The mixer's output is differential before being converted to a single ended signal, RFOutMixer, which is compatible with the high frequency filter bank.

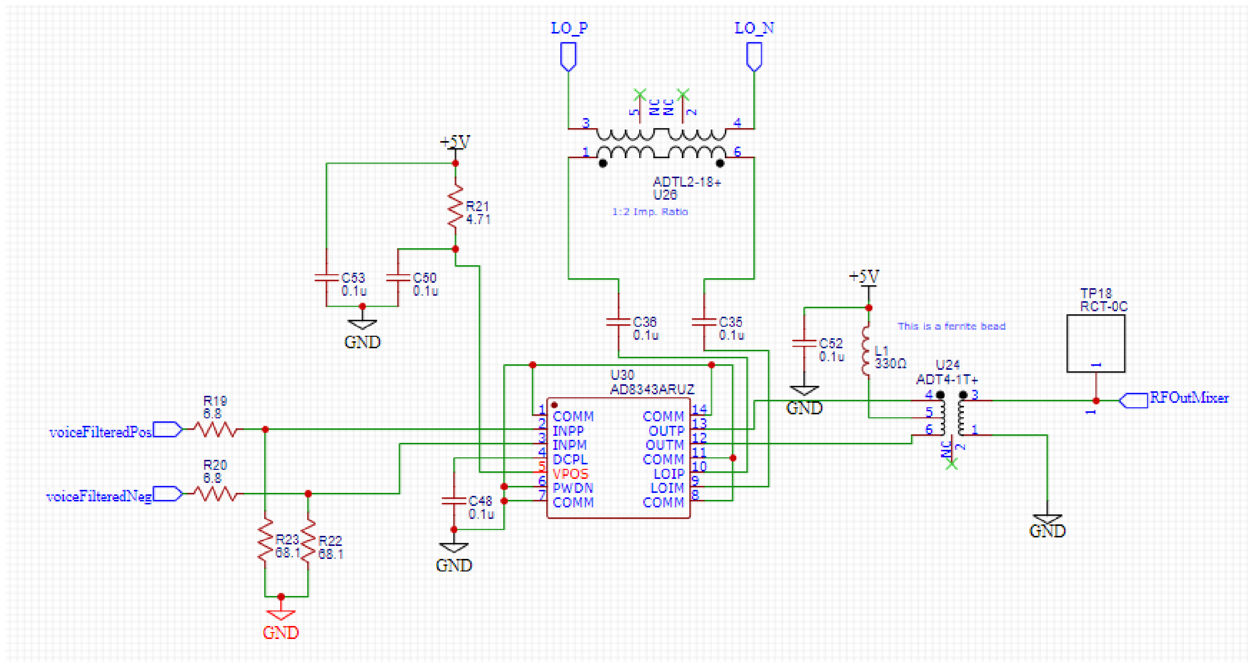


**Figure 8: Mixer & Supporting Circuitry**

## 2.4.4 Local Oscillator

To generate the carrier wave that the mixer will use to modulate and demodulate the audio signal being transmitted or received, the design team selected the Si570 local oscillator manufactured by Silicon Labs. This oscillator was chosen because of its ability to output across the entire frequency range specified in Section 2.1.1. It outputs the carrier wave as a differential signal which is then used as an input to the mixer in the receive and transmit circuits.

## 2.4.5 Push-To-Talk

The radio design also includes Push-To-Talk circuitry consisting of a button that connects either the radio's output to the output of the receiver or the radio's input to the input of the transmitter. Essentially, the Push-To-Talk button connects either the receiver or the transmitter to the I/O ports they need to perform their function.

## 2.4.6 Voltage Regulators

To convert the 28 Volt DC power supply from the airplane into the voltages necessary for the radio's operation, the design includes voltage regulators capable of converting any voltage below 33 Volts into a constant 5 Volt output. This 5 Volt output powers many of the components on the board which require a 5 Volt supply.

In addition to the 5 Volt regulator, the design also includes two additional regulators: one which converts 5 Volts to a constant 3.3 Volt output to power the local oscillator and the lower side of

the oscillator's level converter, and another which converts the 5 Volt supply into a constant -5 Volt output which powers the lower rail of the op amps in the filter bank.

### 2.4.7 Level Shifters

Because several digital components on the board use different levels of digital logic, level shifters were necessary to convert between 5 Volt and 3.3 Volt digital signals. For example, the I2C signals coming from the Arduino use a 5 Volt logic level, but the local oscillator operates from a 3.3 Volt supply, so to avoid damaging the local oscillator, level shifters are necessary to transform the digital SCL and SDA signals comprising the I2C bus from 5 Volts to 3.3 Volts.

## 2.5 PCB Design

After reaching the final iteration of the design outlined in Section 2.3 through extensive testing, it became necessary to translate the schematic for the project's final design into a PCB layout.

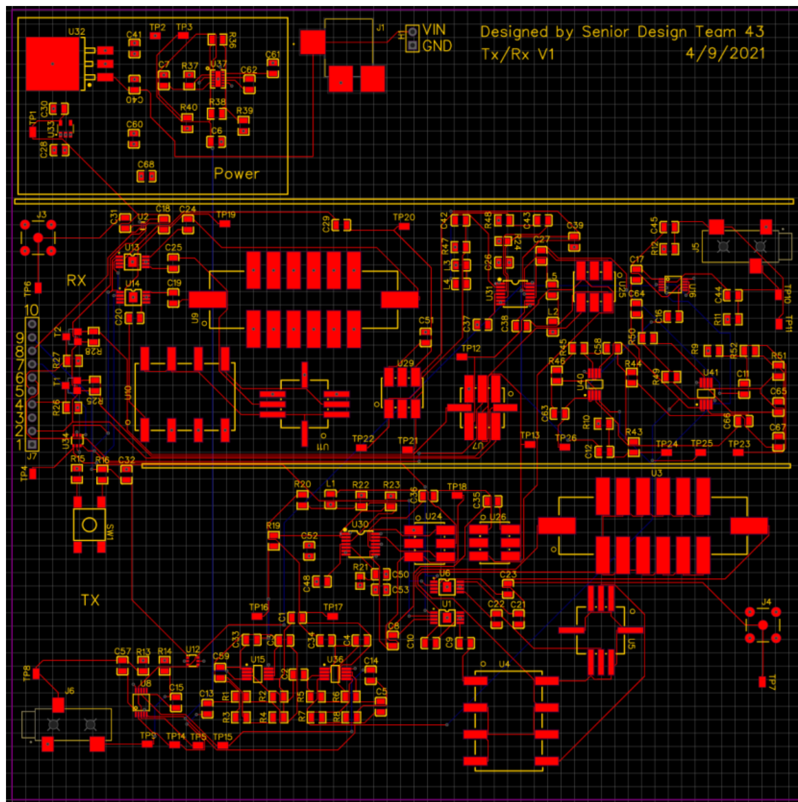


**Figure 9: Final PCB Layout**

The PCB was designed as a four-layer board to reduce the high frequency signals' exposure to electromagnetic interference. The top layer is a high frequency plane containing the board's high frequency signals. Directly under the high frequency plane is an unbroken ground plane which allows signals to return to ground as quickly as possible since it can connect directly to component grounds with vias. Under the ground plane is an unbroken power plane supplying five volts of DC power. This layer is mainly for convenience and allows vias to directly connect components' power pins to the power plane. The bottom plane consists of extra space for routing

low frequency traces such as control signals from the Arduino. This allows for more efficient routing of the high frequency traces on the top plane, keeping them shorter and thus less susceptible to electromagnetic interference.

Several other considerations influenced the PCB's design, chief among them the efficiency and logicality of its component placement and signal routing. The PCB's maximum size is defined in the project's requirements as 6" x 6". This space was partitioned into sections for the receiver, transmitter, and various power supplies. This kept the board's high frequency traces as short and organized as possible by maintaining proximity between connected components.

With respect to routing, manual trace placement was impractical because of the sheer number of necessary connections. As such, an auto-router was used to route traces automatically. This was done in several stages. First, the auto router routed the high frequency traces on the high frequency plane. Once these traces were routed, the auto router was then used to connect all of the board's other nets except for the ground and power nets. A few traces needed to be manually routed because the auto router was unable to route them. Finally, nets on the ground and power planes were directly connected to their respective pads, completing the routing.

After the PCB design was finished, it was run through a design rule check (DRC) to ensure that it fit the fabricators' specifications. After passing DRC, the design was used to generate the Gerber files which were sent to be printed into the final board.

## 2.6 Security Concerns & Countermeasures

### 2.6.1 Physical Security

With respect to the confidentiality of the voice signal that the radio transmits, the project's client specifically requested that the radio implement unencrypted voice communications, so security was not a concern regarding the radio's modulation, transmission, and reception of an audio signal.

Since the radio's intended operating environment is an airplane cockpit, the design team assumed that airlines and other potential customers will have physical security measures in place to secure these cockpits from intruders. This preexisting physical security should suffice to stop attackers from physically accessing the radio and/or its interface.

### 2.6.2 Cybersecurity

Regarding the communication between the radio and its user interface, SNMP version 1, the protocol that the interface uses to communicate with the radio, does not provide encrypted transfer of data. This poses a mild security risk, but it is mitigated by the fact that the radio and the user interface will communicate via a cabled ethernet connection as opposed to a wireless one. This means that an attacker would need access to the physical ethernet cabling connecting the radio and the computer running its user interface in order to intercept their communications. Furthermore, the only information that could be obtained from sniffing traffic between the radio and its interface would be the radio's operating frequency and its status

as either a receiver or transmitter. An attacker's ability to view this information seems unthreatening compared to other damage they could potentially inflict if they had the prerequisite physical access to a plane's avionics. Thus, the design team concluded that the radio's communication with its user interface is sufficiently secure if it occurs over a cabled Ethernet connection as intended.

If a user wishes to connect the radio and its user interface wirelessly, this design provides a clear path to upgrade from the unencrypted SNMP version 1 protocol to SNMP version 3 which does encrypt network traffic. This would not entail changing the MIB for the SNMP software, only the formation and reception of packets by the radio's microcontroller and user interface software. Encrypting traffic between the radio and its user interface using SNMPv3 would mitigate any risk associated with an attacker intercepting the radio's wireless control signals because the attacker would need to break the encryption to either hijack the radio or make sense of its control data.

# 3 Implementation

After finalizing the theoretical radio design, the design team began the process of implementing it as a printed circuit board (PCB) governed by a microcontroller interacting with software running on a desktop PC. This involved several steps elaborated below, including:

- Assembling components into a functional PCB
- Writing Arduino code to control the PCB's local oscillator
- Customizing preexisting software to provide a user interface for the Arduino
- Integrating the PCB with the microcontroller & software interface

## 3.1 PCB Assembly

The first step in the implementation process was to prepare the PCB for soldering in a reflow oven. Solder paste was spread over the circuit board using a stamped metal stencil with holes directly over the pads. This ensured that solder paste covered each pad without spreading to the rest of the board. Each component compatible with the oven was then carefully placed onto a pad after a team member ensured it had the correct orientation.

Once all components were correctly placed, the whole assembly was placed in the reflow oven. The reflow oven heats up according to a preset pattern tailored for the solder paste used. This allows the solder paste to melt cleanly and evenly before eventually hardening on each component's pad. This process quickly and efficiently soldered many components to the PCB that would have been extremely difficult to solder by hand.

One of the downsides of using the reflow oven is that if too much solder gets applied to pads in close proximity, the heating process can connect the solder and create a short circuit.
These unintended shorts need to be identified and fixed before the board can be used. In most cases, a visual check was sufficient to identify short circuits. Team members used a strong magnifying glass to examine each integrated circuit mounted on the PCB. In most cases it was easy to see when solder connected separate pads. In addition to the magnifying glass, a multimeter also helped to identify unintended component shorts. If the multimeter revealed a resistance close to zero Ohms between adjacent pads, the PCB's layout was referenced to determine if the pins in question were meant to be directly connected. If not, the reading indicated the presence of a short. Team members needed to fix each unintended short by applying flux to the shorted pads before using a soldering iron to heat the solder connecting them. Each time the solder was heated, some of it stuck to the soldering iron and was removed from the pads. A team member would repeat this process until the resistance between the pads was no longer negligible.

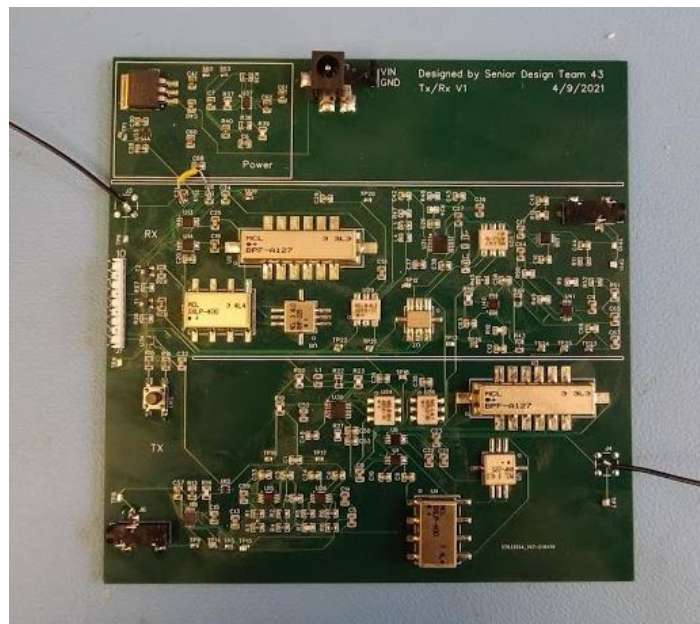


**Figure 10: Final PCB**

Once all the unintended shorts were fixed, several components still needed to be manually soldered to the board. This owed primarily to the fact that several components were heat sensitive and could not tolerate the sustained high temperatures in the reflow oven. Furthermore, several other components did not arrive in time and required that suitable replacements be soldered on in their place.

## 3.2 Local Oscillator Programming

One of the parts on the PCB – the local oscillator (LO) – requires programming via an inter-integrated circuit (I2C) bus to specify its output frequency. The oscillator's performance is

critical to the radio since its output frequency specifies the channel that the radio will receive or transmit on, and configuring the oscillator via I2C is not a trivial task.

Establishing I2C control over the local oscillator required a device capable of transmitting commands via an I2C bus. To fulfill this task, the design team chose an Arduino Uno because of its simplicity and ease of use. An open-source software library named Wire.h already provides compact functions that an Arduino programmer can use to send and receive information using the I2C protocol. The Arduino Uno board also contains designated pins for the serial data line (SDA) and serial clock line (SCL) wires comprising the I2C bus as well as the pull-up resistors necessary for the bus's operation.

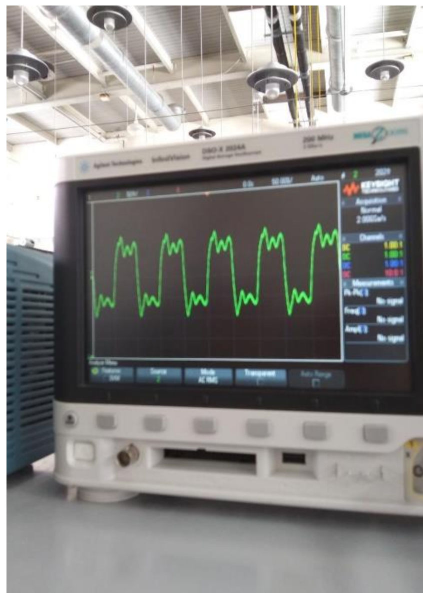

**Figure 12: LO Oscilloscope Output**

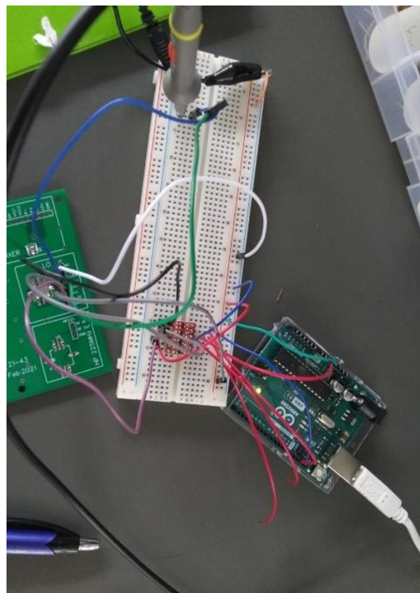

**Figure 11: LO Test Circuit**

After choosing the Arduino Uno as the microcontroller used to communicate with the oscillator, it became necessary to program the specific oscillator chosen for the PCB, the Si570, according to instructions in its datasheet. This involved calculating several parameters based on the desired output frequency and then writing them to the appropriate registers in the oscillator via an I2C bus.

One issue that arose while programming the oscillator was that a formula in the its datasheet required one of the oscillator's floating-point parameters to be stored in integer form after being multiplied by $2^{28}$. This created an enormous result that was too big for the Arduino's floating point ALU to calculate. To resolve this issue, the design team converted each digit in the fractional part of the original floating-point parameter to an integer by multiplying by a power of ten, multiplying the resulting integer by $2^{28}$, and then converting the result back to floating point form by dividing out the power of ten that was originally used to shift each digit. The products of each digit in the floating-point value and $2^{28}$ were then summed up to calculate the value of

the original parameter multiplied by $2^{28}$. This process produced the value specified in the oscillator's datasheet by circumventing the Arduino's impotent floating point ALU with integer multiplication.

After following the instructions in the oscillator's datasheet, the design team developed source code capable of calculating the parameters necessary to set the local oscillator's output to an arbitrary value and subsequently writing those parameters to the appropriate registers over an I2C bus. The Arduino code used to control the local oscillator is included in Appendix IV.

### 3.3 SNMP User Interface

The design team adapted an open-source library called "Agentuino" to implement SNMP control of the Arduino. This library establishes a Management Information Base (MIB) on the Arduino that stores the radio's current operating frequency and its status as a transmitter or receiver. It also allows the Arduino to support SNMP get and set requests that allow a remote host to transmit information to and from the MIB, making the Arduino (and, by extension, the radio) programmable via an SNMP connection.

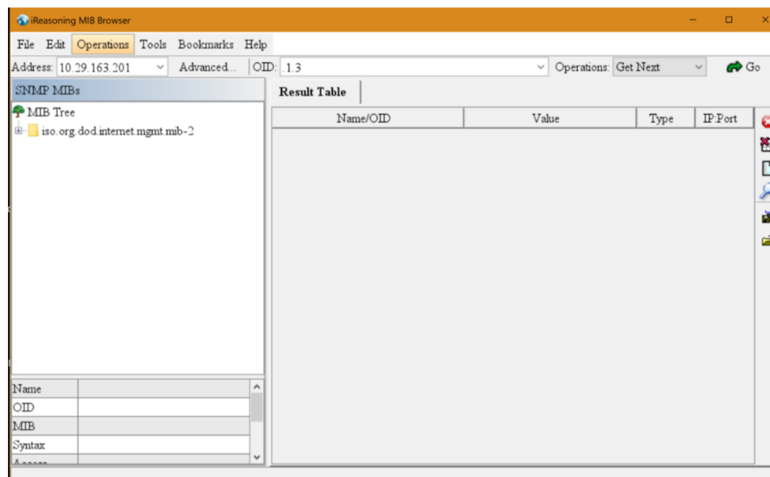


**Figure 13: MIB Browser User Interface**

The MIB stored on the Arduino contains object identifiers (OIDs) for variables to store the local oscillator's output frequency and the radio's status as a transmitter or receiver. This allows another device on the same local area network (LAN) as the Arduino to query the radio's operating state and change its operating frequency. Open-source software called iReasoning MIB Browser (iRMIBB) was chosen to interact with the Arduino's MIB from a remote host. The design team chose iRMIBB because of its simplicity and availability.

One issue that emerged while establishing SNMP control of the Arduino was that Agentuino only supported sending and receiving variables up to 8 bits in size. The radio operates across a wide range of frequencies with high resolution, making it difficult to encode a frequency using only 8 bits. To resolve this issue, the team configured iRMIBB and Agentuino to communicate using strings of characters as opposed to single integers for set and get requests. These strings

could then be converted into floating point variables on the Arduino using a simple type cast, allowing a user to specify any frequency within the radio's operational band using a set or get request on iRMIBB.

### 3.4 Integration

After successfully creating a working PCB, code to program the PCB's oscillator with an Arduino, and a user interface to control the Arduino using SNMP, it became necessary to integrate these separate design units into a functional product. To accomplish this, cabled Ethernet connections were established between the Arduino, a host PC running MIB Browser, and a common LAN. This allowed MIB Browser to send get and set requests to the Arduino's MIB using the Arduino's IP address. The Arduino could then use the frequency specified in its MIB to calculate the parameters necessary to make the local oscillator output at that frequency before writing those values to the oscillator over the I2C bus. After the Arduino successfully programs the oscillator, the Push-To-Talk button will determine if the PCB's receive or transmit hardware is engaged to either modulate or demodulate an audio signal. Thus, together the three functional blocks form a working radio.

# 4 Testing Process & Results

### 4.1 General Testing Plan

The design team created a plan to effectively test the performance of various components as well as the system as a whole. The top-level overview of the updated plan is as follows:

- Develop and order PCBs to test each specific component
- Test each individual component (mixer, LO, Arduino, filters, etc.) in isolation and verify that they satisfy the requirements for the final design.
- Develop and order a PCB that implements the final design.
- Test the performance of the individual components in the final circuit to ensure successful integration
- Test the performance of the radio transceiver as that of a "black box" (simply comparing the output with the expected signal for a given input)

### 4.2 Component Testing & Results

Throughout the project, testing was severely hampered by restrictions associated with the ongoing COVID-19 pandemic. Minimal access to lab equipment and restricted lab hours made it extremely hard to comprehensively test several components as well as the fully integrated design. For example, the design team was never provided with the equipment to measure RF signals at or above 300 MHz even though the project's requirements specify that the radio should be able to transmit and receive at frequencies up to 400 MHz. As such, the following results

reflect a simple, qualitative test of the full radio as opposed to a thorough, quantitative analysis, but the design team is confident that more comprehensive testing performed with the necessary equipment could verify the radio's correct operation.

For individual component testing, a development PCB was designed and ordered which simply contained the footprint for each component in isolation, allowing them to be tested outside the context of the full system.

The following list of parts that were tested is not comprehensive because many facets of the design were changed after the development board was ordered, making it impossible for space to be reserved on it for new and replacement components.

Other parts such as the Push-To-Talk system simply were not well-suited to unit testing because they required the fully integrated radio design to operate.

### 4.2.1 High Frequency Filters

The three high frequency filters were tested using the waveform from the local oscillator as an input. The local oscillator was chosen as the filters' test input because of the complete lack of alternative test equipment. Each filter was tested with frequencies in its pass band and stop band.

Tests for the low pass filters were inconclusive due to the inadequate technology involved. They did allow low frequencies to pass as expected, but their stop band could not be tested because the oscilloscopes in the lab could not accurately measure any signal above 400 MHz.

The limited testing that could be completed for the band pass filters indicated that they functioned as expected. The BPFs with a pass band from 118 MHz to 137 MHz were tested using frequencies of 90 MHz, 126 MHz, and 180 MHz. The 126 MHz signal was not attenuated significantly while the 90 MHz signal and 180 MHz signal were in fact attenuated as they should have been.

Similar to the band pass filters, extensive testing could not be completed for the high pass filters, but the experiments that were performed indicated that they worked correctly. The high pass filters had a pass band above 225 MHz, so testing them involved using a 100 MHz signal, 180 MHz signal, and a 230 MHz signal. The 180 MHz and the 230 MHz signal had minimal attenuation while the 100 MHz signal was in fact attenuated. Ideally, the 180 MHz signal should have been attenuated, but this was not the case. These test results indicate that the band pass filters had the correct corner frequency, 225 MHz, but the results at 180 MHz indicated that the filter was not as strong as it ideally should have been.

### 4.2.2 Mixer

The initially selected mixer, the LT5510, had connections that were exposed as header pins for easier access on the development PCB and interfaced to a breadboard. The breadboard in turn provided all of the necessary passive components for the proper operation of the mixer, along with power and input signal voltages. The LO signal was provided to the breadboard by the Si570 LO, which by default outputs a differential square wave at 10 MHz. The first attempted test was designed to demonstrate LT5510's ability to down-convert and demodulate AM signals: it used a 10 MHz carrier wave from the LO to modulate a 600 Hz sine wave from a function

generator. Unfortunately, the test results were inconclusive: the mixer did not produce output. The team concluded that this behavior resulted from insufficient biasing for the open-collector outputs of the transistor-based mixer. Closer inspection of the datasheet revealed a deficiency in the test setup, specifically a lack of the proper inductance and capacitance values in the supporting circuitry for the mixer. The original test setup was deficient because the device's datasheet was misleading and did not properly specify that the part was exclusively a downconverter – it was presented as a general-purpose RF mixer.

These tests led the design team to replace the ineffective mixer with a proper RF mixer that supported both up- and down-conversion.

### 4.2.3 Local Oscillator

Once the code to control the local oscillator had been completed, it was used to set the oscillator's output frequency.  To verify that the output waveform matched the expected one, an oscilloscope was used to observe the differential output signal from the oscillator.  The oscilloscope used for testing was limited to a maximum frequency of 200 MHz, so the oscillator was programmed to output at 10 MHz, and then its output frequency was iteratively increased by 10 MHz until it reached the 200 MHz maximum that the oscilloscope could measure. For each 10 MHz increment between 10 and 200 MHz, the actual output frequency differed from the expected value by less than 5%, and the differential peak-to-peak amplitude was within the acceptable range of 2 to 3 volts.

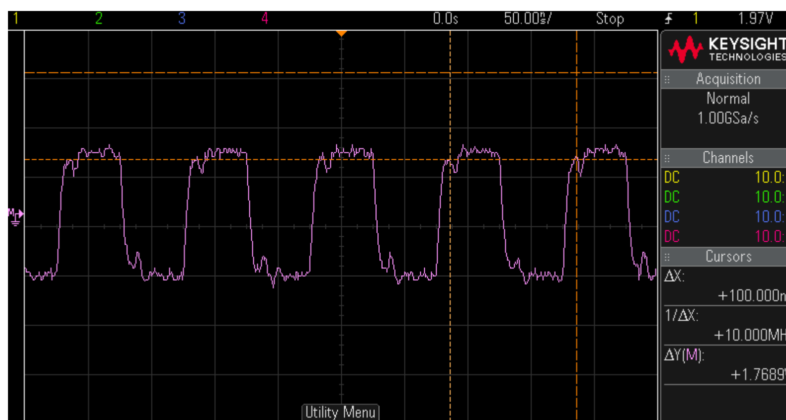


**Figure 14: Differential LO Output**

### 4.2.4 MIB Browser and Agentuino

Initial testing for the SNMP control software started with a simple pushbutton circuit wired to a pin on the Arduino.  The status of this pin was stored in the MIB on the Arduino as a variable used to test the SNMP get command.  After deploying the modified Agentuino library on the Arduino, MIB Browser was able to read the value of the variable storing the button's state with a get command.

After the get command's correct operation was established, the SNMP set command was also tested to make sure it worked properly with MIB Browser and Agentuino.  This was done by

running a test script on the Arduino that repeatedly printed a value in its MIB to a serial monitor before attempting to change that MIB variable with the set command. The serial monitor showed that a set command in MIB Browser successfully changed the corresponding value in the Arduino's MIB

The next step in the testing process was to test the set Command. The goal was to change the value of a variable in the MIB using MIB Browser and to display it on the serial monitor. After ensuring that the Community Name field in MIB Browser was set to "PUBLIC," the set command executed successfully.

**Figure 15: Testing Set Command in MIB Browser**

### 4.2.5 Miscellaneous Parts

Other miscellaneous parts such as the multiplexers and demultiplexers were also tested to confirm they would work as expected at high frequencies. When testing these parts, the team first verified that they worked as expected at low frequencies before testing them at the RF frequencies present in the design to verify that their operation didn't change. The equipment available for testing did not allow for testing the full range of operational frequencies, so they were only exercised over the lower portion of the frequency band.

The main components tested in this manner were the multiplexers. Multiplexer testing consisted of supplying one input with a signal, starting with a low frequency of around 1 kHz, and leaving the other input disconnected. Correct operation was verified by observing the output when the enable pin was set to high or low with the output expected to change depending on the value of the enable pin. This test was performed at the highest frequencies the signal generator could output, namely 25MHz. The demultiplexers were tested with a similar process.

## 4.3 System Testing & Results

As with unit testing, system testing was severely hampered by restrictions associated with the ongoing COVID-19 pandemic. As such, the following procedure and results are not a

comprehensive test of the radio, but instead consist of a simple check attempting to verify its correct operation because the available equipment did not allow for anything better.

The final system-level testing broadly consisted of two parts: testing the signal chains of the receiver and transmitter individually, and testing the transmitter and receiver simultaneously using two separate final boards. During the first part of this testing process, an audio signal was inputted into the transmitter, modulated using a VHF carrier, and observed on an oscilloscope after it was filtered through a VHF bandpass filter. The transmitter's output shown in Figure 16 matched the modulated signal we were expecting to observe.

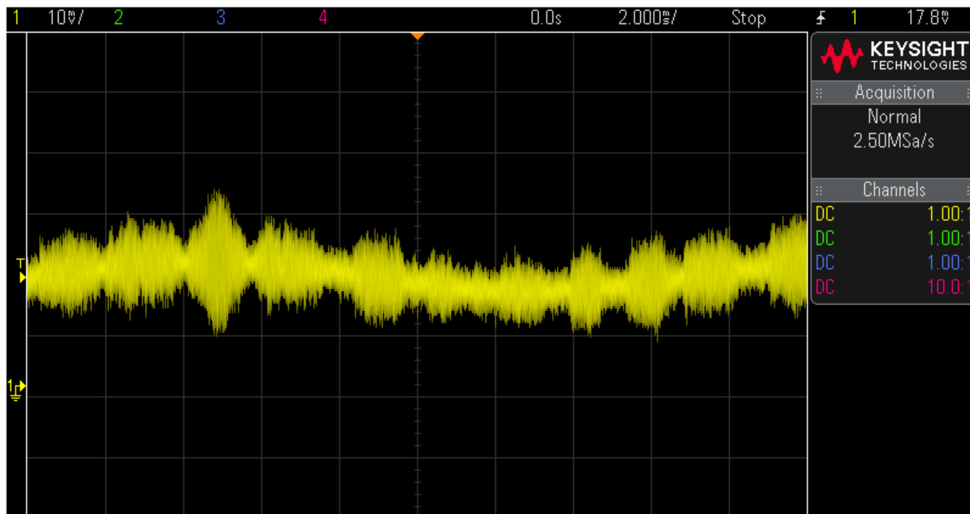


**Figure 16: Modulated Transmitter Output**

The signal generator available in the lab was not capable of generating an AM-modulated signal with a carrier frequency beyond 25 MHz, making it insufficient for the second half of testing. Therefore, it was decided to use one board as a transmitter, the other as a receiver, and have the two boards communicate with each other. During this process it was discovered that the output RF filter bank on the transmitter board was damaged; therefore, the decision was made to connect the receiver board's RF input directly to the transmitter board mixer's RF output, bypassing the transmitter's RF output filter bank. After this modification, the transmitter was able to successfully modulate audio output from a desktop computer using a 120 MHz carrier, and the receiver was able to successfully demodulate the signal and output it to the same computer where it was saved as an MP3 file. The final recording was clearly recognizable as the initial audio file, a recording of Rick Astley's iconic "Never Gonna Give You Up," with some additional noise and attenuation resulting from improper RF output filtration and subpar board-to-board RF connections.

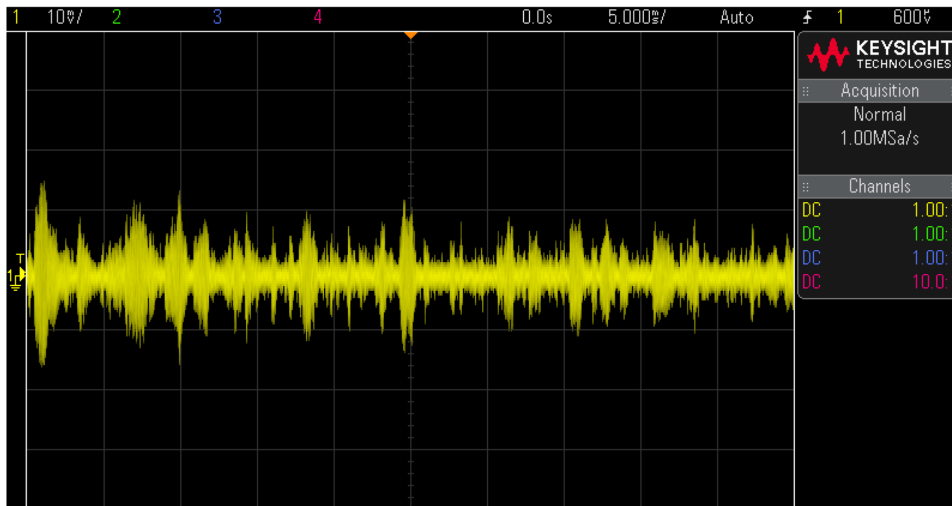


**Figure 17: Modulated Music Broadcasted by the Transmitter**

# 5 Context: Related Products & Literature

This project implements the transmit and receive hardware for a backup radio as well as software to establish remote control of the radio from a host PC. Since it does not entail designing a full radio, it does not compare to any preexisting products. As such, any comparison of this design to commercially available radios will pertain only to a hypothetical complete radio that includes the receive/transmit hardware and software interface that this project produced.

Since this project produced a radio intended for military as well as commercial use, very few related projects have documentation available in the public domain. One radio that does have similar specifications to this project is the SOVERON®AR radio produced by Rohde and Schwarz [1]. The Rohde and Schwarz product is much more complex than this project since it functions as an aircraft's main radio instead of as a backup and covers a wider frequency band. However, both this project and the SOVERON®AR radio share the same civil standards as well as the generic 28 V DC power input available on most aircraft.

Other backup radios generally come in the form of small handheld devices intended for smaller aircraft. Most of these handheld radios are not made for military use so they do not cover the UHF frequency band that this project does, but they do cover its VHF frequency requirement. Handheld radios such as the ones produced by Leonardo are also much simpler than this project in that they have no SNMP control interface and are typically powered by batteries as opposed to the airplane's 28 V DC supply [2].

In short, this project occupies a unique niche in the radio market. The radio that emerged from this project boasts more features and a larger frequency band than a typical backup radio, but it is still simpler and easier to use than a device intended as an aircraft's main radio. The unique balance between this project's robust, military-grade operating requirements and simple, streamlined design position it to present a wide range of customers with an appealing alternative to conventional handheld backup radios.

# Appendix I: Operation Manual

## Before Starting

Open-source user software known as MIB Browser controls the backup radio by interacting with code running on the Arduino (see Appendix IV for details). Before attempting to set the radio's receive and/or transmit frequency via MIB Browser or read the radio's configuration information, the user should ensure that they have MIB Browser downloaded on a computer connected to the same local area network (LAN) as the Arduino controlling the circuit. The Arduino should connect to the LAN via an Ethernet shield. The user should also use a serial cable to flash the source code "SNMP_LO.ino" onto the Arduino after adding the libraries defined at the top of this file to the build path in the Arduino IDE. Appendix IV contains SNMP_LO.ino.

It is critical before attempting to program the radio with MIB Browser that the parameters for the MAC address of the Arduino's ethernet shield and the IP address of the Arduino be updated in the source code file "LO_SNMP.ino." The Arduino's IP address can be arbitrarily assigned if the user is an administrator on the LAN being used and the assigned address does not conflict with the address of another device on the network. If the user is not an administrator on the network, a simple script named "DHPC.ino" has been included in Appendix IV that can be flashed onto the Arduino to automatically request an IP address using DHCP (Dynamic Host Configuration Protocol) and print it to the Arduino's serial line for the user to view.

## Controlling the Radio

### Setup

1. Having established cabled ethernet connections on a common LAN between the Arduino and a computer with MIB Browser installed, the user should open MIB Browser on the said computer.

2. In MIB Browser, the user should type the Arduino's IP address in the address bar shown in Figure 18.

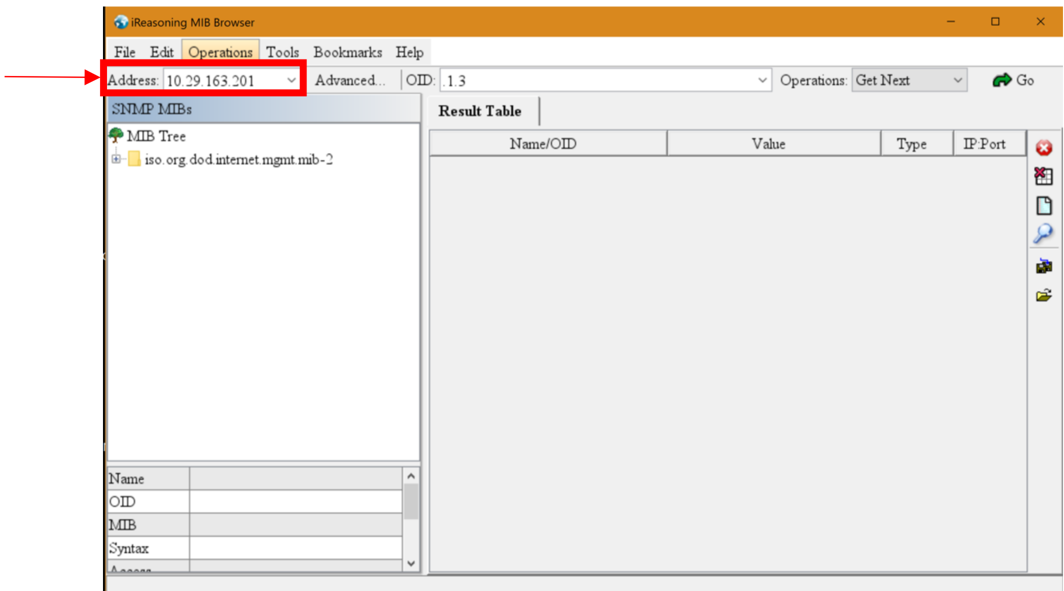


**Figure 18: MIB Browser Interface**

## Reading Configuration Information

1. To read the radio's current operating configuration, the user should first enter the OID of the variable they wish to read in the "OID" field highlighted in Figure 19. The OID for the radio's operating frequency is 1.3.6.1.4.1, and the OID for the radio's operational state (either transmit or receive) is 1.3.6.1.4.2. (In OID 1.3.6.1.4.2, a value of 1 indicates that the radio is transmitting, and 0 indicates that it is receiving). Please note that these are the default OIDs used to store these variables in SNMP_LO.ino, but if the user has

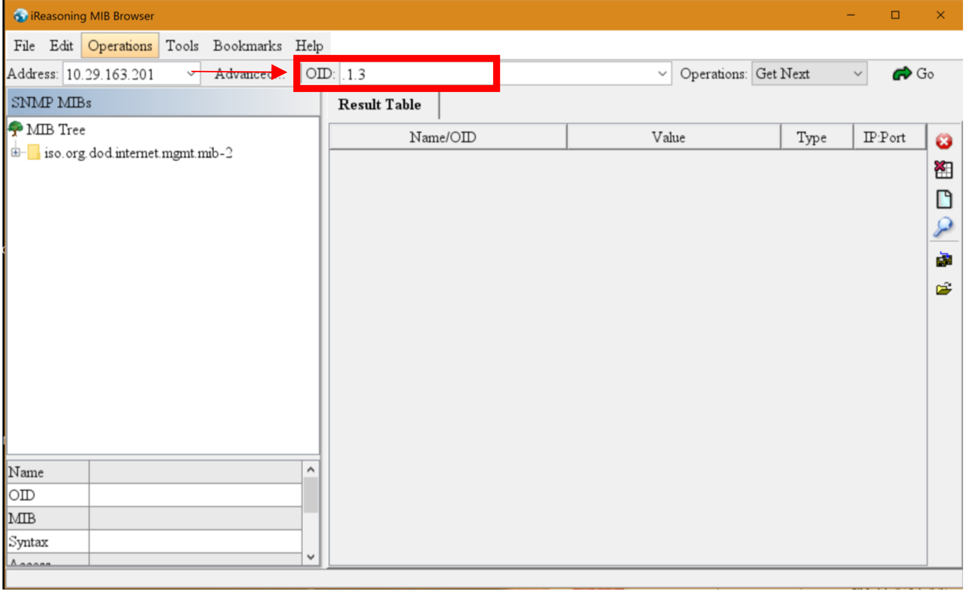


**Figure 19: MIB Browser Interface**

reconfigured the MIB defined in this file then the appropriate user defined OID should be entered in MIB Browser's OID field instead.

2. Click the drop-down menu for "Operations" and select "Get" from the list of commands.

3. Click "Go." If everything is configured correctly, a new entry should appear in the Result Table with the OID and value of the entry that the user requested from the MIB.

### Writing Configuration Information

1. To configure the radio's operating frequency, the user should first enter 1.3.6.1.4.1, the OID of the frequency variable in the MIB, in the "OID" field on MIB Browser shown in Figure 19. Please note that this is the default OID used to store frequency in SNMP_LO.ino, but if the user has reconfigured the MIB defined in this file then the appropriate user defined OID should be entered in MIB Browser's OID field instead.

2. Click the drop-down menu for "Operations" and select "Set" from the list of commands.

3. Click "Go." If everything is configured correctly, the target OID should be updated to its new value in the MIB. The user can attempt to read the frequency value back to MIB Browser in order to confirm that the radio was successfully configured.

### Push To Talk

Pressing the Push-To-Talk button on the radio will toggle between transmitting and receiving modes.

1. If the button is pressed, the radio is in transmit mode and will broadcast any inputted audio signal at its current operating frequency.
2. If the button is unpressed, the radio is in receive mode and will output any audio signal it receives at its current operating frequency.

# Appendix II: Alternative Designs

No alternative designs were considered for the radio. The plan was always to implement a direct conversion architecture.

# Appendix III: Other Considerations

To become civilly certified, this radio will need to pass testing that demonstrates its ability to perform well in the presence of extreme temperatures and other external factors. Since the resources to test these conditions were not available during this project, these aspects of the radio's operation were not tested, but future continuations of the project will need to adjust the design as necessary to ensure that it meets civil certification standards.

Other facets of a full radio design that this project did not involve such as a power supply card and an antenna will also need to be designed and implemented to integrate this project's receive/transmit hardware and software interface into a complete radio.

# Appendix IV: Code

### SNMP_LO.ino

```
#include <Streaming.h>        // Include the Streaming library
#include <Ethernet.h>         // Include the Ethernet library
#include <SPI.h>
#include <MemoryFree.h>
#include <Agentuino.h>
#include <Flash.h>
#include <string.h>
#include "LO.h"
#include <Wire.h>

#define LO_I2C_ADDR 0x75
#define Fxtal 114.2  //MHz (old 570)
//#define Fxtal 94.86 //MHz (new 570)

static byte mac[] = { 0xA8, 0x61, 0x0A, 0xAE, 0x70, 0x58};  //Mac Address for Arduino Ethernet
Shield (Abbey's Arduino)
//static byte ip[] = { 10, 29, 163, 201};               //IP Address for Arduino Ethernet Shield
(Abbey's Arduino - School)
static byte ip[] = {192, 168, 1, 2};               //IP Address for Arduino Ethernet Shield (Abbey's
Arduino - School)
//static byte mac[] = { 0xA8, 0x61, 0x0A, 0xAE, 0x70, 0x58};  //Mac Address for Arduino Ethernet
Shield (Bens's Arduino)
//static byte ip[] = { 10, 29, 163, 201};               //IP Address for Arduino Ethernet Shield (Ben's
Arduino)

int32_t *relayPointer;
static char freq_str[10] = "0.00";
double freq_num;

// RFC1213-MIB OIDs
// .iso (.1)
// .iso.org (.1.3)
// .iso.org.dod (.1.3.6)
// .iso.org.dod.internet (.1.3.6.1)
// .iso.org.dod.internet.mgmt (.1.3.6.1.2)
```

```
// .iso.org.dod.internet.mgmt.mib-2 (.1.3.6.1.2.1)
// .iso.org.dod.internet.mgmt.mib-2.system (.1.3.6.1.2.1.1)
// .iso.org.dod.internet.mgmt.mib-2.system.sysDescr (.1.3.6.1.2.1.1.1)
// .iso.org.dod.internet.mgmt.mib-2.system.sysContact (.1.3.6.1.2.1.1.4)
// .iso.org.dod.internet.mgmt.mib-2.system.sysName (.1.3.6.1.2.1.1.5)
// .iso.org.dod.internet.mgmt.mib-2.system.sysLocation (.1.3.6.1.2.1.1.6)
// .iso.org.dod.internet.mgmt.mib-2.system.sysServices (.1.3.6.1.2.1.1.7)
const char sysDescr[] PROGMEM     = "1.3.6.1.2.1.1.1.0";  // System Description
const char sysContact[] PROGMEM   = "1.3.6.1.2.1.1.4.0";  // System Contact
const char sysName[] PROGMEM      = "1.3.6.1.2.1.1.5.0";  // System Name
const char sysLocation[] PROGMEM  = "1.3.6.1.2.1.1.6.0";  // System Location
const char sysServices[] PROGMEM  = "1.3.6.1.2.1.1.7.0";  // System Services


//My Custom OID's
const char FREQUENCY[]        PROGMEM    = "1.3.6.1.4.1";    //Frequency
const char TX_RX                          PROGMEM      = "1.3.6.1.4.2";                    //tx/rx
state

// RFC1213 local values
static char locDescr[]          = "SNMP V1 Button and LED test";            // read-only (static)
static char locContact[50]       = "amwilder@iastate.edu";
static char locName[20]          = "Abbey Wilder";
static char locLocation[20]       = "Iowa State University";
static int32_t locServices        = 6;                              // read-only (static)

uint32_t prevMillis = millis();
char oid[SNMP_MAX_OID_LEN];
SNMP_API_STAT_CODES api_status;
SNMP_ERR_CODES status;

void pduReceived()
{
  SNMP_PDU pdu;
  api_status = Agentuino.requestPdu(&pdu);
  //
  if ((pdu.type == SNMP_PDU_GET || pdu.type == SNMP_PDU_GET_NEXT || pdu.type ==
SNMP_PDU_SET)
    && pdu.error == SNMP_ERR_NO_ERROR && api_status == SNMP_API_STAT_SUCCESS ) {
    //
    pdu.OID.toString(oid);
```

```
if ( strcmp_P(oid, sysDescr ) == 0 ) {
 // handle sysDescr (set/get) requests
 if ( pdu.type == SNMP_PDU_SET ) {
  // response packet from set-request - object is read-only
  pdu.type = SNMP_PDU_RESPONSE;
  pdu.error = SNMP_ERR_READ_ONLY;
 } else {
  // response packet from get-request - locDescr
  status = pdu.VALUE.encode(SNMP_SYNTAX_OCTETS, locDescr);
  pdu.type = SNMP_PDU_RESPONSE;
  pdu.error = status;
 }
} else if ( strcmp_P(oid, sysName ) == 0 ) {
 // handle sysName (set/get) requests
 if ( pdu.type == SNMP_PDU_SET ) {
  Serial.println("sysName Entered Set If");
  // response packet from set-request - object is read/write
  status = pdu.VALUE.decode(locName, strlen(locName));
  pdu.type = SNMP_PDU_RESPONSE;
  pdu.error = status;
 } else {
  // response packet from get-request - locName
  status = pdu.VALUE.encode(SNMP_SYNTAX_OCTETS, locName);
  pdu.type = SNMP_PDU_RESPONSE;
  pdu.error = status;
 }
 //
} else if ( strcmp_P(oid, sysContact ) == 0 ) {
 // handle sysContact (set/get) requests
 if ( pdu.type == SNMP_PDU_SET ) {
  // response packet from set-request - object is read/write
  status = pdu.VALUE.decode(locContact, strlen(locContact));
  pdu.type = SNMP_PDU_RESPONSE;
  pdu.error = status;
 } else {
  // response packet from get-request - locContact
  status = pdu.VALUE.encode(SNMP_SYNTAX_OCTETS, locContact);
  pdu.type = SNMP_PDU_RESPONSE;
  pdu.error = status;
 }
 //
} else if ( strcmp_P(oid, sysLocation ) == 0 ) {
```

```
  // handle sysLocation (set/get) requests
  if ( pdu.type == SNMP_PDU_SET ) {
    // response packet from set-request - object is read/write
    status = pdu.VALUE.decode(locLocation, strlen(locLocation));
    pdu.type = SNMP_PDU_RESPONSE;
    pdu.error = status;
  } else {
    // response packet from get-request - locLocation
    status = pdu.VALUE.encode(SNMP_SYNTAX_OCTETS, locLocation);
    pdu.type = SNMP_PDU_RESPONSE;
    pdu.error = status;
  }
  //
} else if ( strcmp_P(oid, sysServices) == 0 ) {
  // handle sysServices (set/get) requests
  if ( pdu.type == SNMP_PDU_SET ) {
    // response packet from set-request - object is read-only
    pdu.type = SNMP_PDU_RESPONSE;
    pdu.error = SNMP_ERR_READ_ONLY;
  } else {
    // response packet from get-request - locServices
    status = pdu.VALUE.encode(SNMP_SYNTAX_INT, locServices);
    pdu.type = SNMP_PDU_RESPONSE;
    pdu.error = status;
  }
  //
}

else if ( strcmp_P(oid, FREQUENCY) == 0 ) // Set Frequency
{
  // handle sysName (set/get) requests
  if ( pdu.type == SNMP_PDU_SET )
  {

    status = pdu.VALUE.decode(freq_str, strlen(freq_str));

    //Convert from string to double
      freq_num = atof(freq_str);

    //generate N1, HS_DIV
      int F_Params[2];
      Gen_Params(F_Params, freq_num);
```

```
     //Calculate Fdco and FRFREQ
      double Fdco = (freq_num * (double)(*F_Params) * (double)(*(F_Params + 1)));
      double RFREQ = Fdco / Fxtal;

     //Convert N1, HS_DIV, and RF_REQ to values to be written to the oscillator
      uint32_t N1_reg = N1_Lookup(*F_Params);
      uint32_t HS_DIV_reg = HS_DIV_Lookup(*(F_Params + 1));
      uint32_t RFREQ_upper_reg = RFREQ_Upper_Lookup(RFREQ);
      uint32_t RFREQ_lower_reg = RFREQ_Lower_Lookup(RFREQ);


     pdu.type = SNMP_PDU_RESPONSE;
     pdu.error = status;
    }
    else
    {
     status = pdu.VALUE.encode(SNMP_SYNTAX_OCTETS, freq_str);
     pdu.type = SNMP_PDU_RESPONSE;
     pdu.error = status;
    }
   }
   else {
    // oid does not exist
    // response packet - object not found
    pdu.type = SNMP_PDU_RESPONSE;
    pdu.error = SNMP_ERR_NO_SUCH_NAME;
   }
   Agentuino.responsePdu(&pdu);
  }

  Agentuino.freePdu(&pdu);

}

void setup()
{
  pinMode(1, INPUT);  //set pin 1 to read rx/tx state
  Serial.begin(9600);

  Ethernet.begin(mac, ip);              //Initialize Ethernet Shield
```

```
  Wire.begin(); // Wire communication begin
  api_status = Agentuino.begin();           //Begin Snmp agent on Ethernet shield

  if ( api_status == SNMP_API_STAT_SUCCESS ) {
   Agentuino.onPduReceive(pduReceived);
   delay(10);
   return;
  }
  delay(10);
}

void loop()
{
  Agentuino.listen();
  Serial.println(freq_num);

  TX_RX = digitalRead(1);
}
```

**DHCP.ino**

```
#include <SPI.h>
#include <Ethernet.h>

// Enter a MAC address for your controller below.
// Newer Ethernet shields have a MAC address printed on a sticker on the shield
byte mac[] =

  { 0xA8, 0x61, 0x0A, 0xAE, 0x70, 0x58};

void setup() {

  // You can use Ethernet.init(pin) to configure the CS pin

  //Ethernet.init(10);  // Most Arduino shields

  //Ethernet.init(5);   // MKR ETH shield

  //Ethernet.init(0);   // Teensy 2.0

  //Ethernet.init(20);  // Teensy++ 2.0
```

```
//Ethernet.init(15);  // ESP8266 with Adafruit Featherwing Ethernet

//Ethernet.init(33);  // ESP32 with Adafruit Featherwing Ethernet

// Open serial communications and wait for port to open:

Serial.begin(9600);

while (!Serial) {

  ; // wait for serial port to connect. Needed for native USB port only

}

// start the Ethernet connection:

Serial.println("Initialize Ethernet with DHCP:");

if (Ethernet.begin(mac) == 0) {

  Serial.println("Failed to configure Ethernet using DHCP");

  if (Ethernet.hardwareStatus() == EthernetNoHardware) {

    Serial.println("Ethernet shield was not found.  Sorry, can't run without hardware. :(");

  } else if (Ethernet.linkStatus() == LinkOFF) {

    Serial.println("Ethernet cable is not connected.");

  }

  // no point in carrying on, so do nothing forevermore:

  while (true) {

    delay(1);

  }

}
```

```
  // print your local IP address:

  Serial.print("My IP address: ");

  Serial.println(Ethernet.localIP());
}

void loop() {

  switch (Ethernet.maintain()) {

    case 1:

      //renewed fail

      Serial.println("Error: renewed fail");

      break;

    case 2:

      //renewed success

      Serial.println("Renewed success");

      //print your local IP address:

      Serial.print("My IP address: ");

      Serial.println(Ethernet.localIP());

      break;

    case 3:

      //rebind fail

      Serial.println("Error: rebind fail");

      break;

    case 4:
```

```
    //rebind success

    Serial.println("Rebind success");

    //print your local IP address:

    Serial.print("My IP address: ");

    Serial.println(Ethernet.localIP());

    break;

   default:

    //nothing happened

    break;

  }
}
```

**LO.ino**

```
#include <wire.h>
#include "LO.h"

void Gen_Params(int F_Params[], double Freq){

 int i = 0;
 int j = 0;
 uint32_t N1, HS_DIV;

 for(i = 0; i < 7; i++){

  //get HS_DIV value
  if(i == 0){
   HS_DIV = 4;
  }
  else if(i == 1){
   HS_DIV = 5;
  }
  else if(i == 2){
```

```
    HS_DIV == 6;
  }
  else if(i == 3){
    HS_DIV == 7;
  }
  else if(i == 4){
    HS_DIV == 9;
  }
  else if(i == 5){
    HS_DIV == 11;
  }

  for(j = 0; j < 64; j++){

    //try each N1 value, calculate Fdco, and verify
    if(j == 0){
      N1 = 1;
    }
    else{
      N1 = j * 2;
    }

    double Fdco = Freq * N1 * HS_DIV;

    if((Fdco < 5670) && (Fdco > 4860)){
      F_Params[0] = N1;
      F_Params[1] = HS_DIV;
      return;
    }
    else if(Fdco > 5670){
      break;
    }
  }
 }
}


void hardcode_test(){

  //current config: 137 MHz
  uint32_t reg7_data = 0xA0;
  uint32_t reg8_data = 0xC2;
```

```
uint32_t reg9_data = 0xB3;
uint32_t reg10_data = 0x0A;
uint32_t reg11_data = 0x3D;
uint32_t reg12_data = 0x6C;

//read reg 135
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(135);
Wire.endTransmission();

//request 1 byte from reg 137
Wire.requestFrom(LO_I2C_ADDR, 1);

//block while the wire isn't available
while(!Wire.available()){}

uint32_t reg_135_data = Wire.read();

//read reg 137
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(137);
Wire.endTransmission();

//request 1 byte from reg 137
Wire.requestFrom(LO_I2C_ADDR, 1);

//block while the wire isn't available
while(!Wire.available()){}

uint32_t reg_137_data = Wire.read();

//write reg values:
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(7);
Wire.write(reg7_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(8);
Wire.write(reg8_data);
Wire.endTransmission();
```

```
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(9);
Wire.write(reg9_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(10);
Wire.write(reg10_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(11);
Wire.write(reg11_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(12);
Wire.write(reg12_data);
Wire.endTransmission();

//clear bit 4 in LO:
reg_137_data = reg_137_data & 0xEF;

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(137);
Wire.write(reg_137_data);
Wire.endTransmission();

//set bit 6 of reg 135:
reg_135_data = reg_135_data | 0x20;

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(135);
Wire.write(reg_135_data);
Wire.endTransmission();

//clear bit 6 of reg 135:
/*
reg_135_data = reg_135_data & 0b10111111;

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(135);
```

```
  Wire.write(reg_135_data);
  Wire.endTransmission();
  */

  return;

}

void Reset_LO(){

  //solo LO
  /*
  uint32_t reg7_data = 0xAD;
  uint32_t reg8_data = 0x42;
  uint32_t reg9_data = 0xA8;
  uint32_t reg10_data = 0xB4;
  uint32_t reg11_data = 0x54;
  uint32_t reg12_data = 0xF5;
  */

  //mixer board
  uint32_t reg7_data = 0xAD;
  uint32_t reg8_data = 0x42;
  uint32_t reg9_data = 0xA8;
  uint32_t reg10_data = 0x44;
  uint32_t reg11_data = 0x24;
  uint32_t reg12_data = 0x16;

  //read reg 135
  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(135);
  Wire.endTransmission();

  //request 1 byte from reg 137
  Wire.requestFrom(LO_I2C_ADDR, 1);

  //block while the wire isn't available
  while(!Wire.available()){}

  uint32_t reg_135_data = Wire.read();

  //read reg 137
```

```
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(137);
Wire.endTransmission();

//request 1 byte from reg 137
Wire.requestFrom(LO_I2C_ADDR, 1);

//block while the wire isn't available
while(!Wire.available()){}

uint32_t reg_137_data = Wire.read();

//write reg values:
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(7);
Wire.write(reg7_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(8);
Wire.write(reg8_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(9);
Wire.write(reg9_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(10);
Wire.write(reg10_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(11);
Wire.write(reg11_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(12);
Wire.write(reg12_data);
Wire.endTransmission();
```

```
  //clear bit 4 in LO:
  reg_137_data = reg_137_data & 0xEF;

  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(137);
  Wire.write(reg_137_data);
  Wire.endTransmission();

  //set bit 6 of reg 135:
  reg_135_data = reg_135_data | 0x20;

  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(135);
  Wire.write(reg_135_data);
  Wire.endTransmission();

  //clear bit 6 of reg 135:
  /*
  reg_135_data = reg_135_data & 0b10111111;

  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(135);
  Wire.write(reg_135_data);
  Wire.endTransmission();
  */

  return;

}

 void Read_LO_Config(){

  int i = 0;

  for(i = 7; i < 13; i++){

    //request reg data:
    Wire.beginTransmission(LO_I2C_ADDR);

    //specify register
    Wire.write(i);
```

```
    Wire.endTransmission();

    //ask to receive one byte from specified reg address
    Wire.requestFrom(LO_I2C_ADDR, 1);

    //block while the wire isn't available
    while(!Wire.available()){}

    int val = Wire.read();
    Serial.print ("Reg ");
    Serial.print(i);
    Serial.print(" value: ");
    Serial.print(val, HEX);
    Serial.print("\n");
    delay(1000);
  }
}

 void Write_LO_Values(uint32_t N1_reg_val, uint32_t HS_DIV_reg_val, uint32_t
RFREQ_reg_upper_val, uint32_t RFREQ_reg_lower_val){

   /*
   * Read reg 135
   */
  //specify reg 135
  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(135);
  Wire.endTransmission();

  //request 1 byte from reg 137
  Wire.requestFrom(LO_I2C_ADDR, 1);

  //block while the wire isn't available
  while(!Wire.available()){}

  uint32_t reg_135_data = Wire.read();

  /*
   * Read reg 137
   */
  //specify reg 137
  Wire.beginTransmission(LO_I2C_ADDR);
```

```
Wire.write(137);
Wire.endTransmission();

//request 1 byte from reg 137
Wire.requestFrom(LO_I2C_ADDR, 1);

//block while the wire isn't available
while(!Wire.available()){}

uint32_t reg_137_data = Wire.read();

/*
 * print reg values
 */
Serial.print("Before executing Reg_Write: \n");
Serial.print ("Reg 135 value: ");
Serial.print(reg_135_data, BIN);
Serial.print("\n");

Serial.print ("Reg 137 value: ");
Serial.print(reg_137_data, BIN);
Serial.print("\n\n");

//Try to set bit 4 of reg 137:

reg_137_data = reg_137_data | 0x10;

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(137);
Wire.write(reg_137_data);
Wire.endTransmission();

/*
 * Write new HS_DIV, N1, and RFREQ values
 */

//shift HS_DIV to the left 5
HS_DIV_reg_val = HS_DIV_reg_val << 5;

//break N1 up into upper & lower parts
uint32_t N1_6_2 = N1_reg_val >> 2;
uint32_t N1_1_0 = N1_reg_val << 6;
```

```
//Fix RFREQ values:
RFREQ_reg_lower_val = RFREQ_reg_lower_val | ((RFREQ_reg_upper_val & 0xF) << 28);
RFREQ_reg_upper_val = RFREQ_reg_upper_val >> 4;

//make sure RFREQ_upper is only 6 bits
uint32_t RFREQ_37_32 = RFREQ_reg_upper_val & 0x3F;

//get the other parts of RFREQ:
uint32_t RFREQ_31_24 = (RFREQ_reg_lower_val >> 24) & 0xFF;
uint32_t RFREQ_23_16 = (RFREQ_reg_lower_val >> 16) & 0xFF;
uint32_t RFREQ_15_8 = (RFREQ_reg_lower_val >> 8) & 0xFF;
uint32_t RFREQ_7_0 = RFREQ_reg_lower_val & 0xFF;

//calculate reg 8 & 9 values
uint32_t reg_7_data = HS_DIV_reg_val | N1_6_2;
uint32_t reg_8_data = N1_1_0 | RFREQ_37_32;

//write reg values:
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(7);
Wire.write(reg_7_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(8);
Wire.write(reg_8_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(9);
Wire.write(RFREQ_31_24);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(10);
Wire.write(RFREQ_23_16);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(11);
Wire.write(RFREQ_15_8);
```

```
  Wire.endTransmission();

  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(12);
  Wire.write(RFREQ_7_0);
  Wire.endTransmission();

  //clear bit 4 in LO:
  reg_137_data = reg_137_data & 0xEF;

  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(137);
  Wire.write(reg_137_data);
  Wire.endTransmission();

  //set bit 6 of reg 135:
  reg_135_data = reg_135_data | 0x20;

  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(135);
  Wire.write(reg_135_data);
  Wire.endTransmission();

  //clear bit 6 of reg 135:
  /*
  reg_135_data = reg_135_data & 0b10111111;

  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(135);
  Wire.write(reg_135_data);
  Wire.endTransmission();
  */

  return;

}

uint32_t N1_Lookup(int N1_number){

  //decrement the input value to get the appropriate reg value
  uint32_t N1_reg_val = N1_number - 1;
```

```
  //if the user is trying to write an odd value, round it up
  if(N1_number % 2 == 1)
   N1_reg_val++;

  return N1_reg_val;
}

 uint32_t HS_DIV_Lookup(int HS_DIV_number){

  uint32_t HS_DIV_reg_val;

  //return values based on the table in the datasheet
  if(HS_DIV_number == 4){
   return 0b000;
  }
  else if(HS_DIV_number == 5){
   return 0b001;
  }
  else if(HS_DIV_number == 6){
   return 0b010;
  }
  else if(HS_DIV_number == 7){
   return 0b011;
  }
  else if(HS_DIV_number == 9){
   return 0b101;
  }
  else if(HS_DIV_number == 11){
   return 0b111;
  }

  //return 0 if an invalid divider is requested
  else{
   return 0;
  }
}


 uint32_t RFREQ_Lower_Lookup(double RFREQ_number){

  uint32_t dec_array[9];
  uint32_t sum = 0;
```

```
double intermed;

//get fractional part of RFREQ_number
while(RFREQ_number > 1.0){
 RFREQ_number = RFREQ_number - 1;
}

//TODO: multiply by 2^28 without overflow
//     2^28 = 268,435,456

for(int i = 0; i < 9; i++){

 //shift i decimal places to the left & convert to int
 if(i == 0){
  intermed = RFREQ_number * 10;
 }
 else{
  intermed = intermed * 10;
 }

 dec_array[i] = (long)intermed;

 //remove upper digits
 while(dec_array[i] >= 10){
  if(dec_array[i] > 1000000){
   dec_array[i] -= (long)1000000;
  }
  else if(dec_array[i] > 10000){
   dec_array[i] -= (long)10000;
  }
  else{
   dec_array[i] -= (long)10;
  }
 }

 //Serial.print("\n\tisolated digit: ");
 //Serial.print(dec_array[i]);

 //multiply digit by 2*28
 dec_array[i] = dec_array[i] * 268435456;

 //Serial.print("\n\tdigit times 2^28: ");
```

```
    //Serial.print(dec_array[i]);

    uint32_t divisor = 10;

    for(int j = 0; j < i; j++){
      divisor = divisor * 10;
    }

    //Serial.print("\n\tdivisor: ");
    //Serial.print(divisor);

    //unshift digit
    dec_array[i] = dec_array[i] / divisor;

    //add digit to total
    sum += dec_array[i];
  }

  return sum;
}

uint32_t RFREQ_Upper_Lookup(double RFREQ_number){

  uint32_t digits[2];

  //remove fractional part of RFREQ w/cast:
  uint32_t RFREQ_upper_reg_val = (uint32_t)(RFREQ_number);

  //store upper digit in array
  digits[1] = RFREQ_upper_reg_val / 10;

  //store lower digit in array
  while(RFREQ_upper_reg_val >= 10){
    RFREQ_upper_reg_val -= 10;
  }
  digits[0] = RFREQ_upper_reg_val;

  //multiply each digit by 2^28, then shift right to 0th & 1st index respectively
  for(int i = 0; i < 2; i++){
    digits[i] = digits[i] * 268435456;

    digits[i] = digits[i] >> (28);
```

```
    if(i){
      digits[i] = digits[i] * 10;
    }
  }

  //recombine & return digits
  RFREQ_upper_reg_val = digits[0] + digits[1];

  return RFREQ_upper_reg_val;
}
```

# Appendix V: Abbreviations & Acronyms

- ALU – Arithmetic Logic Unit
- AM – Amplitude Modulation
- ATC – Air Traffic Control
- BPF – Band Pass Filter
- DHCP – Dynamic Host Configuration Protocol
- DRC – Design Rule Check
- IC – Integrated Circuit
- iRMIBB – iReasoning Management Information Base Browser
- LAN – Local Area Network
- LNA – Low Noise Amplifier
- LO – Local Oscillator
- LPF – Low Pass Filter
- MIB – Management Information Base
- OID – Object Identifier
- PCB – Printed Circuit Board
- PTT – Push-to-Talk
- RT – Receiver/Transmitter
- SCL – Serial Clock Line
- SDA – Serial Data Line
- SNMP – Simple Network Management Protocol
- UHF – Ultra High Frequency
- VHF – Very High Frequency

# Appendix VI: Works Cited

[1] "R&S Series 5200 Radios." Rohde & Schwarz, Munich, Germany, May-2020.

[2] "Home," *Leonardo*. [Online]. Available: https://www.leonardocompany.com/en/home. [Accessed: 23-Oct-2020].