COMPENG 3DY4 Project Report

GROUP 35

Anuja Perera | Jack Vu | Ryan Brubacher | Hunter Boyd

perera5@mcmaster.ca | vuj23@mcmaster.ca | brubachr@mcmaster.ca | boydh4@mcmaster.ca

2025-04-04

1 Introduction

This project aims to develop a software-defined radio system capable of handling frequency-modulated signals. The system, programmed in C++, operates in real time on a Raspberry Pi. It processes I/Q samples received from an RTL dongle to generate desired mono, stereo, and RDS outputs. This setup allows the system to deliver audio outputs directly to aplay and RDS data to the standard terminal. The project emphasizes integration of complex, industry-level specifications into a constrained environment, demonstrating practical challenges and digital signal conflicts essential for real-time computing systems.

2 Project Overview

This project pursues the implementation of a real-time software-defined radio (SDR), which is a shift from the traditional hardware-based radios to more flexible software implementations. SDRs enable different demodulation schemes to be implemented entirely in software, without changing physical components. In the context of a SDR designed for the reception of frequency modulated audio and data, SDRs allow for the demodulation of these signals using programmable blocks, given in-phase (I) and quadrature (Q) data acquired by the RF hardware.

Frequency modulation (FM) transmits message signals by varying the rate of change of the carrier wave's frequency in proportion to the amplitude of the message signal. For example, if the amplitude of the message signal increases, the carrier frequency will also increase and alter the shape of the frequency modulated signal. This signal can be recovered via FM demodulation. FM demodulation involves detecting the frequency variations in the modulated signal by computing phase differences in the I/Q data taken from the RF hardware and converting it back into audio or data. FM channels are located in the 88-108 MHz band and increase in 200 kHz steps. These channels include mono and stereo audio, as well as the Radio Data System (RDS), which provides metadata like station identifiers or song information.

The FM receiver in the SDR of this project is represented as a series of basic building blocks that are used to process a continuous stream of data. This data is brought in via FM Hardware as interleaved I/Q samples at a high RF sample rate (i.e. 2.4 MSamples/sec, 960 KSamples/sec, 1.152 MSamples/sec), then downsampled in the FM frontend block to an intermediate (IF) sample rate (i.e. 240 KSamples/sec, 160KSamples/sec, 128 KSamples/sec) to reduce computational load while preserving signal content. The signal is then put through digital signal processing (DSP) blocks including finite impulse response (FIR) filters for channel selection and anti-aliasing, FM demodulation to recover the baseband audio signal, phase-locked loops (PLLs) to stabilize frequencies of subcarriers in stereo and RDS modes, and resamplers to match IF or output sample rates (i.e. 48 KSamples/sec, 40 KSamples/sec, 44.1 KSamples/sec). These blocks are connected to form the three processing paths —mono, stereo, and RDS— which are all found from the same demodulated signal. By modeling and putting together these DSP blocks, the SDR can demodulate real FM broadcasts in real time on the Raspberry Pi. The blocklike modular structure of this project allows for easy performance optimization, verification, and extension.

3 Implementation Details

Lab Background

Building upon the foundational blocks acquired in Labs 2 and 3, we gained a comprehensive understanding of key DSP techniques such as convolution, demodulation, filtering, and impulse response generation. Initially explored using Python for simplicity and understanding, these concepts were then translated into C++ to meet performance requirements for real-time processing. Additionally, Lab 4 was crucial for integrating testing and benchmarking practices into our development process. Understanding g-test allowed us to ensure accuracy and useability of our functions.

Mono

Due to most of the Mono mode 0 code being already implemented in Python during lab 3. Our first step was to refactor this model into C++ to integrate it with the project's real-time pipeline. We faced issues with variable sizing as we had too small of an array size at most points of the signal pipeline in the mono path, such as after downsampling by 10 in the RF front end. This led to segmentation faults when doing convolutions and forced us to have a deeper knowledge of the signal sizes at each point of the signal path and adjust variable sizes accordingly. Additionally, we encountered an implementation bug where we were debugging by writing messages to cout, which interfered with the binary output to my_audio.bin. We resolved this by redirecting debug messages to cerr. These were key lessons that saved us a lot of hassle further down the line.

After refactoring the model code to C++, we split up the Unix pipe by dumping the data into an audio bin and then played that audio bin after using aplay. This was done to evaluate the signal flow and ensure non-real-time functionality. After the audio was validated, we had confidence that our general signal flow for mono modes 0 and 1 was correct. We then began working on real-time implementation of mode 0 and mode 1 by incorporating downsampling into the convolution function. We developed a unit test where we compared the results of convolving array blocks and then downsampling manually to our combined convolution and downsampling function. This led to us recognizing a bug where the output audio exhibited periodic artifacts, traced to an indexing error in the combined convolution and downsampling algorithm. After this test case passed, we had mono modes 0 and 1 working in real time.

Next, we began implementing mode 2 by developing the resampler, which required a fractional resampling ratio of 147/800. We modelled the slow version of the resampler by first modelling the upsampler in Python, inserting 146 zeros between samples. Our approach was to get the slow version of the resampler working with the Unix pipe split into two parts, as we did for more 0. After this was validated, we would move on to unit testing the resampler and then finally running it in real time. The first conceptual issue we faced when implementing the slow version was figuring out the filter coefficient matrix for the low-pass filter. We realized that we needed an expanded matrix by a factor of 147, and we needed to apply a filter gain of 147 due to the increase in sample rate caused by zero-padding in the upsampler. This led to us creating a new filter function that took in the up-sample factor and computed the coefficient matrix using U*Fs for the sample rate and outputted a matrix that was U times bigger with a gain of U applied to every index. After this was fixed, we began working on the fast resampler. We created a unit test to compare the results of the slow resampler to the fast resampler to ensure they matched. Conceptually it was difficult to understand how it would work. To develop the polyphase implementation, we started by studying the inefficiencies of the slow resampler. We realized a polyphase approach could skip the full upsampled convolution by directly computing only the retained output samples, using phase-specific filter coefficients. For each output sample, we calculated its corresponding input index and phase (using (down * i) % up), then applied only the relevant filter taps (stepping by up = 147) to avoid zero multiplications. This led to us finishing the fast resampler and passing the test case.

We then plotted PSD plots using gnuplot in C++ and matplotlib in Python at the audio output in both C++ and Python to ensure they matched. After this was verified, we had all mono modes working on the virtual machine. We streamed live RF data from our dongle into the Raspberry Pi and validated that it worked with live data.

Stereo

Our approach to developing the stereo branch started out with trying to get the stereo path to output audio when processing in a single pass. This was modelled in stereoBasic.py. The main initial conceptual challenges we faced were with understanding the PLL, its inputs and its outputs. We also added the necessary delay function in python, using a shift register like structure. We took the reference code for the band-pass filter in the lecture slides and converted it to python code. We got the single pass version of stereo working with relative ease as it was not too many new difficult functions and used already modelled and understood functions from the mono path.

We then began modeling the block processing version in stereoBlock.py The first step in developing the Stereo path was refactoring the given PLL python code to work with state saving. We realized we needed to create a struct for saving the PLL state that stored the integrator, phaseEst, feedbackI, feedback, lastNcoOut and Trig Offset. This state would be passed into the function and updated to enable the PLL to process blocks of data. We ensured that the PSD plotted from the output of stereoBlock and stereoBasic paths matched. We followed the block processing outline that we used in mono modelling and had our stereo path working with block processing.

We then moved onto refactoring our stereoBlock.py file into C++. We ran into many implementation bugs in C++. One of the main bugs we had was due to having to deal with manually sizing the signal vectors in C++. When combining the mono and audio channels in C++ we were not allocating enough space to store the combined signal, this led to our audio not outputting. We realize that we needed to the audio output vector needed to be 2 * the left and right channel size. After this was fixed, we had audio being output but it was not correct and sounded super distorted. This was due to a small bug where we were not giving the right input to the convolution on the mono path. We needed to give the delayed demodulated data to the convolution function, but we were giving just the normal demodulated data from the frontend. After this was fixed, we were getting audio in real time, but it was not split up nicely on the left and right side of the headphones. This was because we were accidently declared a delayed buffer of size 101 when we were using a shift register like structure for our all-pass filter. The delay size was changed to 50 to match the delay on the stereo path and then we had Stereo mode 0 and 1 working fully in real time.

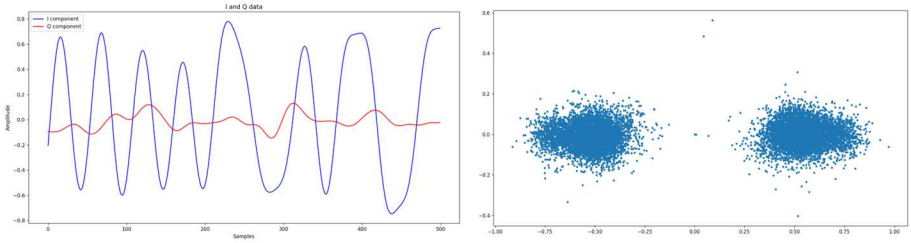
Next, we incorporated the resampler from the mono path to do stereo mode 2 and 3. Since this resampler was well tested and understood, incorporating it into the stereo path was relatively simple. After this was implemented, we had all stereo modes working in real time. We compared the PSD plots at the output of each mode to the stereoBlock.py to ensure we implemented the signal path correctly. After this was verified, we had all stereo modes working on the virtual machine. We streamed live RF data from our dongle into the raspberry pi and validated that it worked with live data.

RDS

The first new function developed for RDS was the sampling function used to recover the clock and data from the root raised cosine (RRC) filter output. Initially, we designed the function to process a block of size 'SPS' (Samples Per Symbol) and select the middle value from that block (index SPS//2) as the sample. While this approach worked temporarily, it did not account for phase drift in the signal. To address this, we updated the function to calculate the maximum index from the first block and offset that index by ‘SPS’ to get each subsequent sample. The maximum index was recalculated every 100 'SPS' blocks. This method worked initially, but it led to the first bug we encountered in the RDS system. When there were two consecutive high or low samples, recalculating the max index caused the samples to misalign with the

following ones. To debug this issue, we graphed the output waveform from the RRC filter and plotted the sampling points. This revealed the issue and led us to develop a final version of the sampling function. In the final design, the offset was adjusted by ±1 towards the max value every 100 'SPS' blocks, ensuring that outlier samples wouldn't significantly affect the offset.

We then focused on tuning the phase-locked loop (PLL) to the carrier frequency. This process included generating constellation graphs of the samples to assess their symmetry and the tightness of the clusters. We also plotted the I-wave outputs from the RRC filter and compared them to the Q-waves to ensure that the maximum energy was being directed to the I-waves. The values we found through trial and error was a phase adjust of 0 degrees and a normBandwidth of 0.0045. Once we achieved an adequate constellation graph and sample quality, we shifted focus to updating all functions for block processing. Although we couldn't fully resolve the issues by the end, we had some successes with the functions. The Manchester and differential decoding functions were both updated to handle previous values, with the Manchester decoding also improving to properly handle cases with or without extra unprocessed samples. Additionally, the sample retrieval function was under troubleshooting due to bugs related to how it accessed previously passed-in samples and offset values.



I and Q data outputted and normalized from RCC filter(left). Constellation graph of samples(right).

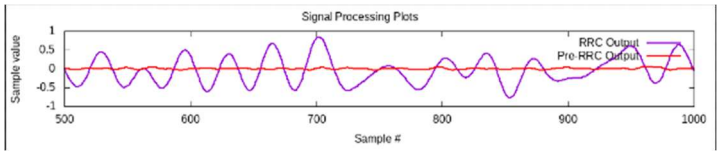
The second and third functions we designed after validating the sampling function were the Manchester and differential decoding functions. While the differential decoding function was straightforward, the Manchester decoding function caused some issues. Initially, we assumed that index 0 of the array would always represent the first sample to calculate the first bit. However, this assumption was incorrect, as index 0 could sometimes be the second sample instead. To solve this, we modified the function to loop through the input samples twice, each time assuming either index 0 or index 1 was the first sample. It then compared the number of errors from each assumption and kept the format that resulted in fewer errors for the remaining function calls.

The final functions were designed to calculate the syndromes, achieve synchronization, and then extract the radio data from each block. To accomplish this, we created a function that, while unsynchronized, checks each set of 4 consecutive 26-bit blocks to see if they form the syndromes in the pattern A-B-(C or C')-D. If the pattern is not matched, the function shifts to the next bit and checks again for 4 syndromes. The primary issue we encountered was an incorrect bit in our parity matrix. To identify this, we passed in individual checkwords with the message bits zeroed out and checked which one was incorrect. After fixing the issue, the function synchronized quickly for the raw file we were using for testing. Once synchronized, the function extracts the character data from each frame. For the program information, it appends each character to the end of a string without refreshing it.

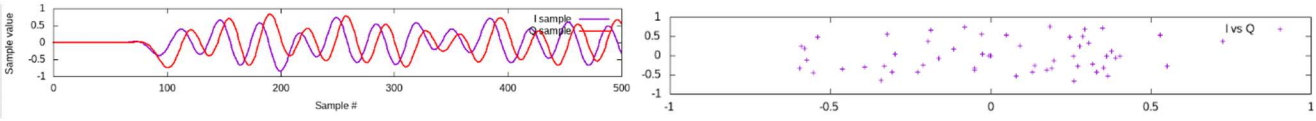
When implementing the block processing approach for RDS, we ran into multiple challenges with the system. One significant issue was the handling of vectors, as they were cleared at the beginning of every run. What happened was Q data was being plotted but I data was not. After hours of debugging the problem was having a vector.clear(), deleting all our data. In the end our issues likely strayed from errors in previous sample handling. At the beginning of our plots, we'd always see discrepancies in the data, but it'd slowly adjust to the expected values. Unfortunately, at this point we ran out of time and couldn't investigate further.

Following the modelling of RDS, it became much easier to implement the C++ version since the groundwork for implementation was already made. Since the multithreading was already implemented for the frontend and audio, we decided to directly make a new thread and test RDS from there. Like the audio thread, the RDS thread contained all the RDS processes such as convolution, RRC filtering, and normalizing. The backend contained all the constants. Beginning this portion of the project we ran into data retrieval errors, discussed in the Threading section.

The first function translated from python to C++ is the root-raised cosine filter, where we directly translated the python function to C++. We then created a normalize function that took the signal inputs and found the maximum to have an amplitude of one. This was not directly tested with the python version, but we were able to test the carrier data outputs pre-RRC vs RRC, and it displayed expected data, similar to the python version.



To compare the I and Q samples and create a constellation plot, we had to implement a new PLL function that would have the two numerically controlled oscillator outputs. This involved adding a second output that was multiplied by a sin wave. We then tested the functionality by printing the ncoI and ncoQ outputs, where the results were proven by $\sin^2(\text{ncoQ}) + \cos^2(\text{ncoI}) = 1$. However, during this phase we encountered an issue where the Q samples showed excessive magnitude, appearing as if it was a phase shift of the I samples. Despite extensive debugging efforts, including adjusting the PLL parameters, thread inputs, and procedure methodology, the results remained the same. We attempted to print the constellation but since our I and Q samples were inaccurate; the constellation followed the same pattern.



Although the I and Q retrieval was incorrect, we proceeded to implement Manchester decoding and differential decoding, attempting to emulate the single pass python model. We were able to get 1's and 0's printed in the terminal but without proper sampling, we could not verify our work. Next, the parity check was attempted but we lacked the deep understanding of this functionality and combined with the incorrect I and Q values, this was as far as we got. At this point it was the final hours of the project, and we chose to stop here and do a final verification of our other modes.

Threading

Our threading implementation strategy for the SDR system involved dividing the workload into multiple executions path to utilize the multi-core capabilities of the Raspberry Pi. Key components were the RF front-end, audio processing thread, and RDS processing thread. Conceptually a large challenge was figuring out what components of the signal pipeline to place in each thread. The setup required clear division between front-end processes, which handled I/Q data capturing and digital filtering, while the back end processed specific operation modes like stereo, mono and RDS.

Tests focused on the audio thread functionality, where we'd pop data from the queue. We began with mode 0, checking if fm_demod data was being received and then transferring our processes into the thread. Challenges arose with managing audio distortion due to the additional computations which occured from logging data and running aplay. To combat this, any form of vector logging or gnuplot graphing would be done with aplay sending audio data to a bin file. User inputs were then added into the main function for us to easily change between modes such as Mono mode 1 stereo mode 3 etc. This allowed us to fully test the

functionality of every mode when running it in the threaded configuration. Since mono and stereo were fully implemented, putting them into the threads was straightforward once the conceptual challenges were overcome.

The introduction of the RDS thread presented most of our challenges, particularly managing two consumers threads with a single queue. We encountered difficulties with data distribution between these threads. Initially, we tried either pushing data twice into one queue or utilizing two separate queues. However, these methods often resulted in one thread receiving data while the other ended up with nothing. Examining our push function, we experimented with both copying and moving the data into the queue, but the first push consistently received data while the second queue was empty. We stuck with the dual queue idea, this time implementing shared pointers, where the threads would pop a pointer and fm_demod would be dereferenced in the respective threads. After this shift we were able to receive fm_demod data into both threads, and we were able to proceed with the RDS functionality.

Up until the integration of the RRC integration of the RDS thread, both the audio and RDS threads had been performing without any timing conflicts. However, we did have a glimpse of potential issues when we encountered a segmentation fault that prematurely terminated the RDS thread. This was due to inadequate size declarations for the mixer, disrupting the RDS thread, and halting the audio output. The early termination showed us that with our current system, if one thread ended earlier, it would impact the entire functionality of the system.

4 Analysis and Measurements:

Each path and mode incur its own runtime per audio sample or data bit, depending on the computational complexity required to produce the respective output. All modes share a common set of computations from the RF front end, which includes FIR filtering and decimation of the I/Q samples down to the intermediate frequency (IF) sample rate resulting in:

$$Front\ End\ MACs = \frac{(2 * If_{Fs} * NUM_{TAPS})}{Output_{Fs}}$$

The mono path then requires an FIR lowpass filter and decimation to the output sample rate, resulting in:

$$Mono\ MACs = \ Front\ End\ MACs + \frac{(Output_{Fs} * NUM_{TAPS})}{Output_{Fs}}$$

The stereo path contains all the computations from the mono path, plus two bandpass filters and a final low pass filter, resulting in:

$$Stereo\ MACs = \ Mono\ MACs + \frac{(2 * If_{Fs} * NUM_{TAPS} + Output_{Fs} * NUM_{TAPS})}{Output_{Fs}}$$

The RDS path contains the RF front end, as well as 2 band pass filters, a low pass filter, a rational resampler and a root raised cosine filter.

$$RDS\ MACs = Front\ End\ MACs + \frac{(3 * If_{Fs} * NUM_{TAPS} + 2375 * SPS * NUM_{TAPS} + 2375 * SPS * NUM_{TAPS})}{Output_{Fs}}$$

Mono and Stereo MACs					
Mode	Input Samples	Intermediate Samples	Output Samples	Mono MAC's per sample	Stereo MAC's per sample pair
0	2,400,000	240,000	48,000	1111	2222
1	960,000	160,000	40,000	909	1818
2	2,400,000	240,000	44,100	1200	2400
3	1,152,000	128,000	44,100	687	1374

RDS MACs				
Mode	SPS	Intermediate Samples	Output Samples	RDS MAC's per bit
0	26	240,000	1187.5	112567
2	39	240,000	1187.5	117819

For the mono path, the only non-linear operations occur in the RF front end, where the fm_demod function applies an atan operation to each input I/Q sample. As a result, the average number of non-linear operations is approximately the number of intermediate samples divided by the number of output samples. In the stereo path, additional non-linear operations are introduced in the phase-locked loop (PLL), which uses both atan and cos functions. The RDS path shares this same structure, with non-linear operations from the RF front end and the PLL contributing to the total count, however the output samples is much lower, resulting in more operations per bit.

Non-Linear Operations			
Mode	Mono Non-Linear Operations per sample	Stereo Non-Linear Operations per sample pair	RDS Non-Linear Operations per bit
0	5	15	606
1	4	12	NA
2	5.4	16.3	606
3	2.9	8.7	NA

Runtime Measurements Using sample0.raw for 5 seconds of I/Q samples (ms)										
DSP Block	**Frontend I Filter**	**Frontend Q Filter**	**Demod**	**Mono Output Filter**	**Mono All Pass**	**Stereo Carrier Filter**	**Stereo Channel Filter**	**Stereo PLL**	**Stereo Mixer**	**Stereo Output Filter**
Mode 0 101 taps	368.88	365.4	36.2975	75.35	6.95465	361.00	362.68	274.67	6.11	72.47
Mode 1 101 taps	379.37	376.02	30.3692	64.84	6.96209	382.63	365.32	280.40	4.72	62.72
Mode 2 101 taps	385.22	394.28	39.6347	264.19	7.68579	365.28	367.80	276.29	5.46	266.79
Mode 3 101 taps	305.07	302.08	19.461	269.96	5.03014	292.25	290.45	224.57	3.54	247.90
Mode 0 13 taps	70.78	63.88	39.74	67.03	9.97403	73.40	67.39	328.58	9.66	13.18
Mode 0 301 taps	1026.63	1005.61	41.4808	207.434	6.44321	1020.04	1014.25	275.045	6.29303	206.13

When comparing the MAC analysis with the measured runtimes, the results generally align, with minor deviations attributed to differences in function implementations. According to the MAC analysis, mode 2 requires the most MAC operations, followed by mode 0, mode 1, and mode 3, which has the least. However, when summing the runtimes for each mono path, mode 2 has the longest runtime at 1091ms, followed by mode 3 at 901ms, mode 1 at 858ms, and mode 0 at 852ms. This order contradicts the MAC analysis, as modes 3 and 1 should have faster runtimes than mode 0. The discrepancy is resolved by excluding the resampler's runtime, which changes the order to mode 2, mode 0, mode 1, and mode 3, aligning the results with the theoretical MAC analysis. This indicates that the resampler contributes disproportionately to the runtime, and once removed, the results match the expected calculations.

For mode 0, the runtime behavior with 13 and 301 taps aligns well with expectations: the 13 taps lead to a significant runtime reduction at 251ms, while the 301 taps cause a substantial increase to 2288ms. This is consistent with the fact that the number of taps directly impacts the number of calculations in each filter block.

5 Proposal for Improvement

Integrating a user interface for managing RDS data and audio playback would significantly enhance user interaction by providing control over audio settings and real-time information. This could be implemented using a framework like React for seamless integration with our C++ backend. Instead of having the mode be called through the terminal, our main function would receive the input from the front end and begin our threads with the desired mode. Additionally, implementing a Git pipelining system that automatically tests our software updates using pre-existing audio bin files and RDS program information. Progressing through the project, someone would develop scripts on our Git repository using the expected outputs and compare results. This would improve productivity throughout the group as it automates the testing process, ensuring that each change adheres to specific performance benchmarks before integration.

To optimize our system for simpler hardware, several improvements can be made, as many functions are currently unoptimized. The first optimization involves replacing direct function calls for sin, cos, and atan with lookup tables (LUTs), which are interpolated to approximate these values. This change reduces the time complexity of these non-linear functions to O(1), greatly speeding up the relevant calculations. However, this comes with the trade-off of reduced precision and increased memory usage. In resource-constrained systems, where processing speed is often more critical than memory, this trade-off is generally acceptable. Another optimization focuses on enhancing vector management in C++. Currently, vectors are dynamically resized using resize() and clear(), leading to frequent memory reallocations. By using reserve() to pre-allocate memory, we can avoid unnecessary reallocations, which improves performance by reducing the overhead of memory management. Finally, replacing floating-point arithmetic with fixed-point arithmetic can significantly boost calculation speed, although it will result in a slight reduction in precision for the final results.

6 Project Activity

Week	Progress	Contributions
Feb 10	Project Released	Everyone read through the project document
Feb 17	Reading Week	Everyone had reading week
Feb 24	Began modelling mono mode 0 and created convolution with downsampling	Anuja: Wrote the convolution with down sampling function and unit tests for it Ryan: Wrote the framework for the mono path in C++, wrote the simple convolution functions in C++ Hunter: Refactored efficient fmDemod function to C++, unit tested convolution with down sampling function Jack: Wrote the mono single pass and block process modelling with up and down sample, resampler not implemented at this point
Mar 3	Started setting up threading and implementi	Anuja: Debugged and implemented mono mode 0 and 1. Starting working on the slow resampler. Confirmed unit tests for the downsampler with convolution. Ryan: Optimized the simple convolution and convolution with downsampling to use some of the algorithms from lab 4

	ng mono mode 0 in C++	Hunter: Unit tested efficient convolution with downsampling function, attempted debugging for mono mode 0 implementation. Jack: Found error in mono mode 0 implementation, block size declaration issue causing static output, had proper audio now.
Mar 10	Finished implementing real time mono modes 0-3	Anuja: Debugged and finished the signal flow for all mono modes in C++. implemented the slow resampler, then the fast resampler. Developed unit tests for the fast resampler. Ryan: Debugged the convolution with downsampling in C++ as it was used in the mono modes 0-3. Fixed problem with incorrect previous sample indexing. Hunter: Implemented PLL with state saving in C++, bandpass impulse response, familiarized myself with stereo path. Jack: Read documents and lecture notes on stereo path and multithreading. Began up sampling and downsampling process for mono modes 2 and 3
Mar 17	Started and finished modelling, implementation and real time stereo modes 0-3	Anuja: Implemented the initial threading which included the mono in one thread and the RF-Front end in another thread. Debugged stereo modes 0 and 1, including bugs with the bandpass filter. Modelled stereo mode 0 and 2 with block processing and the resampler. Ryan: Wrote the modelling python files for both single pass and block processing stereo, wrote the framework for stereo mode 0 implementation. Hunter: Debugged issue across all stereo modes where the left and right channels were appearing in both channels. Fixed it by changing the order of the delay block for the mono path which originally came after the mono output filter instead of before. Jack: Implemented stereo modes 1, 2, 3, finding up and down values and achieved valid audio outputs. Mixed up all pass and downsampling, which resulted in audio but no channel splitting. Began implementation for threading containing both audio and RDS threads.
Mar 24	Started and finished modeling RDS, started implementing RDS	Anuja: Updated project.cpp so that we could specify which modes (M, S, RDS) and number (0,1,2,3) to run the code in for faster debugging. Debugged RDS modeling for getting the I and Q path and constellation diagrams. Implemented the signal flow up to CDR in a separate thread for RRC. Ryan: Wrote the modelling for single pass RDS, wrote and designed the python code for all new functions used (sampler, Manchester and differential decoding, parity check matrix), worked to convert single pass RDS to block processing and helped debug it (focusing on the sampler). Wrote the C++ version of the parity check function. Hunter: Wrote RDS block processing model, tried converting RDS sampling blocks to state saving and debugged it until the deadline. Began RDS implementation up to sampler. Jack: Began RDS implementation with Hunter, following the RDS workflow and programming in the lab together. Tested Hunter's code and fixed common bug issues, until project file compiled. Implemented new RDS functions such as normalize, getsamples, RRC, and tested, where proper RRC data was plotted. Attempted decoding implementations but ran out of time.
Mar 31	Completed report and timing analysis	Anuja: Stereo implementation section, Mono implementation section, proposal for improvement. Ryan: RDS single pass implementation section, all MAC calculations, second paragraph of proposal for improvement.

		Hunter: Project overview, benchmarked all DSP blocks, measurement section of Analysis and Measurements Jack: Introduction, Lab discussion, RDS block processing and C++ discussion, Multithreading discussion, proposal for improvement, and formatting

Challenges

Anuja: One of the main challenges I faced was developing and testing the resampler for mono mode 2 and 3. This challenge consisted of understanding the filter coefficient scaling, handling fractional resampling ratios, and designing a testable unit in C++ demanded a deep understanding of DSP concepts. Another major hurdle was ensuring the stereo path functioned correctly in real time. This involved debugging in C++ such as allocating correct buffer sizes and resolving convolution bugs that led to distorted audio output. These challenges were crucial to overcome, as the proper functioning of the resampler and stereo modes was essential for further modes and real-time system performance.

Ryan: The primary challenge I encountered was designing and testing the single-pass RDS path. Developing the sampling function resulted in numerous bugs, and resolving these issues was critical for the proper functioning of the RDS system. The recovered bits are highly sensitive to the sampling points, so ensuring accuracy in this stage was essential. The parity matrix had its own set of challenges, particularly in verifying that the correct matrix operations were being applied. This was crucial because even a single incorrect bit could prevent the system from synchronizing with the signal.

Hunter: The most challenging problem I worked on was debugging the RDS block processing constellation. After the sampling stage, the constellation failed to form correctly, unlike in the single-pass version. The root cause was improper state saving. While filtering and demodulation properly handled state transitions correctly, the sampling stage relies on leftover samples and alignment from the previous block, which we couldn't consistently maintain. Without accurate state saving, the sampled symbols became misaligned, leading to an incorrect constellation. Debugging this is non-trivial but necessary, as it directly impacts our ability to decode RDS symbols correctly in a real-time system.

Jack: The main challenge was implementing and testing multithreading with both audio and RDS paths active. A single-producer, single-consumer queue was straight forward, but with two consumers, one thread would receive data while the other wouldn’t. To resolve this, I implemented separate queues for each thread, eliminating any competition. I also switched to using pointers to avoid copying full data blocks. This revealed several minor yet critical issues, such as function order for modes and vector size mismatches. Fixing this was crucial as threading is the major communication framework in the project.

7 Conclusions

Throughout this project, our team deepened our understanding of real-time system design and optimization through the development of a software-defined radio. We addressed challenges in high-throughput data processing, strict timing constraints, and complex transformations like modulation, demodulation, and filtering, requiring low latency and reliable performance. The project bridged theory and practice, letting us apply concepts from coursework, such as Fourier analysis and digital filtering, in a hands-on setting. We also gained experience in collaborative project management, including task division, coordination across development stages, and using version control software like Git to collaborate on a large software project. Overall, the project enhanced both our technical capabilities and our preparedness for future real-time, high-performance system challenges.

8 References

[1] N. Nicolici, “Project Document,” Computer Systems Integration Course at McMaster University, https://avenue.cllmcmaster.ca/d2l/le/content/633404/viewContent/5043264/View (accessed Apr 4, 2025).