# Final Report for Project Lab 3 at Texas Tech University

**Mason Marnell**

Chase Ohlenburger (group member)

Giulia Piombo (group member)

R# 11617302

Texas Tech University

May 10th, 2022

## Abstract

This paper describes the technical information, code mechanics, hardware, and design concepts that were used in Group 8’s lab project. For the project, a software-defined radio dongle was used and, in tandem with the Python programming language, can receive commercial FM radio, FRS radio, and NOAA Weather Radio signals. The program can decode/demodulate the samples taken and export it to a real-time audio stream. A graphical user interface (or GUI) was also created to make the program easier to use and aesthetically simple.

Table of Contents

## List of Figures

## 1. Introduction

Digital Communications Project Lab (colloquially known as CompE Lab 3) is a relatively software-oriented lab in which groups of two to three individuals work on a project that usually pertains to signal processing or wireless communication. For Group 8's project, a custom software-defined radio (SDR) application was chosen.

As will be discussed in this report, Python is used to create a program that can tune to a given center frequency and decode the information broadcasted on that frequency. Due to hardware, antenna, and time limitations, only a select three frequency bands were able to be decoded. The methods used to decode signals will be showcased and described below. Commercial FM radio, FRS radio, and NOAA Weather Radio are all able to be received and decoded, with admittedly surprisingly high-fidelity results.

## 2. Software-Defined Radio

In the recent past, radio had been made possible by dedicated analog hardware, or hardware that is physically tuned to a specific frequency or band of frequencies. With the advent of modern digital information processing however, radio can now be received and processed digitally, with a much wider frequency range and much lower cost than traditional analog hardware. Components such as amplifiers, modulators, demodulators, mixers, filters, and more can be done either in software or with software-accessible embedded hardware. One could theoretically both transmit and receive signals using SDR, however the ability to transmit signals comes at a very high monetary cost and is not needed for this project regardless.

Shown below is a diagram of how SDR hardware functions in general. The SDR takes the signal it receives from the antenna and converts the band of interest to baseband, then receives the samples and sends the data to a digital processor.

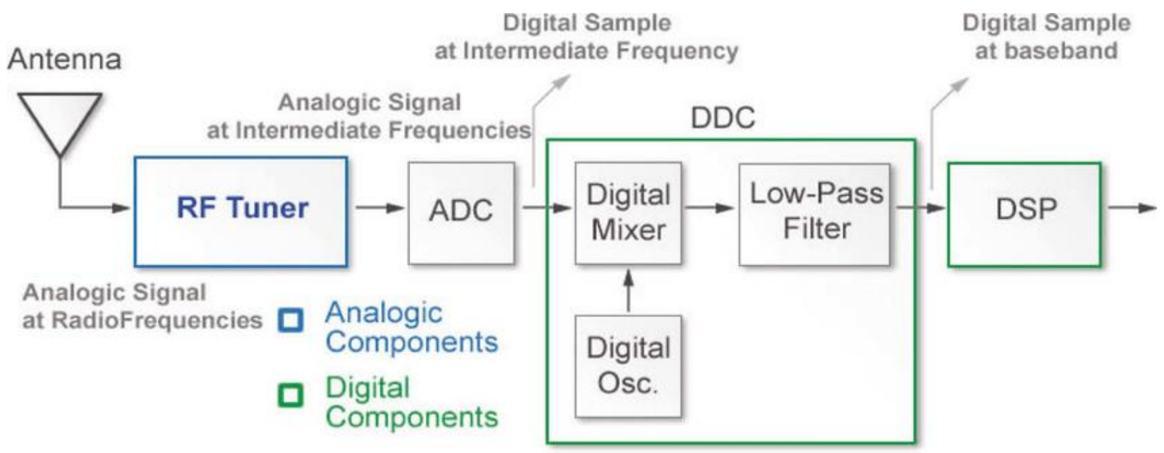

Antenna
Digital Sample at Intermediate Frequency
Digital Sample at baseband
Analogic Signal at Intermediate Frequencies
DDC
RF Tuner
ADC
Digital Mixer
Low-Pass Filter
DSP
Digital Osc.
Analogic Signal at RadioFrequencies
Analogic Components
Digital Components

Figure 1: SDR Diagram [1]

## 3. Hardware

While the software is the main focus of this project, some hardware is required as well. Multiple SDR dongle models have been used for testing, however both of them are based on the RTL2832U chipset found in the now antiquated DVB-T TV tuners. The main dongle used is the RTL-SDR Blog V3, which seems to be the dongle-of-choice for most amateur SDR enthusiasts.

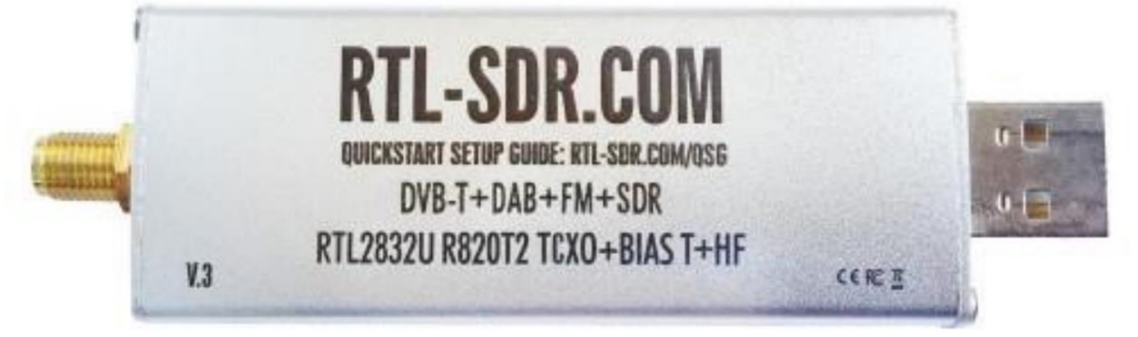

RTL-SDR.COM
QUICKSTART SETUP GUIDE: RTL-SDR.COM/QSG
DVB-T+DAB+FM+SDR
RTL2832U R820T2 TCXO+BIAS T+HF
V.3

Figure 2: RTL-SDR Dongle [2]

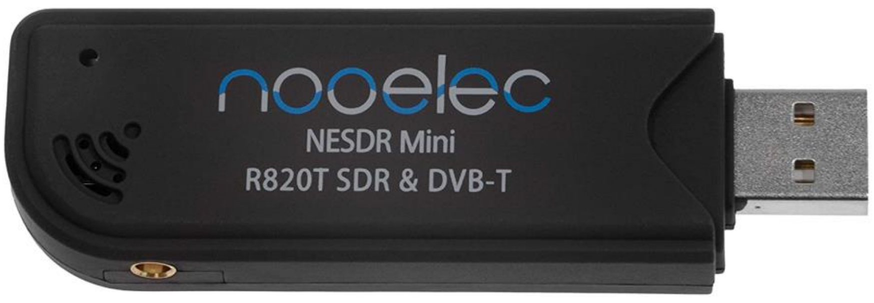


Figure 3: NooElec SDR Dongle [3]

Both dongles came with an antenna and stand, with the RTL-SDR coming bundled with a large telescoping antenna. The RTL-SDR has a bandwidth of up to 2.4MHz, a usable frequency range of 500 kHz to 1766 MHz, a typical current draw of around 280 mA, and a 1PPM TCXO. The TCXO (or Temperature Compensated Crystal Oscillator) is a component in the dongle that slightly alters the frequency characteristics of the dongle based on temperatures. This effectively makes the RTL-SDR immune to frequency slurring due to temperature variance.

All group members additionally had their own Windows based laptop to use for the project. The laptops need not be top of the line, though a relatively decent processing speed, a USB 2.0 port, a Windows operating system, and three logical CPU threads are required for the project in its final state.

## 4. Signal Types and Information

As mentioned preciously, there are three bands of FM signals that are able to be decoded by this application, the first being commercial FM Radio. This band has a frequency range of 87.9 to 107.9 MHz, and bandwidth of 200 kHz (though the actual mono audio signal is only 19 kHz wide). This signal band was the main goal and starting point for the project, as it seemed it would be the best foundation to both learn and implement radio reception and signal processing concepts. Commercial FM implements something known as pre- and de-emphasis, which will be mentioned later. Commercial FM is one of the most popular frequency bands in the VHF range, and even somehow became synonymous with the word "radio" due to its 20th century popularity and widespread use.

The second decodable band that was implemented is FRS, which is most commonly used for handheld radio (or "walkie-talkie") communications. FRS utilizes two bands around 462 and 467 MHz, and shares all of its channels with the higher power GMRS spec. The bandwidth is reported to be 12.5 kHz, though in practice higher bandwidth values gave better results with surprisingly no channel overlap. Squelch, a technique used to mute audio output when no transmission is being received, was investigated but was deemed "a thing that we could have done if we had more time". The group looked into solutions such as using a noise floor filter or utilizing sub-audible tone detection for squelch, but ultimately decided to focus our efforts on other aspects of the programming.

The third and last decodable frequency band is the NOAA Weather Radio. This band ranges from 162.4 to 162.55 MHz, and is dependent on the region of broadcast (162.4 MHz is the primary frequency for Lubbock). The technical bandwidth of this signal is 16 kHz, but in the same was as FRS, higher bandwidth values ended up giving better results. The NOAA Weather Service is a 24-hour broadcast of weather and atmospheric data pertaining to a region. The results for NOAA were not as clear as commercial FM, but the messages nonetheless were easily decipherable from the audio alone.

The project had meant to incorporate more signals, and the one with the most research put into it was APRS (or automatic packet reporting system). APRS has a frequency band of 144.390 to 145.825 MHz, with a bandwidth of 10 kHz. APRS, unlike all three of the previously mentioned decodable analog FM signal bands, is digital. It has a bit rate of 1200 bits per second, and can carry data such as GPS coordinates, weather data, messages, and etc. Although one would think digital signals would be easier to decode and process, the project met a roadblock at attempting to decode APRS signals. Counterintuitively, the project was not able to incorporate APRS into the list of decodable signals, though this signal band was attempted at the rump end of the semester and would have no doubt been implemented with an adequate timeframe.

## 5. Data Stream Model

As seen in Figure 4, a packet-based approach is being used for the project. While this is almost necessary given how the RTL-SDR takes samples in batches (at least by default), the design choice is nonetheless useful for being able to do multiple things with the data received. Though the FFT display and audio output were not able to be done simultaneously due to the matplotlib library not being thread-safe (which means the library is extremely difficult to successfully run on multiple threads), the sample math and packet formatting would have allowed for them to run concurrently given there were no library complications.

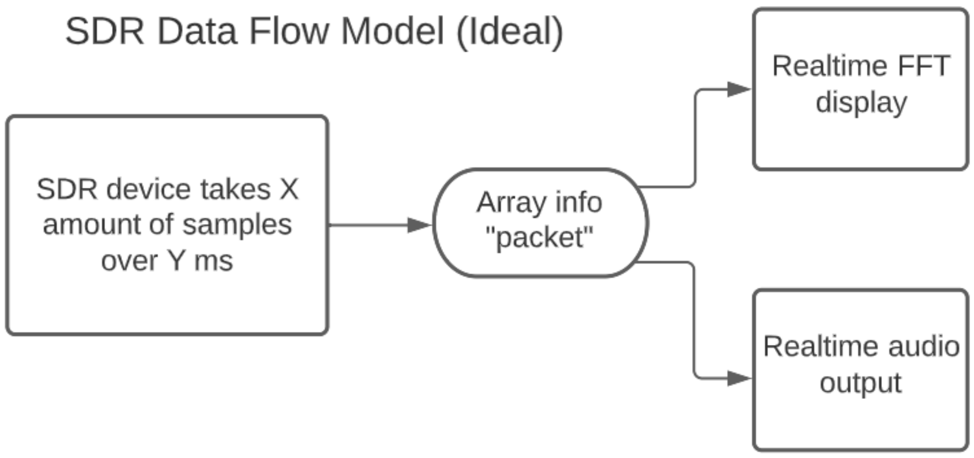

SDR Data Flow Model (Ideal)
SDR device takes X amount of samples over Y ms
Array info "packet"
Realtime FFT display
Realtime audio output

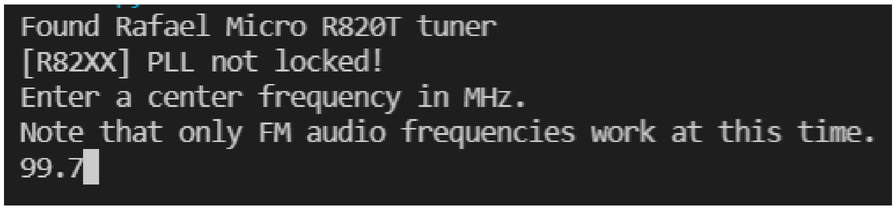
Figure 4: Data Packet Flowchart

## 6. Programming Prerequisites

The setup process for all of the required code libraries and applications was tedious, but it will be explained in short. For the applications needed, all group members installed SDR# (a fully fledged SDR application we used for testing), Python 3, and VSCode. Some of the Python libraries that are currently required are: rtldsr, numpy, scipy, matplotlib, struct, pyaudio, threading, and time, among others. In addition to python libraries, a library that was originally coded in C++ called librtlsdr was used. Precompiled binaries were able to be found, and as such librtlsdr was able to be installed after much tedious experimentation. rtlsdr is a Python library that wraps librtlsdr and allows Python programs to interact with the dongle.

## 7. Main Code Overview

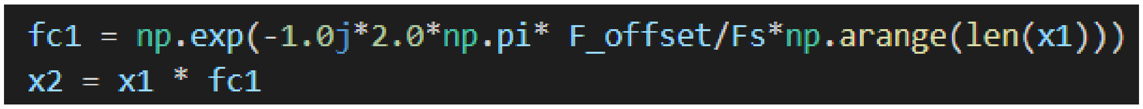
There are three distinct functions that allow the realtime audio to work. takeSamples() communicates with the RTL-SDR and takes in samples. sampleMath() does math on the given samples and converts them to binary data to be read as audio data. playAudio() writes the audio data to the pyaudio bitstream which plays the audio out of the laptop's speakers. Figure 5 shows an antiquated way for the user input a center frequency for testing. This was later replaced by the GUI.



```
Found Rafael Micro R820T tuner
[R82XX] PLL not locked!
Enter a center frequency in MHz.
Note that only FM audio frequencies work at this time.
99.7
```

Figure 5: Console Frequency Input Demo

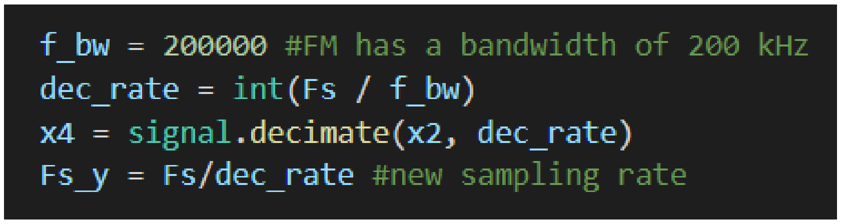
While the first and last functions are relatively straightforward coding-wise, sampleMath() is rather complicated. First the data is mixed down to baseband so that it can be further manipulated.



```
fc1 = np.exp(-1.0j*2.0*np.pi* F_offset/Fs*np.arange(len(x1)))
x2 = x1 * fc1
```

Figure 6: Mix-down Code

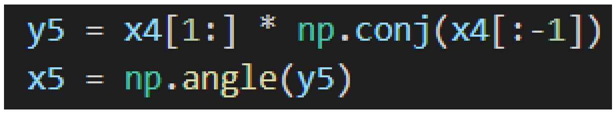
Next, the signal is down sampled using decimation so that the spectrum is narrowed.



```
f_bw = 200000 #FM has a bandwidth of 200 kHz
dec_rate = int(Fs / f_bw)
x4 = signal.decimate(x2, dec_rate)
Fs_y = Fs/dec_rate #new sampling rate
```

Figure 7: Decimation Code

After this, a polar discriminator is used to demodulate the reduced signal.



```
y5 = x4[1:] * np.conj(x4[:-1])
x5 = np.angle(y5)
```

Figure 8: Polar Discriminator

Due to the way FM audio signals are pre-emphasized in the US (to avoid an uneven volume distortion as the signal travels), a de-emphasis filter is applied. Then, the resulting array is decimated again to get just the mono audio band. As seen in Figure 9, FM audio is broadcast on multiple bands for both mono and stereo audio.

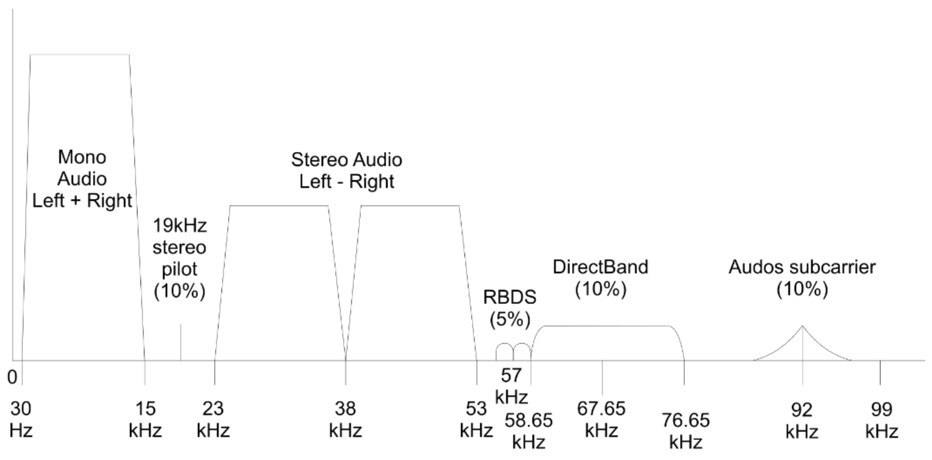
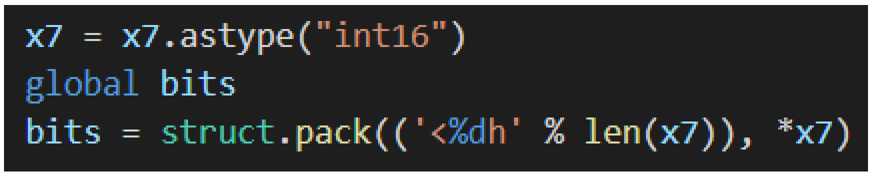


Figure 9: FM Audio Broadcast Spectrum [4]

Finally, the array is converted into signed 16-bit audio samples and saved to the variable that eventually gets sent to the audio stream. A blog post by Fraida Fund was instrumental in guiding us through the signal demodulation process[5].



```
x7 = x7.astype("int16")
global bits
bits = struct.pack(('<%dh' % len(x7)), *x7)
```

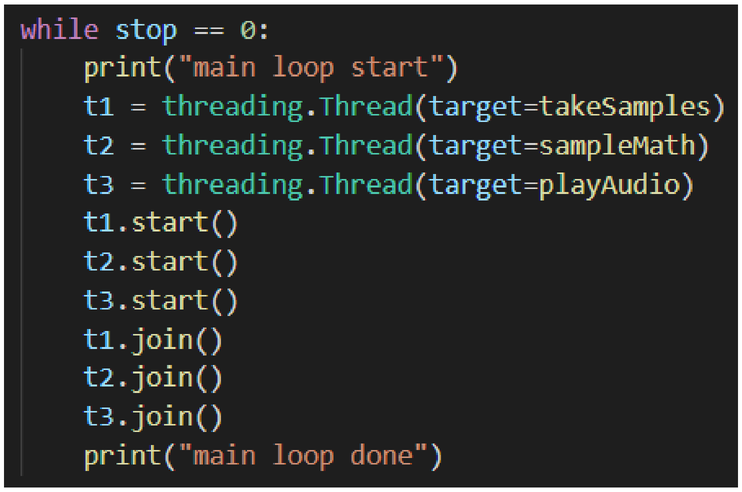
Figure 10: Converting to Audio Data

## 8. Multithreading

After experimenting with different chains of processing, it was clear that the program would need to be able to process multiple things in parallel. It was decided to use multithreading rather than multiprocessing or asyncio. Multiprocessing would have been hard or impossible to use global variables, and asyncio has a larger focus on improving processing times for I/O tasks, not necessarily timed synchronicity.

| | | | |
|---|---|---|---|
| Block 1 | Take x amount of samples | | |
| Block 2 | Do math on those ^ samples | Take x amount of samples | |
| Block 3 | Write to audio stream | Do math on those ^ samples | Take x amount of samples |
| Block 4 | | Write to audio stream | Do math on those ^ samples |
| Block n | | | etc. |

Figure 11: Multithreading Example Diagram



```python
while stop == 0:
    print("main loop start")
    t1 = threading.Thread(target=takeSamples)
    t2 = threading.Thread(target=sampleMath)
    t3 = threading.Thread(target=playAudio)
    t1.start()
    t2.start()
    t3.start()
    t1.join()
    t2.join()
    t3.join()
    print("main loop done")
```

Figure 12: Multithreading Code

While the code above intuitively seems the same as non-multithreaded processing, it actually allows the different blocks of code to run in groups of three simultaneously. Most importantly, each block of code takes the same amount of time; just when the audio stream gets to its last sample, another batch of audio samples is written to the stream and the audio continues.

## 9. GUI

As for the GUI (or graphical user interface), the PySimpleGUI library was chosen. Kivy started out as the GUI library of choice, but was soon found to be unnecessarily complicated to work with. Shown in the figures below are the 4 different GUI pages utilized to interface with the backend of the program. Buttons can be clicked on to either open other pages, or activate the functionality of the program. The GUI code is relatively straightforward and tedious, and as such will not be discussed here but will rather be included in the source code file attached to this paper.

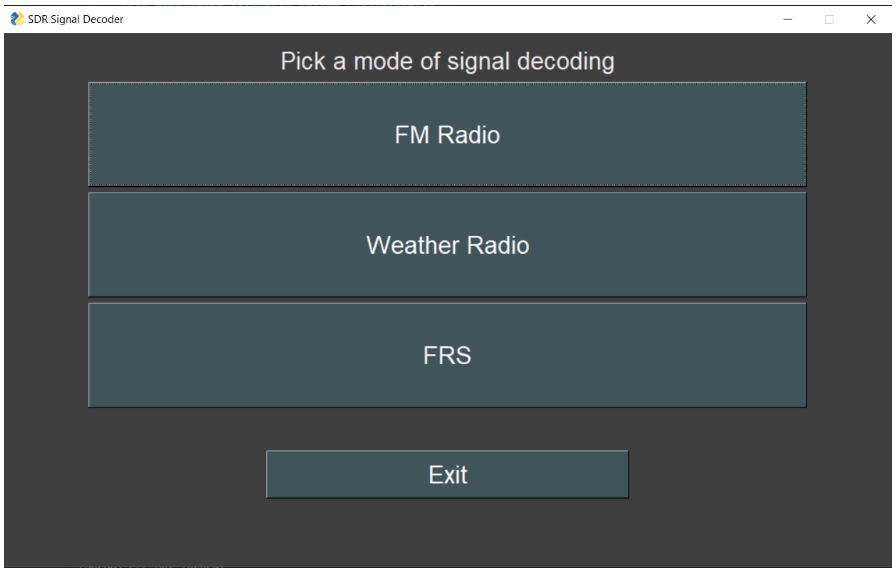
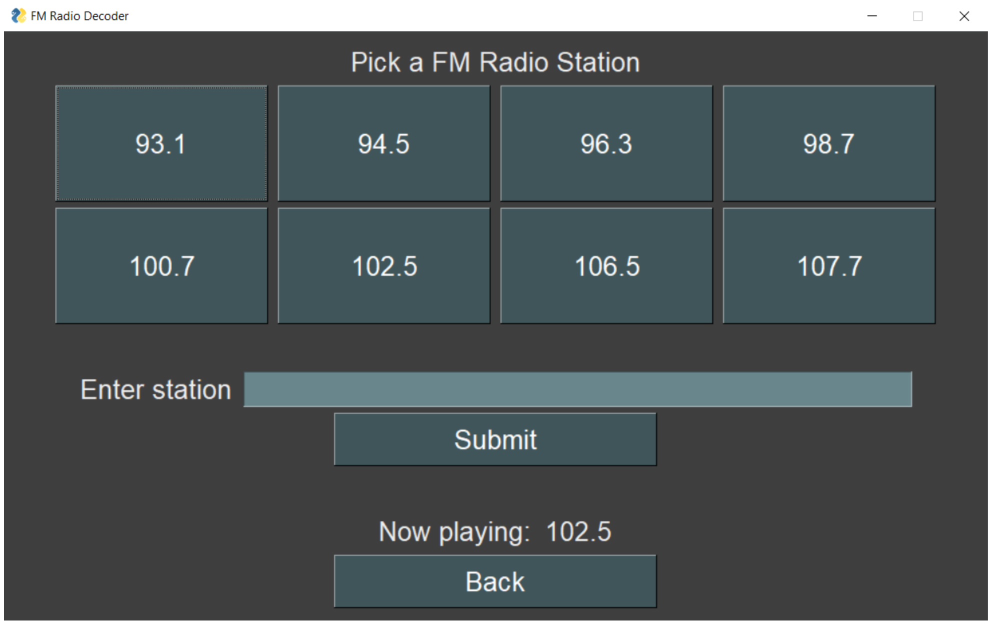
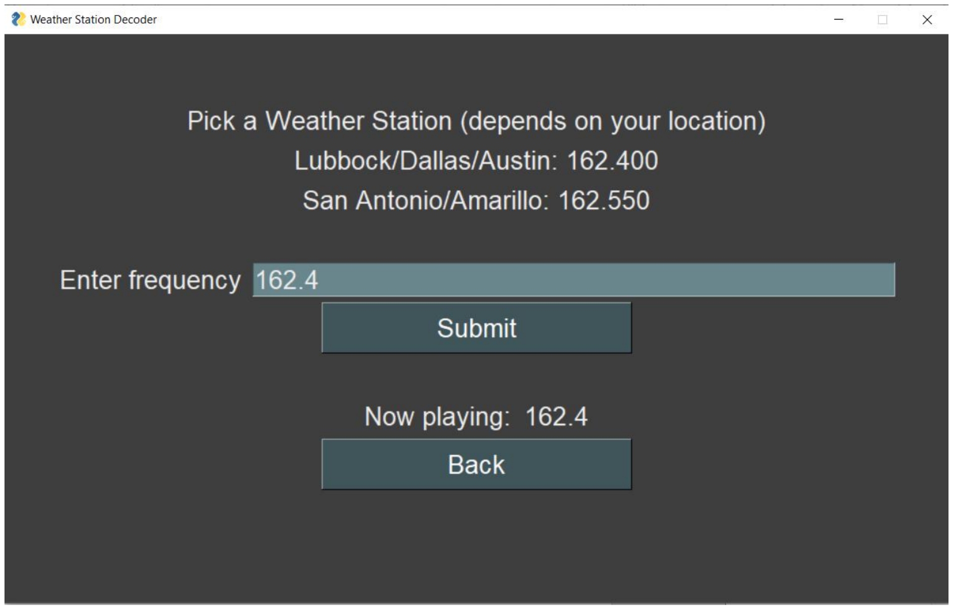


Figure 13: Main GUI Page



Now playing: 102.5

Figure 14: FM Radio GUI Page



Pick a Weather Station (depends on your location)
Lubbock/Dallas/Austin: 162.400
San Antonio/Amarillo: 162.550

Enter frequency 162.4

Now playing: 162.4

Figure 15: NOAA Weather Radio GUI Page

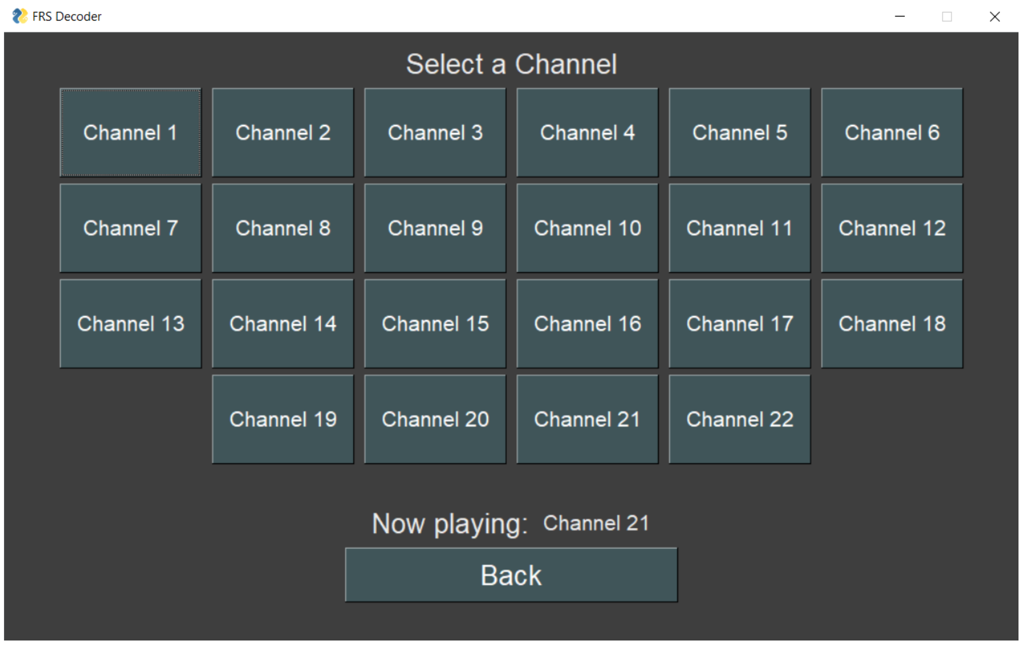


Figure 16 : FRS Radio GUI Page

## 10. Safety and Ethics

It is hard to imagine that any part of this project has a higher safety risk than anything else one would do in their daily life. That being said, one could (for instance) fall from a large height while using a tall antenna, scald themselves on the dongle if it were to overheat, or short circuit a connection if tinkering with the device. None of these (so far) have been a legitimate concern.

Since this project's goal is to receive openly broadcasted signals, there is essentially no perceivable ethical concern unless private military or government frequencies were able to be decoded. Legality in that area is grey.

## 11. Demo Day Functionality

The functionality of the project on demo day was almost ideal. The project completed what it originally set out to do – successfully demodulate and decode multiple bands of frequencies and present the information to the user in a digestible way. All of the coding concepts showcased in this report were implemented, along with some structural code to help mesh with the GUI. With that said, there were two flaws that somewhat affected the functionality of the project.

The first flaw was, as described previously, we were not able to get the FFT display and audio output to run concurrently. We felt that focusing our efforts on cleaning up the project and making sure it was functional took priority over an aesthetic design feature. Secondly, the GUI ran into an issue with freezing during audio playback. This was theorized to be due to the fact that threads were not used for the GUI code, only for the sample processing. Given more time, this issue could have been resolved methodically (and tediously).

Seen below in Figure 17 is an image of a trifold board created to showcase the general concept of SDR, the functionality of the application, and the signals that can and cannot be decoded.

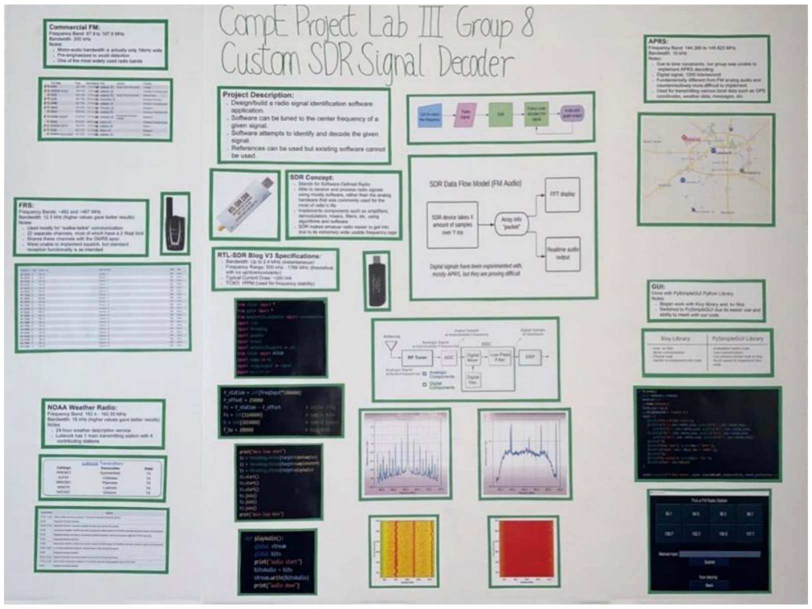


Figure 17: Demo Day Trifold Board [6]

## 12. Conclusion

Regardless of the aforementioned bugs, the project is able to meet all the requirements set forth in the project description. The application can decode three different signal bands and interpret them as the transmitter intended. It is interesting to consider the ways the project could be improved and expanded given another semester, though the program that was written appears to be a good ground-up application that accomplishes a well-defined goal.

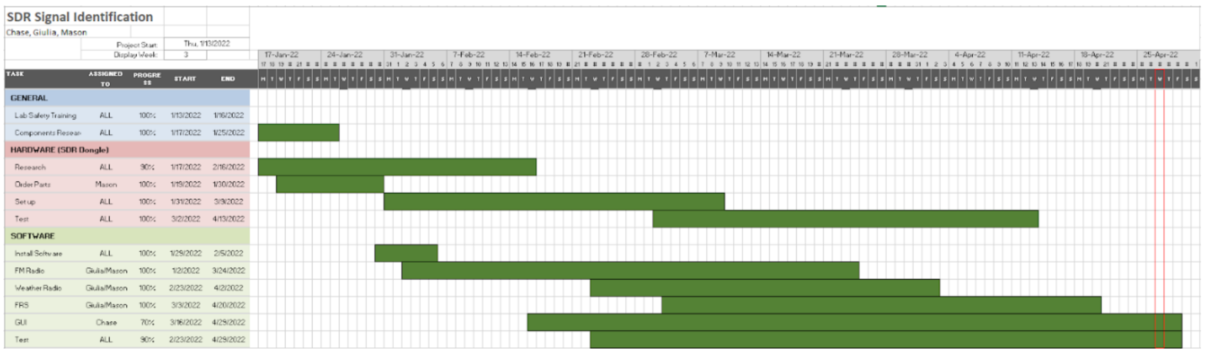
I feel as though I have learned more than I had expected about software-defined radio and radio in general, and I am proud of the results that have fruited from this group's combined and continuous efforts.

## References

1. Fernández, José. “Block Diagram of the SDR Receiver Fig 2,” 2015, https://www.researchgate.net/figure/Block-Diagram-of-the-SDR-Receiver_fig2_303253115 (22 March 2022)
2. RTL-SDR Blog. “RTL-SDR Blog V3 Datasheet,” 2018, https://www.rtl-sdr.com/wp-content/uploads/2018/02/RTL-SDR-Blog-V3-Datasheet.pdf (22 March 2022)
3. NooElec. “Mini Maestro 12-Channel USB Servo Controller,” 2022, https://www.amazon.com/NooElec-NESDR-Mini-Compatible-Packages/dp/B009U7WZCA/ (22 March 2022)
4. Murray, Arthur. “Typical spectrum of composite baseband signal, including DirectBand and a subcarrier on 92 kHz,” 2022, https://en.wikipedia.org/wiki/FM_broadcasting (22 March 2022)
5. Fund, Fraida. “Capture and decode FM radio,” 2016, https://witestlab.poly.edu/blog/capture-and-decode-fm-radio/ (22 March 2022)
6. Ohlenburger, Chase. Discord messaging, 10 May, 2022

## Appendix A

### Gantt Chart



SDR Signal Identification

Chase, Giulia, Mason

Project Start: Thu, 1/13/2022

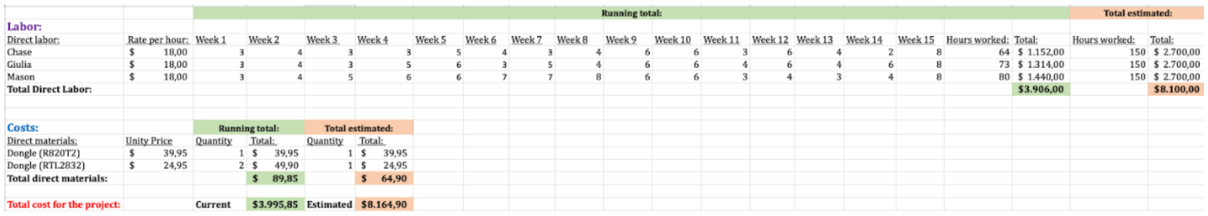
Display Week: 3

| TASK | ASSIGNED TO | PROGRESS | START | END |
|---|---|---|---|---|
| GENERAL | | | | |
| Lab Safety Training | ALL | 100% | 1/13/2022 | 1/16/2022 |
| Components Resear | ALL | 100% | 1/17/2022 | 1/25/2022 |
| HARDWARE (SDR Dongle) | | | | |
| Research | ALL | 90% | 1/17/2022 | 2/16/2022 |
| Order Parts | Mason | 100% | 1/19/2022 | 1/30/2022 |
| Set up | ALL | 100% | 1/31/2022 | 3/9/2022 |
| Test | ALL | 100% | 3/2/2022 | 4/13/2022 |
| SOFTWARE | | | | |
| Install Software | ALL | 100% | 1/29/2022 | 2/5/2022 |
| FM Radio | Giulia/Mason | 100% | 1/2/2022 | 3/24/2022 |
| Weather Radio | Giulia/Mason | 100% | 2/23/2022 | 4/2/2022 |
| FRS | Giulia/Mason | 100% | 3/3/2022 | 4/20/2022 |
| GUI | Chase | 70% | 3/16/2022 | 4/29/2022 |
| Test | ALL | 90% | 2/23/2022 | 4/29/2022 |

A Gantt chart is used to visually represent time spent on different aspects of a project. Before completion, it is used to gauge how much time should be put into various tasks. After completion, it is used to show that distribution of time in a concise and informative way.

## Appendix B

Budget



| | | Running total: | | | | | | | | | | | | | | | | | Total estimated: | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Labor:** | | | | | | | | | | | | | | | | | | | | |
| Direct labor: | Rate per hour: | Week 1 | Week 2 | Week 3 | Week 4 | Week 5 | Week 6 | Week 7 | Week 8 | Week 9 | Week 10 | Week 11 | Week 12 | Week 13 | Week 14 | Week 15 | Hours worked: | Total: | Hours worked: | Total: |
| Chase | $ 18,00 | 3 | 4 | 3 | 3 | 5 | 4 | 3 | 4 | 6 | 6 | 3 | 6 | 4 | 2 | 8 | 64 | $ 1.152,00 | 150 | $ 2.700,00 |
| Giulia | $ 18,00 | 3 | 4 | 3 | 5 | 6 | 3 | 5 | 4 | 6 | 6 | 4 | 6 | 4 | 6 | 8 | 73 | $ 1.314,00 | 150 | $ 2.700,00 |
| Mason | $ 18,00 | 3 | 4 | 5 | 6 | 6 | 7 | 7 | 8 | 6 | 6 | 3 | 4 | 3 | 4 | 8 | 80 | $ 1.440,00 | 150 | $ 2.700,00 |
| **Total Direct Labor:** | | | | | | | | | | | | | | | | | | **$3.906,00** | | **$8.100,00** |

| | | Running total: | | Total estimated: | |
|---|---|---|---|---|---|
| **Costs:** | | | | | |
| Direct materials: | Unity Price | Quantity | Total: | Quantity | Total: |
| Dongle (R820T2) | $ 39,95 | 1 | $ 39,95 | 1 | $ 39,95 |
| Dongle (RTL2832) | $ 24,95 | 2 | $ 49,90 | 1 | $ 24,95 |
| **Total direct materials:** | | | **$ 89,85** | | **$ 64,90** |
| **Total cost for the project:** | | Current | **$3.995,85** | Estimated | **$8.164,90** |

The budget charts are theoretical amounts of money that would be spent to allow us to complete our project had we actually been working for an employer. The total as of the end of the project came out to just over $3,900.

Regarding our budget used from the stock room, almost $90 was used out of our allotted $100. We were able to borrow a handheld FRS walkie-talkie set from Professor McArthur, and as such did not need to have our budget increased for that potential purchase.