COE3DY4 Project Report

Group 61

Biswajit Saha – Jibin Mathew - Khawja Labib - Sai Satwika Kurra
sahab2@mcmaster.ca – mathej30@mcmaster.ca - labibk@mcmaster.ca - kurras@mcmaster.ca

April 5, 2024

Introduction

This project aims to create a Software Defined Radio (SDR) system on a Raspberry Pi 4, utilizing a Realtek RTL2832U RF dongle to receive real-time Frequency Modulated (FM) mono/stereo audio and digital data via the Radio Data System (RDS) protocol. The focus is on developing and implementing software to process these signals, with user interaction facilitated through the command line. The SDR system is designed to adapt to various operational modes by altering sampling frequencies and associated parameters, ensuring dynamic processing capabilities. Developed in C++ with Python for logic verification and testing, this setup represents a comprehensive approach to applying and advancing SDR technology.

Project Overview

This project explores FM broadcasting, where carrier waves modulate in frequency to encode information. It focuses on the demodulation of FM signals within Canada's 88.1 MHz to 107.9 MHz range, specifically targeting positive frequencies (0 to 100 KHz). Our attention centers on three key sub-channels: mono (0-15 kHz) for primary audio, stereo (23-53 kHz) for enriched sound, and RDS (54-60 kHz) for digital information, each pivotal to the FM broadcast experience.

In the RF Front-End Block, a low pass filter initially isolates desired FM channel frequencies by attenuating higher frequency components, followed by a decimation process with a factor of 10 to achieve an intermediate frequency (IF) for advanced processing. The system accommodates multiple operational modes, adjusting the input data rate to either 2400 Ksamples/sec for mode 0 and 2, 1152 Ksamples/sec for mode 1, or 1920 Ksamples/sec for mode 3, with mode 0 set as the default configuration.

For the mono path, IF values derived from the RF front-end undergo further filtration through a low-pass filter. Subsequently, these values are expanded and decimated according to the selected mode, culminating in an Audio FS output of 48 Ksamples/sec for mode 0, 36 Ksamples/sec for mode 1, 44.1 Ksamples/sec for mode 2 and 3. This output, formatted as 16-bit signed integers, is then relayed to an audio player for playback.

Stereo path utilizes IF values from the RF front-end, diverging into two distinct processes for stereo recovery extraction and stereo channel extraction. Both paths employ band-pass filters to meticulously extract relevant frequencies. A phase-locked loop (PLL) with an integrated numerically controlled oscillator is pivotal in this stage, purifying the sinusoidal signal and minimizing noise interference. The refined signals from these paths are subsequently amalgamated with the mono audio FS output, rendering left and right audio channels at a sampling depended on the mode, thereby achieving high-quality stereo sound.

The Radio Data System (RDS) path was incorporated to harness digital information broadcasted alongside the conventional FM signal. Initially, the RDS Channel Extraction employs a band-pass filter tuned to the 54-60 kHz frequency range, isolating the RDS subchannel. Subsequent RDS Carrier Recovery includes squaring nonlinearity to generate a 114 kHz tone, subsequently filtered and locked to a clean 57 kHz carrier with a PLL and NCO setup. In the demodulation phase, the signal undergoes low-pass filtering and rational resampling, tailored to the IF sample rate, adjusting the symbol rate to 41 for mode 0 and 12 for mode 2. This process is refined with a root-raised cosine filter, aligning the signal to the correct symbol rate for accurate data extraction. The final stage, RDS Data Processing, utilizes Manchester and differential decoding to convert symbols to a usable bitstream, ensuring frame synchronization and error detection for robust data integrity. The processed RDS bitstream, at a consistent rate, feeds into the application layer, producing decoded radio data that enhances user interaction by providing station information, traffic updates, and more, within the FM broadcast spectrum.

Implementation Details

Building Blocks:

The labs are focused on developing utility functions for our project and introducing digital signal processing (DSP). In the first lab, we aimed to understand and implement DSP basics for software-defined radios in Python. The four functions that were focused on in this lab were creating an impulse response for a low-pass filter (LPF), creating a convolution-based filter, and using Fourier transforms to verify the above create functions. The LPF's impulse response and a basic filter were easily implemented because they were based on mathematical equations and pseudocode that was provided with the lab document.

However, we encountered issues when implementing block processing. Understanding convolution and how to save states required effort and clear visualization of what we are trying to achieve. We learned to think of the state as overlapping values in a long line of data points and these overlaps is allowed us to use the entire array of filter coefficients. Once we understood this concept, implementing the code was easier for the time being.

Lab 2 involved converting our Python models to C++. This task was straightforward, but learning C++ specifics, such as vectors, required us to refresh what we have learned in previous course. Additionally, the refactoring of convolution with states to C++ also exposed a knowledge gap we had, in terms of how states truly worked to enable convolution. This lack of through understanding was only rectified with the help of the teaching assistant. Additionally, this lab also gave us exposer to GNU plots, this would be a crucial asset at later stages in the project, aiding with debugging.

Lab 3 served as project preparation. Through this lab we were able to set up the RF front end and Mono mode 0 in python. This lab also helped us understand and solidify the concepts further.

Intermediate frequency processing (RF front end):

RF Front End is used to obtain the data from input streams from the raspberry pi. This also involves processing the I and Q data points. Our implementation of RF Front End consists of two functions, processBlocksFromStdin and processBlock.

The processBlocksFromStdin function does various tasks, these include finding the lowpass filter coefficient for IQ processing as well as reading the data from the input streams. The data is read through a non conditioned for loop that increments an integer variable call block_id. Within this for loop the readStdinBlockData function from iofunc.cpp is called, this function was given as part of the base code and reads the data from the raspberry pi or the .raw files (user defined). It must be noted that this function reads one block of data and then executes the processBlock function.

The processBlock first extracts the in-phase (I) and the quadrature (Q) samples by looping through the input vector coming from processBlocksFromStdin. All the odd elements are placed in a vector that represents the Q samples and all the even elements are placed in a vector representing the I samples. After the I and Q samples are extracted, they are passed in to the convoDS function from filter.cpp, this function completes the low pass filtering that was initiated by finding the coefficients in the processBlocksFromStdin function. It also must be noted that convoDS is a function that does both convolution and down sampling. The I and Q samples are down sampled by a factor of rf_decim. After the down sampling the sampling rate has been stepped down to IF sample rate. The two output vectors of convoDS are then passed into demodulation function called CustomDemod from mono.cpp. This produces a combined demodulated signal called fm_demodulated_signal.

During the implementation stage, we noticed distortion in the output data of our convoDS function. This distortion became apparent when we compared our C++ function's output to that of a Python model developed in lab 3, which also performed convolution and down sampling. After investigating, we discovered that the distortion was caused by incorrect state handling within our convoDS function. It appeared that our function was not properly managing the state information needed for the convolution operation. As a result, the convolution process produced an incorrect output, resulting in the observed distortion in the down sampled signal. To address the issue, we focused on accurately managing state information throughout the convolution process. This included handling previous input samples correctly and updating the state buffer accordingly. Our goal was to ensure that the down sampled data was consistent with the Python model's output while also eliminating the observed distortion.

This error was caused when we combined the steps of down sampling with convolution, as this function was written after to make the design more efficient. Our initial design had convolution and down sampling as separate functions.

Mono processing:

In accordance with the RF Front End's operational protocols, the Software-Defined Radio's mode determines the processing parameters outlined in the team's constraints document. The mode of operation for mono processing is determined in the main function using conditional statements.
In all modes, the cutoff frequency for the low-pass filter is determined using the "impulseResponseLPF" function, both in the main function and during processing in "monoProcessingThread". The initial step in mono processing is executing the "popFirstElement" function which is the process of reading the demodulated signal from the queue filled by "processBlock". Then, conditional statements are checked against the demodulated signal. The first conditional statement checks if the user is trying to run in stereo mode. The second conditional statement checks to see if resampling is needed. If the program determines false for the first conditional statement, there are two paths through which the program could execute. These paths are determined by the mode of operation. Modes 0 and 1 do not require resampling, therefore performing convolution and downsampling via "DSconvo" in "filter.cpp". Modes 2 and 3, requiring resampling, execute "convoResample" from "filter.cpp". Regardless of the path taken in mono processing, the output will have a sampling rate equal to "audio_FS" from the constraints document. Notably, the output vector to which "DSconvo" and "convoResample" write to is defined in the main function as "audio_block". Once either "DSconvo" or "convoResample" finishes execution, the populated and updated "audio_block" is passed through a for loop. This loop converts the float-type audio results into integers, enabling Aplay to read the data. The conversion involves type casting each index in the "audio_block" vector and saving it to a new vector called "audioBlockInt". This is the vector read by Aplay.

The decimation factors are tailored to each mode: 10 for modes 0 and 2, 8 for mode 1, and 6 for mode 3. As mentioned, modes 2 and 3 require resampling and thus need an upscaling factor, called "audio_expand" in our program. This factor is calculated as the ratio of the output to the greatest common divisor (GCD) of the input and output sample rates. While modes 0 and 1 require no upsampling and have an upscale factor of 1, modes 2 and 3 require 147 and 441, respectively, followed by an "audio_decim" factor of 800 for mode 2 and 3200 for mode 3. For our resampler function, we utilize the impulse response generation function from previous labs to calculate the impulse response "h" using the sinc function. In lab three, the signal is only reduced by a factor of 10. To process all modes of operation, convolution requires both downsampling and upsampling. Running them separately is too slow for real-time operations. As instructed in the lecture, we combine upsampling and downsampling into the convolution function. The main idea is to only calculate the necessary components. To calculate input_size * U/D elements, we skip the 0s in the upsampling by tweaking the index, rather than performing the actual upsampling.

Initially, our mono processing functions were not validated, resulting in audio distortions during sample "testing.raw" files in mono mode 0. Based on our prior debugging experiences with the RF Front

End, we suspected a typo in previous lab code transfers or a problem with the newly written DSconvo function, which combines convolution and down sampling. To identify the problematic function, we tested each one against a proven version from previous labs. This testing included the “DSconvo” and “impulseResponseLPF” functions, as well as a separate “convoResample” test that used different convolution and resampling functions. The tests revealed that the error originated with the “impulseResponseLPF” function. A detailed comparison of our current “impulseResponseLPF” in filter.cpp to the Lab 2 version revealed a missing normalised cutoff in the denominator of our calculation. Correcting this error and subsequent testing confirmed that the distortion problem was resolved.

Like the RF Frontend, convolution and down sampling was initially two different functions, it was combined to make the process faster and allow for the program to have a lower execution time.

Stereo processing:

The stereo processing implementation in our project is structurally similar to the mono processing, but it introduces additional complexities with concepts such as Phase-Locked Loops (PLL) and Band-Pass Filters (BPF). Following the project guidelines, we divided our development into three main phases: carrier recovery, channel extraction, and signal combining.

Stereo processing is initiated through a conditional statement that checks to see if the user wants to execute the stereo processing. If the condition statement is true, the stereo processing begins. It also must be noted that the coefficients for the bandpass filter was calculated in the main fucntion before the execution of the conditional statements. This is done through the bandPassFilter function in filter.cpp. There are two sets of coefficients, one for the carrier recovery and the other one for channel extraction. After the conditional statement the first the delayBlock function is executed. This fucntion aligns mono and stereo signals in an SDR system by introducing a precise delay to the mono signal, effectively synchronizing it with the stereo path. By halving the number of taps, which are represented by the size of the state vector in the function, the delay introduced matches the processing time difference between the mono and stereo paths. After the delayBlock function, depending on the mode of operation either DSconvo or convoResample is executed, this calculates the delayed mono component of the stereo processing. Next the stereo function is called, in this function the bandpass filtering of channel extraction and carrier recovery are completed through the execution of DSconvo function, it must be noted that the down sampling factor is set to 1 as we do not require down sampling at this state. Next the output of the carrier recovery bandpass filter is pass into the PLL. The output of the PLL is then mixed with the output of the channel extraction bandpass filter. This is done through a for loop. Finally depending on the mode, using DSconvo or convoResample the final filtering is performed, and sample rate conversion takes place. This output is then combined with the mono audio to produce the final stereo output.

The Phase-Locked Loop (PLL) in Python was initially revised to include states for improved functionality. This Python model underwent significant changes when it was refactored into C++. The first attempt at refactoring involved passing a large number of variables as arguments, making the function calls difficult and time-consuming to understand. To address this complexity, an abstract class called PllState was created and defined in the dy4.h file. This simplified the function calls by combining the numerous arguments into a cleaner, more manageable format. The PllState abstract class is instantiated just before the PLL function is called, which effectively organizes the variables and simplifies the implementation process.

We ran a few tests to ensure that our updated Phase-Locked Loop (PLL) worked as expected. First, we plotted the PLL output for the entire block. We wanted to see if it resembled a sine wave as it should. We did this using a Discrete Fourier Transform (DFT) function from our fourier.cpp file. We also used logfunc.cpp functions to collect data for the plot. We then plotted the data using GNU plot. We noticed that the output of the PLL was not as expected, when we reviewed the code, we noticed that we had not changed the NCO scale to 2. After this fix and we ran another test, this time we were able to confirm the PLL worked properly within one block, we tested it between blocks. We saved the PLL outputs from two consecutive blocks in separate CSV files. File one contained the output from Block 0, while File two contained the output from Block 1. Next, we extracted the final 100 data points from the output of block zero and the initial 100 points from the output of block one. These points were merged in the order of block 0 and block 1. Through this test we were able to verify that the PLL switched between blocks as expected.

RDS:

The implementation for RDS has a similar start as stereo, the coefficients for the two bandpass filters and the coefficients for the lowpass filter and raised root cosine are calculated in the main function. Using the popFirstElement function, we started by retrieving demodulated I/Q signals from a shared queue. Our procedure went as follows: first, we used the DSconvo function to filter the signals inside the RDS frequency band (54 kHz to 60 kHz); next, we delayed and squared the signals to increase their non-linearity. After applying a second DSconvo for bandpass filtering (from 113.5 kHz to 114.5 kHz), we used the RDS_fmPll function to modify the Phase-Locked Loop (PLL). After mixing, convoResample was used to resample the output, then DSconvo was again used to run it through a low-pass filter before smoothing it out with a Root Raised Cosine (RRC) filter. To ensure correct data interpretation, we removed the in-phase and quadrature components for Clock Data Recovery (CDR) and carried out Manchester decoding. Finally, differential decoding was used to further isolate the signal. This was the extend to which we were able to implement RDS processing. We were also not able to verify some these steps as we had run out of time. The process of validation only got till the RDS PLL. The accuracy of the following function and operation have not been verified.

We were able to verify that the output that was produced by the PLL was acceptable within reason. The method used for validation was very similar to the one used in validating the stereo PLL. Initially, the RDS PLL test revealed that out PLL was producing incorrect errors. Upon investigating we realised that the ncoScale was set to its default value of 1.0. After the adjustment was made to the ncoScale to be 0.5, the expected output was observed from the RDS PLL.

Multithreading:

In our project, we used a multi-threaded design to process digital radio signals, focusing on tasks like mono processing, stereo processing, and Radio Data System (RDS) decoding. A shared queue is at the foundation of our approach, allowing threads to transfer demodulated I/Q signals more efficiently. One of our threads is dedicated to running the processBlocksFromStdin function, which reads block data from the standard input, demodulates each block, and stores the demodulated signals in a shared queue. This process is critical for maintaining a consistent stream of data for subsequent processing.

Another thread handles the monoProcessing function, which includes stereo processing or the RDS function, depending on our project's requirements. This thread receives processed I/Q signals from a queue and performs specific processing steps such as filtering, decimation, and decoding of mono or RDS signals. Mutexes and condition variables are used to manage shared queue access and thread coordination. This approach eliminates data corruption and race conditions, ensuring that our data flows properly and remains intact.

In our project, we encountered a significant limitation: the inability to run RDS and mono processing tasks simultaneously. This limitation results from the lack of a mechanism in our system that allows dual thread access to a single queue element. As a result, a choice had to be made between using the RDS or mono processing at any given time. Moving forward, we intend to address this shortcoming by developing an improved queue management system. These changes are expected to allow simultaneous processing across both pathways, removing current operational constraints.

The popFirstElement function is an essential part of our project, designed to retrieve the first element safely and efficiently from a shared queue shared by both producer and consumer threads. This function runs within a consumer thread and starts by locking a mutex to ensure exclusive access to the queue. It then waits for the cv_not_empty conditional variable, which pauses the thread's operation until the queue is confirmed to be not empty or the processing is marked as complete. This waiting mechanism prevents the consumer from trying to access an empty queue, which could result in errors or inefficient looping.

When the queue is verified to have at least one item or the process is completed, the function safely removes the front item from the queue. This item is then assigned to a fm_demod variable, preparing it for further processing by the consumer thread. To maintain efficient queue operations and signal availability for new items, it notifies one of the waiting producer threads using the cv_not_full conditional variable, indicating that space is available for more elements. This notification is critical for preventing queue overflow and ensuring that producer threads only operate when they can add new elements without exceeding the queue's capacity limits.

Analysis and Measurement

Path	Multiplication and Accumulation
Mono Mode 0	((2*240e3*101)+(48e3*101))/48e3 =**1111**
Mono Mode 1	((2*144e3*101)+(36e3*101))/36e3=**909**
Mono Mode 2	((2*240e3*101)+(44.1e3*101))/44.1e3= **1200.3**
Mono Mode 3	((2*320e3*101)+(44.1e3*101))/44.1e3=**1566.8**
Stereo Mode 0	((240e3*2*101) +(240e3*2*101) + (240e3 *6) + (240e3) +(48e3*101*2)) / 48e3= **2257**
Stereo Mode 1	((144e3*2*101) +(144e3*2*101) + (144e3*6) + (144e3) +(36e3*101*2)) / 36e3 = **1846**
Stereo Mode 2	((240e3*2*101) +(240e3*2*101) + (240e3 *6) + (240e3) +(44.1e3*101*2)) / 44.1e3= **2237**
Stereo Mode 3	((320e3*2*101) +(320e3*2*101) + (320e3*6) + (320e3) +(44.1e3*101*2)) / 44.1e3= **3184**

From the tables provided, we can observe that the measured runtimes correspond with the computational analysis. The runtime per sample follows the predicted trend based on the number of multiplications/ accumulations. Modes 0 and 1 have similar runtimes, which is expected due to their operational similarity. However, Modes 2 and 3 exhibit an increase in runtime, aligning with the anticipated additional computational load. The stereo processing paths approximately double the runtime of the mono paths, which fits with the additional channel processing. The comparison between the runtime of mono Modes 2 and 1 reveals a discrepancy; the analysis suggests a moderate increase, yet the measurements indicate a more substantial increase. This discrepancy can be explained by the additional computational effort required in Mode 2, especially due to the 2 times multiplication factor in the resampler block which is not present in the standard convolution used in runtime calculations. The filter blocks are the most time-consuming among all the components, making the convolution operations the primary computational bottleneck in this project.

	Mode 0	Mode 1	Mode 2	Mode 3	NumTaps = 301	NumTaps = 13
RF Front End Processing						
ImpulsResponse LPF	38	44	40	42	98	12
Extractring I and Q	467	545	1131	744	599	833
convoDS	4493	6730	5403	10626	12516	1188
CustomDemod	54	65	63	124	50	99
Mono Processing						
Dsconvo - Low Pass	886	1703			1259	2475
convoResample - Low Pass			4032	6301		
Stereo Processing						
delayBlock	27	22	34	91	46	29
DSconvo	871	871			961	872
convoResample			1181	1950		
Dsconvo - Carrier (Bandpass)	4344	3478	5538	10904	13898	470
Dsconvo - Channel (Bandpass)	4355	3493	5428	10868	13445	446
fmPll	1007	842	1235	2470	1031	1003
Mixer	13	9	14	30	14	12
Dsconvo - Stereo	885	881			1002	872
convoResample - Stereo			1194	1773		
Stereo Combiner	6	4	6	9	9	5

*The measured time is all in microseconds

	Mode 0	Mode 1	Mode 2	Mode 3
Mono	0.021sec	0.028sec	0.00018sec	0.0000907sec
Stereo	0.021sec	0.028sec	0.00018sec	0.0000907sec

*Total number of seconds per block

The analytical analysis predicts that changing the number of taps will result in proportional changes in computation load, as shown in the tables above. Because the number of operations correlates linearly with the number of filter taps, we expect the RF Front End's operations and associated runtimes to decrease significantly when only 13 taps are used, but to increase when 301 taps are implemented. The runtimes shown in the table confirm this expectation, albeit with some variation. For 301 taps, the runtimes increase by about threefold. Using 13 taps reduces runtime by threefold, rather than the expected factor of ~10. This could be due to output stream bottlenecks. The Mono Path filters show a similar pattern. With a linear relationship, the runtimes for 301 taps triple, while the reduction for 13 taps is not quite tenfold, indicating an acceptable outcome given the considerations for output load. The Stereo Processing follows suit, with the Stereo Carrier BPF and Stereo Extraction BPF changing runtimes based on the number of taps. When we compare using 13 taps to 101 and then 301, we see the runtime goes up, but so does the sound quality. With 13 taps, the audio is noisy and distorted, especially with the higher pitches. But with 301 taps, everything sounds much cleaner and sharper.

Proposal for Improvement

The productivity of development can be improved by implementing the core functionality of the SDR in Python before translating it to C++ like it was done in the labs. The debugging process was slower because we did not take advantage of modelling the channels in Python first. Python offers many libraries for digital signal processing, including filters, which gives us more time to work on the implementation process. Moreover, Python's matplotlib library facilitates easy plotting of data, which is crucial for visualizing signals and debugging. By utilizing Python for rapid prototyping and algorithm development, we can iterate more quickly and efficiently. To implement this, we can start by rewriting the signal processing and filter modules in Python, utilizing NumPy and SciPy for efficient array operations and filter design. Additionally, integrating a plotting interface using matplotlib will enable real-time visualization of signal spectra and time-domain waveforms. Once the functionality is validated in Python, it can be ported to C++ for performance optimization and integration into the existing SDR framework.

Implementing a more versatile queue system from the start, capable of handling multiple accesses, could have avoided the current limitation, in which RDS and mono processing cannot run concurrently. This foresight in design eliminates the need for a full redesign to address this issue. Alternately, adding a separate, dedicated queue for RDS to access demodulated signals would streamline our processing workflow from the start, eliminating bottlenecks and increasing efficiency. By anticipating these requirements early in the development process, we could have avoided the current constraints, resulting in a more flexible design capable of handling simultaneous signal processing tasks without requiring significant future modifications.

Introducing a GUI for the front end can enhance user experience by providing an intuitive interface for controlling radio channels, adjusting parameters, and visualizing signals in real-time. This eliminates the need for users to interact with the software through command-line interfaces or piping commands, making the system more accessible to non-technical users. To implement this, we can leverage Python libraries to develop a user-friendly GUI. The GUI should include features such as channel selection (mono, stereo, RDS), frequency tuning, modulation selection, filter configuration, and real-time spectrum visualization.

6 Project activity

Week	Activity
February 12	Project Specification released
February 19	No Progress
February 26	**Biswajit**: Read project specification with teammates identified constraints and setup project plan **Maham**: Read project specification with teammates identified constraints and setup project plan **Jibin**: Read project specification with teammates identified constraints and setup project plan. **Sai**: Read project specification with teammates identified constraints and setup project plan
March 4	**Biswajit**: Started working on downsampling implementation and mode parameters based on constraints. Debugged issue where we left out the Norm_cutoff from the denominator of our impulseResponseLPF function, which caused lots of noise when testing. Not sure how the norm_cutoff was missing, we think it was accidentally deleted. Debugged this issue by tesing our c++ functions (with our working python functions (firwin and lfilter) by printing the output. **Maham**: Reformatted reading of raw file from python to cpp. Plan was to start with raw file before using RF dongle. Updated the function calls in mono.cpp with appropriate arguments. Tried writing data to a file. **Jibin**: Reformatting of python functions to cpp from labs. Finished initial front end and mode 0 in mono.cpp file. Converting Lab 3 BlockProcessing file into C++. Making new setting up the initial layout for the project.

	Sai: Reformatted python codes from labs to C++. Resampling and decimator implementation for mono to account for Modes 2 and 3. (Resampling had some issues which had to be confirmed with TA; the issue consisted of an error which I made in the upscaling factor for our constraints).
March 11	**Biswajit**: Implemented convolution with down sampling. Implemented resampling for mode 2 and 3. Calculated and Declared mode parameters in interface (all decim and expand factors). **Maham**: Studied constraints document to understand how mode parameters are set. Team decided to move everything from mono.cpp to the project I worked on the processBlock function. Checked if bandpass filter works by using it in lab code. **Jibin**: Fixed front end and implemented the interface. Fixing segmentation errors. Suing Valgrind. Moved the front end and mono processing from mono.cpp to project.cpp file with major changes. Fixed static in mode 0 and 1. **Sai**: Code Implementation for Bandpass filtering, resampling into stereo. Wrote code in python and verified filtering and resampling works using the graphs. Converted the code from python to C++. PLL conversion from python to C++. State saving was implemented into the PLL code.
March 18	**Biswajit**: Debugged issue for when testing modes 2 and 3, gaps in audio when it is playing. Issue was with block size; we were including up sampling factor in the calculation. TA advised us that it was not needed and therefore, fixing that fixed that error. Implemented delay block. Coded the refactoring of the pll from the python pseudo code with sai. **Maham**: Observed debugging and tried to learn the process for future debugging. Finished Stereo **Jibin**: Tested mono with working resampler, Started Stereo. Fixed memory leak. Finished stereo. Worked on initial threading. **Sai**: Stereo (fixing coding errors in python; band pass's cutoff frequencies) and finished it. RDS - PLL and RCC conversion from python to C++ with Biswajit (had trouble understanding quadrature). Researched Manchester decoding and tried to write pseudocode and verify with TA.
March 25	**Biswajit**: Calculated and implemented new parameter (new decim, expand factors and sampling frequency) for RDS mode 0 and 2. Researching and learning about RDS, Manchester and differential decoding. **Maham**: Followed RDS processing path to write initial RDS channel extraction, carrier recovery and demodulation. Participated in stereo mode 2 and 3 debugging by checking calculations of mode parameters (My teammates found the problem before me). **Jibin**: Worked on RDS demodulation (CDR) and Manchester and differential decoding. Implementing the initial RDS from bandpass filter to PLL. **Sai:** Understanding and writing code for Manchester and differential decoding with Jibin. I had difficulty understanding the process of locking; in which I was able to know to think of it as needed for thread safety. Used online resources to get visual representation of Manchester decoding alongside the assistance of TAs.
April 1st	**Biswajit**: Finish report and presentation **Take home:** What is the importance of normalized cut off frequency? The normalized cutoff frequency, calculated as (Fc / (Fs / 2)), is essential in digital filter design, making the filter's performance consistent across different sampling rates by scaling the cutoff frequency (Fc) relative to half the sampling rate (Fs/2), the Nyquist

frequency. This approach ensures the filter behaves uniformly in various systems by keeping the ratio of cutoff frequency to sampling rate constant. In our project, this concept is applied in the `impulseResponseLPF` function to create a low-pass filter (LPF) that isolates desired frequencies and eliminates noise. The function calculates the filter's impulse response for a given number of taps, affecting the filter's selectivity. By using the normalized cutoff frequency, we ensure our LPF adapts to different sampling rates, crucial for the effective demodulation and processing of digital radio signals, thereby maintaining signal integrity and enhancing performance.

Test VMware with different block size:
Performed testing through VMware and changing the number of blocks caused my audio to be not continuous. There was also segmentation fault with bock size was too high.

Maham: Finish report and presentation
Jibin: Finish report and presentation
Sai: Finish report and presentation
Take-home Exercise:
Question: What is the logical explanation for why the value for nCout is supposed to be what it is
Answer: In the function fmPLL for stereo, parameter nCout is set to 2 so that the output cosine has the correct frequency. It is supposed to be greater than the value of PLL. We know that the frequency of PLL is 19 kHz, therefore the frequency of nCout will become 38 kHz (double the amount). Initially, it was important to understand how the logic of a DSBSC modulation was used. In DSBSC, the message signal is multiplied by the carrier signal, resulting in a modulated signal with two sidebands but with no carrier. The following equation;

$$\frac{A_m}{2}[cos(2\pi(f_c+f_m))+cos(2\pi(f_c-f_m))].$$

was crucial in my understanding the visual representation of stereo.
To my understanding, the stereo channel is between 22 and 54 kHz. We want the pilot tone to be at the middle most point. The math for this would be (22 kHz + 54 kHz)/2 = 38 kHz which can be seen in the project document to be the frequency of the nCout. For the output of stereo, we double nCout and as a result, we get the output of stereo duplicated and at a lower power between -16 kHz and 16 kHz (-16 kHz to 16 kHz is 32 kHz in total which stems from 54 kHz – 22 kHz). The same logic is being used for RDS, however, because we are taking the duplicated pilot tone for RDS, we divide nCout by 2 rather multiply by 2 (we use a scale factor of ½ rather than 2). In order to confirm this, the following math was done; 114 kHz (from PLL) divided by 2 is equal 57 kHz. Initially, by just looking at the lecture slides (the first couple slides), I had assumed that the nCout scale was 1, however, after reading the project document, lecture slides (the bottom slides) and understanding the logic, I am able to determine how and why the nCout is why it is what it should be.

7 Conclusion

This project highlighted the importance of critical thinking when developing a solution for a task. We learned how to implement our previous knowledge to solve a problem and explored new functionalities in Linux commands, multithreading, integrating hardware and software together, and real-time integration. We were also able to improve existing skills, such as testing and debugging, as they were necessary tasks we had to perform numerous times throughout the duration of the project. Overall, while this project was quite challenging, it was very rewarding in lessons. It is a memorable experience and will be very useful for our future careers as computer engineers.

8 References

[1] 3DY4 Project Module, Documentation, "COE3DY4 Project Real-time SDR for mono/stereo FM and RDS" Bachelor of Engineering, McMaster University [Online].

[2] 3DY4 Project Module, Documentation, "3DY4 Project - Custom Settings for Group 6[5]1" Bachelor of Engineering, McMaster University [Online].

[3] "bitset - C++ Reference," cplusplus.com, n.p., n.d. [Online]. Available: https://www.cplusplus.com/reference/bitset/bitset/.

[4] "C++ Containers Library," cppreference.com, n.p., n.d. [Online]. Available: https://en.cppreference.com/w/cpp/container

[5] "SciPy Tutorial — SciPy v1.8.0 Manual," SciPy.org, n.d. [Online]. Available: https://docs.scipy.org/doc/scipy/tutorial/index.html#user-guide. [Accessed: 05-Apr-2024].

[6] R. Thelin, "A tutorial on modern multithreading and concurrency in C++," Educative, [Online] https://www.educative.io/blog/modern-multithreading-and-concurrency-in-cpp [Accessed: 05-Apr-2024].