# COE3DY4 – Project Report

Huner Dogra, Sarbjot Ghotra, Isaac Thomas, Charlotte Funnekotter

Group 65

April 5, 2024

## 1: Introduction

The objective of this project was to design a software-designed radio (SDR) system in C++. The system receives an input of a frequency-modulated (FM) signal that contains mono, stereo, and radio data system (RDS) information. The signal is processed on a Raspberry Pi through different modes, each with unique constraints on the sampling frequency. Mono and stereo audio are then outputted from this, as well as information about the song and the radio station broadcasting it. This process is completed in real-time, allowing users to listen and obtain data as it is being broadcasted.

## 2: Project Overview

This project makes use of the Realtek RTL2832U chipset with radio frequency (RF) dongles and the Raspberry Pi 4 for the implementation of a real-time SDR system. The goal was to receive frequency-modulated mono and stereo audio, as well as additional data broadcast using the radio data system (RDS) protocol, in real-time. This was implemented in C++, classifying the system as a software-defined radio (SDR) system. The incoming radio signals were fully processed in a C++ program running on the Raspberry Pi; this program would then output audio that could be listened to in real-time.

The SDR has four modes, each with different input, intermediate, and output sampling rates. The mode impacts the factors of which the data must be decimated and expanded. The specific rates for each mode are highlighted in Figure 2.1. Note: the IF for Mode 3 was updated to 384 KSamples/sec to avoid too much data loss. For the RDS portion of the project, only modes 0 and 2 are available. Each of these two modes has a different samples per symbol rate; these are highlighted in Figure 2.2.

| Settings for group 65 | Mode 0 | Mode 1 | Mode 2 | Mode 3 |
|---|---|---|---|---|
| RF Fs (KSamples/sec) | 2400 | 1440 | 2400 | 1920 |
| IF Fs (KSamples/sec) | 240 | 288 | 240 | 128 |
| Audio Fs (KSamples/sec) | 48 | 36 | 44.1 | 44.1 |

*Figure 2.1: Sampling Frequency Settings for Group 65*

| Settings for group 65 | Mode 0 | Mode 2 |
|---|---|---|
| Samples per symbol (SPS) | 16 | 27 |

*Figure 2.2: Samples per Symbol Settings for Group 65*

The RF hardware (dongles) receives the analog broadcast signal and splits the data into in-phase (I) and quadrature (Q) components. The software program's front-end block takes these samples and passes them through a low-pass filter, decimating them at the same time to achieve a specific intermediate sample rate (this is based on the mode of the SDR). The I and Q data is then combined in an FM demodulator, and this demodulated data is passed on to the mono, stereo, and RDS blocks.

The mono block extracts the mono channel (0 to 15KHz) from the demodulated signal using a low-pass filter, upsampling and downsampling at the same time to achieve the correct audio sampling frequency. The output of this block is the processed mono data.

The stereo block extracts the stereo channel (23 to 53KHz) and the pilot tone (19KHz) from the demodulated signal using band-pass filters. The pilot signal is fed through a phase-locked loop (PLL) and numerically controlled oscillator (NCO) to bring it up to 38KHz, where it can be mixed with the extracted stereo data. The mixed signal is brought to the baseband, where it can be filtered and converted to the appropriate sample rate in the same way as in the mono path. Finally, the stereo samples are added to or subtracted from the mono samples (which are delayed to match the delay of the band-pass filters) to recover the left and right audio channels.

The RDS block recovers the RDS channel (54KHz to 60KHz) and the subcarrier (114KHz) from the demodulated data using band-pass filters. The subcarrier is obtained by squaring the RDS data before putting it through the filter. This filtered subcarrier is then put through a PLL and NCO like in the stereo path, but in this case, it moves the tone to 57KHz so it can be mixed with the (delayed) RDS data. The mixed data, now at the baseband, is filtered and resampled according to the mode. It is then put through a root-raised cosine (RRC) filter to minimize inter-symbol interference and prepare it for clock and data recovery. In clock and data recovery, a bitstream is created corresponding to the highs and lows of the samples. This bitstream is Manchester encoded, so it must be decoded to produce another bitstream which can then be used for frame synchronization. Once synchronized, valid frames are sent to the application layer, which extracts data about the audio.

The frontend, mono/stereo path, and RDS path all run in parallel to facilitate real-time processing.

## 3: Implementation Details

Mono Path

Beginning the first stage of this project, our team consolidated the work completed from the three course labs to create a foundation to build our SDR. Since the only new function that needed to be developed for this section was the resampler, we opted to model all the other components first to ensure a working baseline. This included copying over the LP filter, the block convolution function, and the custom FM demodulation function.

Mode zero and one were relatively straightforward to implement using the given functions, however the performance was inadequate since we hadn't yet implemented optimizations. Therefore, the next stage involved creating a fast downsampler, which ensured that values in the convolution that were lost during the downsampling process weren't needlessly computed. This provided a solid performance improvement, which was a critical component in developing the resampler.

Next, the slow upsampler was implemented, which allowed us to test our mono mode with both upsampling and downsampling capabilities. This verified that our mono mode thus far was accurately recovering the signal. Finally, the fast resampler was implemented once all the other components were deemed recoverable, essentially adding the fast upsampling component. To validate the output of our project, we modified the provided Wavio python script to create WAV audio outputs for our model.

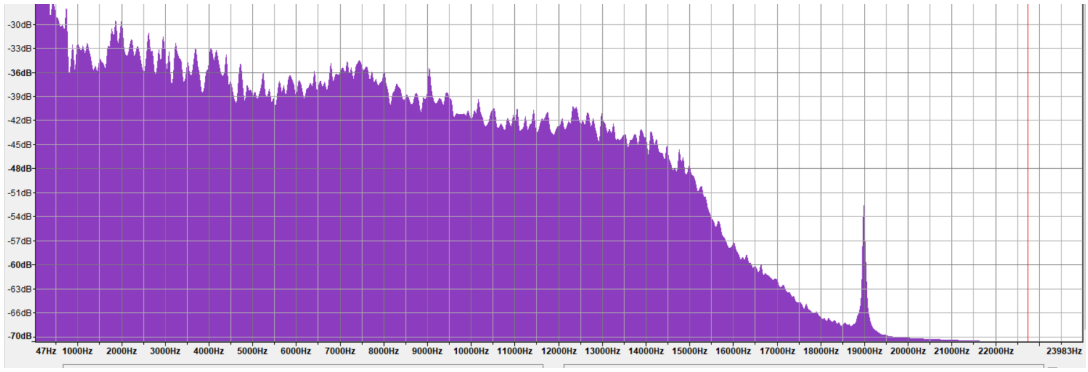


*Figure 3.1: Plot of mono C++ output in the frequency domain, with clearly recognizable pilot tone at 19KHz*

Once all these components were verified to be operational, the model was refactored to C++. Some parts did not work as intended, so we used Gnuplot to create plots at key points of the signal flow graph, allowing us to pinpoint inaccuracies in our refactoring. Furthermore, using Audacity, we ensured that the data we were extracting in the frequency domain was uncorrupted; through key features such as the 19KHz pilot tone at the intermediate stages.

Stereo Path

When embarking on the process of adding stereo functionality, we relied heavily on the functionality created in the mono audio path and its associated functions in C++. We began by adding the simplest functions that were required on the stereo data path, such as the pointwise operations (add, subtract, multiply), as well as the bandpass filter, which was simply a modification of the low pass filter. Plotting the Fourier domain outputs at each stage allowed us to understand how the program was working without requiring an audible output.

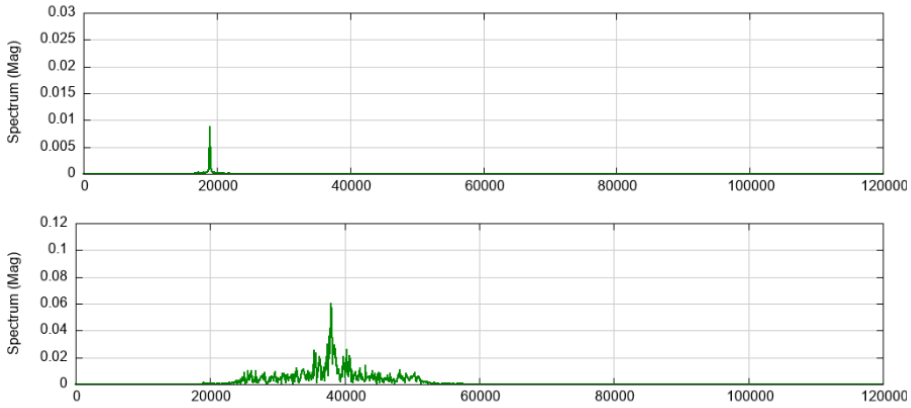


*Figure 3.2: Plots used for debugging, top is pilot tone isolation; bottom is stereo band isolation*

By making use of theoretical concepts such as bandpass filters, modulation and demodulation, we were able to make predictions on what we expected the signals to look like versus what was obtained. This allowed our team to trace down specific issues with a high degree of confidence.

Once these components were deemed to work as intended, the PLL/NCO with state saving was developed next. This allowed us to lock on to the 19KHz pilot tone, ensuring the phase of the signal was aligned. By testing for continuity in the PLL, we were able to determine if we were locking onto the phase correctly.

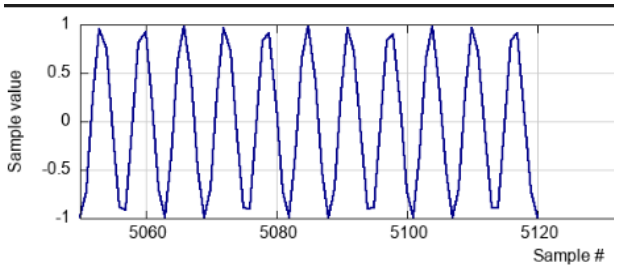


*Figure 3.3: End of a particular block's PLL output*

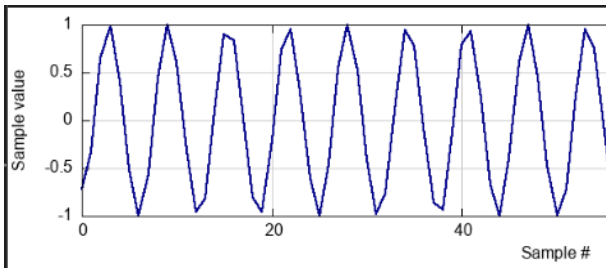


*Figure 3.4: Start of adjacent block's PLL output*

Analyzing the above figures, it was evident that there was a discontinuity between PLL blocks, confirming that this was a block state-saving issue. By investigating this matter further, we were able to correct the bug, providing a completely continuous PLL output between blocks. As another precaution, the block output of the PLL/NCO was generated in the frequency domain. This confirmed that the pilot tone had indeed been shifted to 38KHz in preparation for mixing with the stereo band-passed signal.

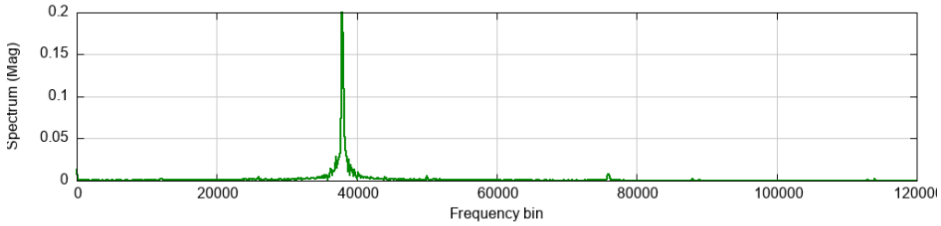


*Figure 3.5: Output of PLL and numerically-controlled oscillator in frequency domain*

Once this stage was completed, the PLL output was pointwise multiplied with the isolated stereo band to bring it down to the baseband frequency, i.e. centered around 0Hz. Finally, the last piece of the signal flow graph was added, which was the all-pass filter. This synced the phase of the mono-path output with the stereo path. Finally, the pointwise addition and subtraction functions were used to merge the mono and stereo paths, by adding and subtracting the stereo path from the mono path to obtain the left and right channels.

As a final step for validation, we used a modified FM signal dump in which each channel played a different song. This allowed us to confirm that our design was indeed obtaining the left and right channels in an isolated manner, without leakage of any one song into the wrong channel.

Multi-Threading

To optimize our code and ensure its ability to run in real-time, we implemented multi-threading. This was a vital step for this radio to produce an uninterrupted live signal. This was implemented using a queue in combination with mutexes, and condition variables. To simplify this task we created a separate

class to handle the usage of the queue, and ensured that it was thread safe. In this class we defined two separate methods; enqueue and dequeue. In the enqueue method we facilitated the push of a desired element into the queue and used the condition variable to send a notification. This notification is received in the dequeue method, where a while loop is locked while the queue is empty (and until the notification is received). This lets the program know that the queue has an element and that it can proceed with dequeuing. By using this class we can simply queue and dequeue in the main class when necessary, without having to manually lock and unlock. In our main class, elements from the output of the FM-demodulation stage were added to the queue (in the front-end stage). These elements were then simultaneously taken from the queue for the audio processing stage (back-end). This process would repeat itself and run concurrently for all blocks of the signal.

When we first added this to our code it would not exit our while loop once an element was queued. Instead, it would get stuck in an infinite loop. To identify this issue we added a print statement where the element was being queued. This allowed us to verify that the elements were being queued as expected. Then we placed a print statement where the element was being dequeued, which verified that this is where the code was stuck. To address this, we inspected both the separate class and the main class to see where the issue could have arised. Once we did this, we realized that we had two while loops (one in the main class, and one in the thread class) that were both waiting for an element to be queued. Once we removed the loop from the main class the code worked as expected, and we were able to move onto RDS.

RDS Path

The RDS portion of the project was not implemented in C++ due to time constraints and therefore did not function in real-time. However, the RDS processing flow was fully modeled in Python.

The first step of RDS processing was channel extraction. To do this, we first used signal.firwin() function from the scipy library to find the filter coefficients for the desired RDS signal. The lower and upper bounds of the filter were chosen as 54 KHz and 60 KHz respectively, as this is where the RDS signal starts and ends. We used the given fmDemodArctan() function to demodulate the FM signal, then convolved this with our filter coefficients (using our convolve function refactored from C++) to produce a filtered output.

The next step was carrier recovery, which was used to obtain the carrier of the RDS signal without the pilot tone. First, we squared the filtered signal and produced a noisy 114 KHz signal (double of the 57 KHz subcarrier). Then, we use the same process as outlined before (using the scipy library to generate filter coefficients and convolving that with the signal to produce a filtered output). The lower and upper bounds of the filter were chosen as 113.5 KHz and 114.5 KHz respectively. After this, the filtered signal is put through a PLL and NCO to produce a clean, phase aligned 57 KHz signal. We adjusted the parameters of the function as specified in the project description, to accurately recover the RDS carrier. Lastly, we implemented an all pass filter on the signal coming out of the channel extraction stage (before squaring). This is done to match the delay introduced by the bandpass filter when processing the 114 KHz signal. This signal will then be mixed with the recovered RDS carrier in the demodulation stage.

Within the demodulation phase of RDS, the recovered carrier is first mixed with the main RDS channel. This mixed signal is moved to baseband and put through a low pass filter so that it can be isolated before proceeding. The cutoff frequency for this low pass filter was equal to 3kHz so that the desired half of the signal can be obtained. Simultaneously during this filtering, the signal is being resampled according to the parameters provided in the constraints document for our group. Using the provided samples per symbol value, we can determine the sampling frequency that the signal should be resampled to (symbol rate). The SPS for mode 0 and mode 2 is 16, and 27 respectively. The mode 0 output frequency is 38kHz and the mode 2 output frequency is 64.125kHz. The resampled and filtered signal is then convolved with the impulse response generated by the Root Raised Cosine Filter function. This applies a gain to the in-phase data while also combating inter-symbol interference. This final convolved signal acts as the input to the Clock and Data Recovery (CDR) section.

The CDR phase consists of Manchester encoding the data from the signal so that it can be prepared for RDS Data Processing. The encoding is done through a method we created called 'manchesterEncode'. Within this encode function, it will first synchronize the encoding by finding the max value within the first two periods of the signal. This max will be used as a starting point for the next for loop where the value of the signal will be recorded in an array every period. Once per block, this value will be checked against a threshold value of 0.05 to make sure that the recorded data is near a peak. If it is not, the synchronization is performed again. At the end of the encoding process, the array will be sent to the Manchester Decoder where it will be converted to a binary stream of data for frame synchronization.

To debug issues in both our understanding and implementation of the RDS frontend, we used constellation plots. Figure 3.6 (left) demonstrates that initially, we were not correctly isolating the in-phase and quadrature components of the signal. Additionally, our Manchester encoding function was not operating as expected. By identifying and correcting the issues in the RDS frontend and encoder, we obtained a constellation plot that indicated that the data was indeed recoverable (shown in Figure 3.6 on the right).

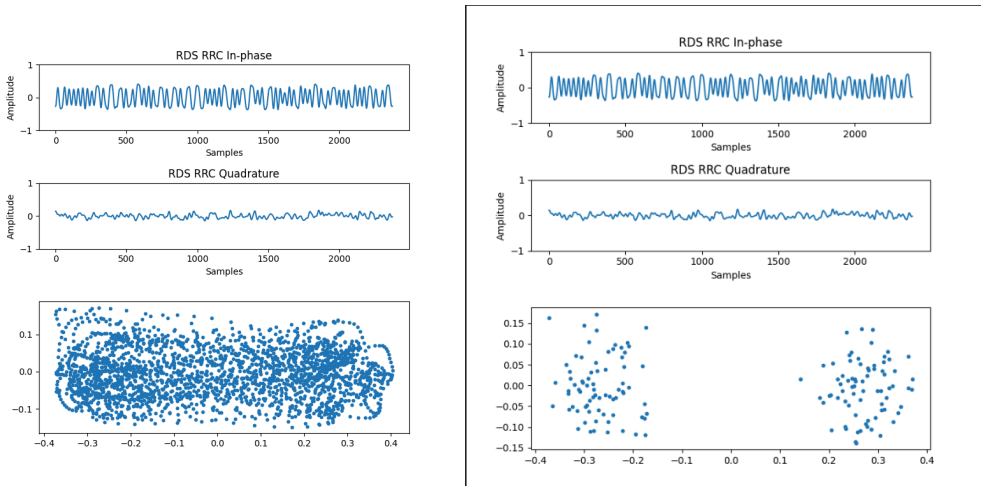


*Figure 3.6: Initial and Final Constellation Plots respectively*

The next step was decoding the Manchester-encoded data. In Manchester encoding, each pair of symbols ("H" and "L") represents one bit; we used 1 for "HL" and 0 for "LH". Before decoding the symbol stream received from clock and data recovery, we had to know which symbols to pair for decoding. To accomplish this, we analyzed a few blocks to check which decoding pattern made the most sense. We skipped the first five blocks entirely to allow the data to stabilize, then we checked blocks 5 through 9 for the correct pattern. For each of these blocks, we paired the symbols in both possible patterns (one with using the last symbol of the previous block, one without) and counted the number of Manchester violations, e.g., pairings of "LL" or "HH". The pairing pattern with the fewest violations was the pattern used for decoding in the rest of the blocks.

With the pairing pattern established, Manchester decoding was a simple matter of taking each pair of symbols and converting it to a bit to be added to a bitstream. Additionally, the output bitstream was differentially decoded, which was implemented by using an exclusive-or operation between the current bit and the previous Manchester-decoded bit. This double-decoded bitstream was then passed on to the frame synchronization and data extraction portion of the program.

Each frame of RDS data is 26 bits, and these frames could be extracted from the decoded bitstream. However, before attempting to extract data from the frames, we had to ensure that the frames were synchronized; otherwise the extracted data would make no sense. To do this, we tried frame synchronization for several sampling windows until a window matched a syndrome check, which confirmed that the frames were synchronized. The sampling windows were retrieved from the bitstream.

If not synchronized, the window would advance one bit at a time until a valid frame was found; otherwise, the window would advance by 26 bits each time, each window a valid frame. If for whatever reason synchronization was lost (indicated by a series of frames that were out of the expected order), the window would begin moving one bit at a time again in an attempt to resynchronize.

For each sampling window, we used matrix multiplication with the parity check matrix (see project specification) to produce a 10 bit syndrome. This was done by manually defining which bits of the message would be 1 and which would be 0, and adding up the values accordingly (to define each bit of the syndrome). The final result was then modded by 2 to produce the appropriate bit (instead of an integer larger than 1). Once each bit of the syndrome was defined, we compared the calculated syndrome to the predefined ones. If any syndrome matched, that meant that a valid offset was identified in the given message signal. If none of the syndromes matched, the sampling window (using the same method as described above) would be advanced by 1 bit and the search would repeat itself. Once a syndrome matched, the message signal to be passed to the application layer was defined as the first 16 bits of the signal received (to disclude the parity with offset bits). Then, the sampling window would advance by 26 to obtain the next set of data. This was done because due to the specification RDS, once the message is synced all valid syndromes should be found 26 bits away from each other. Since each syndrome is associated with an offset and each offset is associated with a specific type of data, identifying the offset would allow us to classify the information that we output in the application layer.

A check for false positives was also implemented by creating a variable to hold the last position where a valid syndrome was found. If a new valid syndrome was found less than 26 bits from the previous one, it is likely a false positive. In order to effectively debug our code if any issues were to arise, we used print statements to show the position of each bit found and the offset associated with it. When we implemented this we found that we were identifying every block type during every check (including false positives). We knew this was not possible due to the nature of the data coming in (it was not possible to have all the data in order all the time, as it would defeat the purpose of frame synchronization). To debug this we had to scrutinize our code line by line and implement more print statements at other areas of the function. This allowed us to identify exactly where our error occurred, and address it accordingly. Once this was done we moved onto the application layer.

The RDS Application Layer is the final step in the RDS phase of the project. This is where the information in the decoded and frame-synchronized bitstream can be converted into a message that can be displayed. Firstly, a Python dictionary utilized key-value pairs to convert both the program type codes and program service names from their corresponding binary values to text. The program type codes utilized the RDS standard table to implement the correct values. The program service names implemented the binary code of each character in the English alphabet, along with key symbols and numbers. This allowed the application layer to decode information such as the name and album in which a song belonged to. The program service name is determined by decoding the first 16 bits of message D. Message A contained the program identification code in binary and message B contained the program type code and decoder control bits.

By logging when a valid frame was found, we discovered that the frames were being found in the correct order starting at block 10; once the frame sequence was established, it was properly synchronized and stayed that way until around block 30. After this, it would lose synchronization and attempt to resynchronize, but it would never find another valid frame. We did not have time to discover why this was the case, but we started looking into it by logging the symbol or bit stream in every step of RDS processing and verifying that they were what we expected. With more time, we could have narrowed down the problem and fixed it so that even when synchronization was lost, it would properly resynchronize to the next set of valid frames.

## 4: Extensive Mode 3 Debugging

After completing the mono audio processing portion, we discovered that there was noticeable distortion in our mode 3 which we later found to be present in stereo audio as well. We initially thought

that there was an indexing issue within our resampler that was being masked when using certain parameters. To address this we first went through the code line by line to ensure that the correct parameters were being passed in, and that they were being scaled appropriately. Once we determined that the correct parameters were indeed being used, we altered our code to use a slow downsampler. We did this because we knew the slow downsampler had no errors, but since mode 3 required both downsampling and upsampling to achieve the desired output frequency we only implemented the slow downsampler for the front-end decimation. When this did not fix the issue, we concluded that our resampler was not failing during front end decimation. However, our issue still may have been with back-end resampling, so we continue to analyze our code. Following a rigorous inspection of the code, we noticed that Nyquist was not being met with our front-end cutoff frequency. This was fixed, but the distortion persisted. After meeting with TAs and Dr. Nicolici, we theorized that block size may have been the cause of this noise. We redid our calculations to ensure that our block size was in the appropriate range. We also used different block size values within that range to see if that would have an effect. Unfortunately, this did not help so we had to continue debugging to find the issue.

At this stage we had another meeting with Dr. Nicolici where we were advised to make a table of our decimation parameters (front-end decimation factor, upsample factor, and downsample factor), as well as the intermediate and final frequency values. We were instructed to change these parameters to produce many varied test cases. By identifying which parameters worked and which did not, we could establish a pattern to find exactly where our code fails. Additional parameters for decimation and upsampling were calculated and alternate cases were created to test modes with similar constraints to the ones assigned to our group. All of these alternate modes were tested, recorded, and compared. We also used another group's mode 3 parameters with our code to see if that would have an effect. Surprisingly, the code worked perfectly. At this stage we thought there may be an issue with our parameters, but we proceeded with the planned tests to verify our theory. The visual in Figure 4.3 shows each of the tested parameter combinations which ultimately helped us identify the issue.

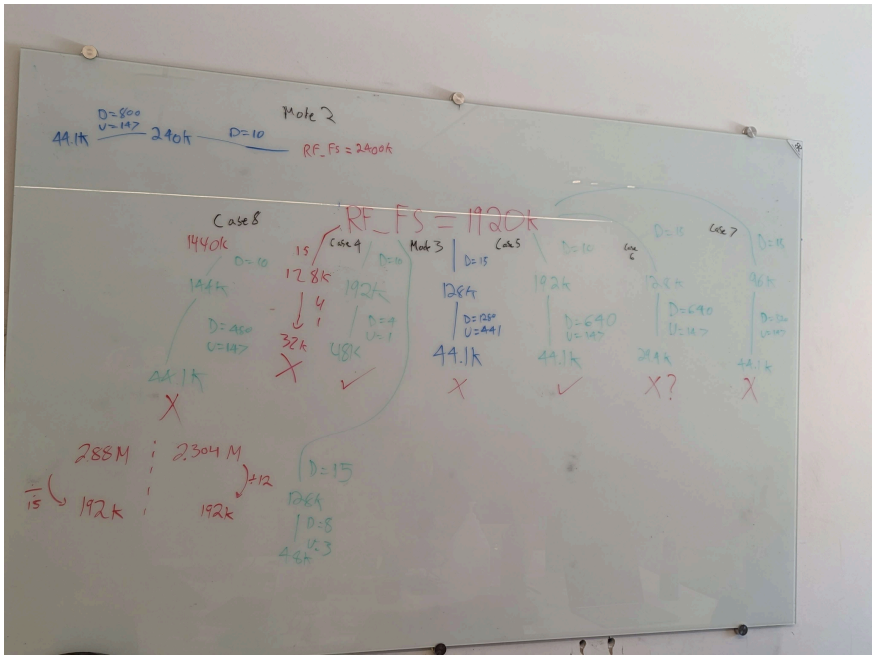


*Figure 4.1: Test Details*

After completing all of these tests, it was easy to identify that a decimation factor larger than 15 in the front-end downsampling would consistently result in a distorted signal. This was further emphasized by the fact that 'Case 4' in the above figure is a variant of mode 0 (with only downsampling).

This confirmed that the issue was not due to our resampler. Therefore, we effectively proved that having a relatively high frontend decimation factor was the main factor that caused some error in our code. However, at this point we still did not know why this was the case and could not directly link it to the parameters we were given. Due to this, we assumed that there must have been a fundamental logic issue in our code. We began debugging our python model of mono and stereo modes rather than continuing with C++. To properly continue debugging this issue, we used the variant of mode 0 in all future tests. This allowed us to test the part of our code that failed (frontend decimation) in isolation.

To properly isolate the issue, we extensively tested the rest of our functions by creating GNU plots at each stage of the audio processing. The data entering and exiting every function for mode 3 was graphed and compared with the working modes. We started at the end of the program, and worked our way up. We confirmed that every signal in the back-end was corrupted in some way. This meant that the issue was indeed in our front end, so we traced it accordingly. We verified that the signal was significantly corrupted after exiting the front-end portion of processing. Since we were now certain that the problem was arising before back-end processing, detailed testing was done with the front-end so that the exact edge cases in which the bug was present could be determined.

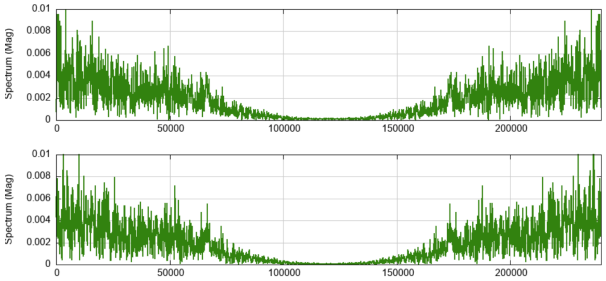


*Figure 4.2: IQ Samples plot after plotting mode 0*

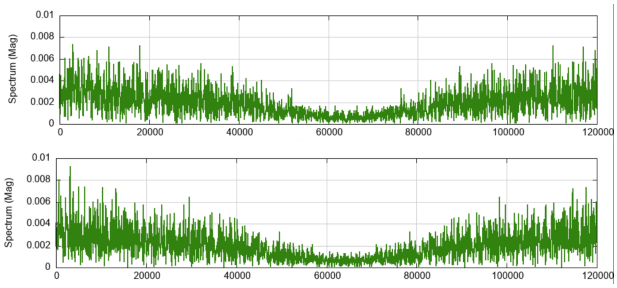


*Figure 4.3: IQ Samples plot after plotting variant of mode 3*

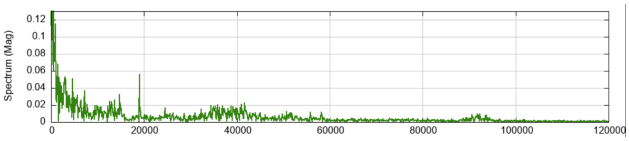


*Figure 4.4: FM demod plot after plotting mode 0*

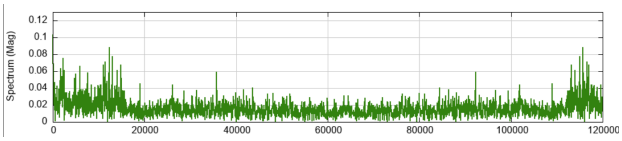


*Figure 4.5: FM demod plot after plotting variant of mode 3*

Upon viewing this plot, we saw that the IQ samples corresponded to what was expected. The signal was heavily corrupted only after the demodulation stage. Although we already identified that the issue was in the front-end we also switched to fmMonoBasic.py to ensure that the issue did not have anything to do with our block processing. The code still failed. Therefore we identified that the issue must be with fmDemod. However, even after using the provided function in python the signal was still corrupted. It was at this point that we once again met with Dr. Nicolici to show him our findings. After this meeting, he tested our parameters with his own code (both in C++ and python). After neither of those worked, our theory that there was an issue with the parameters was confirmed. Our parameters were altered, and we were given a smaller frontend decimation for mode 3.

## 5: Analysis and Measurements

$$(PLL\ In_{samples}) * (2 * cos_{op} + sin_{op} + arctan_{op}) = PLL_{operations}$$

$$PLL_{operations} = 4 * PLL\ In_{samples}$$

*Equation 4.1: Per block computational cost of PLL/NCO*

Looking at the equation above, it is evident that the PLL calls cos twice, and the arctan/sin functions once per for loop iteration. Assuming all the trigonometric functions have a similar time complexity, we can group them together. Therefore, there are 4x the number of PLLIn sample operations per function call, which corresponds to ~1ms cost per block.

| Mode # | Mode Type | Theoretical No. of Multiply+Accums |
|---|---|---|
| 0 | mono/stereo | 1,111 |
| 1 | mono/stereo | 1,717 |
| 2 | mono/stereo | 1,200 |
| 3 | mono/stereo | 1,860 |

*Table 5.1: Analysis of accumulation operations of different modes*

We noted that for each mode, the number of multiply and accumulation operations, in the frontend + mono paths equaled the number of multiply/accum. operations for stereo only.

$$Multiply/AccumOperations = \frac{taps * (IF * 2 + Frequency_{output})}{Frequency_{output}}$$

*Equation 5.1: Formula to find mono/stereo number of accumulates*

Note that the frontend processes I and Q samples separately, and the stereo path processes the pilot and stereo band separately. Furthermore, both the complementary mono and stereo modes have the same output frequencies. Therefore, both the frontend+mono path has the same number of multiply/accums as stereo.

We then did a mode-by-mode comparison detailing the runtime of each function in each mode. These are the results of our findings.

$$Block\ size = c_1 * Decim_{RF} * Decim_{IF} * 2$$

*Equation 5.2: Block sizes were calculated using this equation*

This was to ensure that all the block sizes were divisible by all of the relevant decimation factors pertaining to any particular mode. The constant, c1, was changed to make the output block size approximately between one and five thousand samples. This is the optimal range for Aplay, and was recommended as per the project specs. For RDS in python, we accounted for the RDS decimation factors as well.

The timing analysis of significant runtime functions (1ms+ per block) is outlined below.

| Mode | Function | Run-time (All blocks) |
|---|---|---|
| 0 | Dequeue | 1872 ms |
| | Pilot Convolve | 702 ms |
| | Stereo Convolve | 702 ms |
| | fmPLL | 234 ms |
| | Resample Convolve | <117 ms |
| 1 | Dequeue | 2275 ms |
| | Pilot Convolve | 1050 ms |
| | Stereo Convolve | 1050 ms |
| | fmPLL | 175 ms |
| | Resample Convolve | <88 ms |
| 2 | Dequeue | 1950 ms |
| | Pilot Convolve | 900 ms |
| | Stereo Convolve | 900 ms |
| | fmPLL | 150 ms |
| | Resample Convolve | 150 ms |
| 3 | Dequeue | 3000 ms |
| | Pilot Convolve | 1350 ms |
| | Stereo Convolve | 1350 ms |
| | fmPLL | 300 ms |
| | Resample Convolve | 150 ms |

*Table 5.2: Stereo Runtime*

| Mode | Function | Run-time (All blocks) |
|---|---|---|
| 0 | Dequeue | 1872 ms |
| | Resample Convolve | <117 ms |
| 1 | Dequeue | 2275 ms |
| | Resample Convolve | <88 ms |
| 2 | Dequeue | 1950 ms |
| | Resample Convolve | 150 ms |
| 3 | Dequeue | 3000 ms |
| | Resample Convolve | 150 ms |

*Figure 5.3: Mono Runtime*

| Mode | Function | Run-time (All blocks) |
|---|---|---|
| 0 | All functions | 1872 ms |
| | RF Decimation + Convolve | 1638 ms |
| 1 | All functions | 2275 ms |
| | RF Decimation + Convolve | 2100 ms |
| 2 | All functions | 1950 ms |
| | RF Decimation + Convolve | 1800 ms |
| 3 | All functions | 3000 ms |
| | RF Decimation + Convolve | 2850 ms |

*Figure 5.4: Frontend Runtime*

The runtimes obtained from this analysis are in line with what we expected. The Front-end + Mono path run-time is approximately equal to the stereo path run-time across all modes. This is displayed in the table above.

## 6: Proposal for Improvement

To improve the functionality of our project, the first step would be to optimize the way we implemented threading. We identified some issues with some stereo modes that caused skipping when listening to the radio in real-time. This means that our code is likely processing the signal slower than the

speed at which it is being received. As of now our code implements multithreading, but it creates a new set of threads for the frontend and backend every time a block is processed. As a result of this our de-queueing time is abnormally high, which can be seen in the timing table above. In order to optimize this we would need to create two separate for loops in the main() method, and initialize the threads outside of them. We would then process the blocks from the frontend individually using its own thread, and do the same for the backend. The threads would join only after the for loops have completed and all blocks have been processed. This would allow for the best performance, where the frontend and backend run concurrently, producing a much more optimized solution.

We could also implement RDS in C++ to receive radio broadcasting information in real-time. To do this, we would first need to ensure that our Python model is working as expected. We would need to fix the resynchronization problem to ensure data is always (or almost always) being received.

Then, we would refactor our Python model into C++; this could be done fairly easily by making use of existing functions for filtering and convolution, as well as adding some new functions for things like RDS demodulation, decoding, and frame synchronization. There would likely be some debugging required to get it working as expected. Additionally, we would need to put the RDS path into a new thread so it could run alongside the frontend and audio paths.

Lastly, we could implement a user interface when using the radio. Currently, a relatively complicated command must be typed into the terminal to choose a radio station and a mode. The average person would likely have some difficulty trying to listen to music using our SDR. The user interface would include mode selection, as well as audio type selection (mono or stereo) and whether or not to use RDS. There would also be a way to choose which channel to tune into. Based on the user's selections, the appropriate command would be run behind the scenes, as shown in Figure 5.1. The interface would then display the user's selections while playing the audio, as well as RDS data if desired. It would have a start and stop button to control the reception of signals.

```
rtl_sdr -f <radio channel> -s <frontend sampling rate based on mode> -
    | ./project <mode> <type> | aplay -c <num channels based on type>
    -f S16_LE -r <audio sampling rate based on mode>
```

*Figure 6.1: Command for Running the SDR in Real-Time*

## 7: Project Activity

Each group member's tasks and biggest challenges per week are outlined below.

| | Week 0 | | Week 1 | | Week 2 | | Week 3 | |
|---|---|---|---|---|---|---|---|---|
| | **Tasks** | **Challenges** | **Tasks** | **Challenges** | **Tasks** | **Challenges** | **Tasks** | **Challenges** |
| **Isaac Thomas** | *Reviewed project spec | *Understanding project requirements | *Refactored lab 3 Python code to C++ | *Using the correct data types in C+ and working with vectors | *Debugged mono path (identified and fixed small bugs in RF frontend) | *Understanding what to look for when debugging, creating debugging methods like plotting and audio outputs | *Implemented mono all-pass filter (delay block) *Implemented band-pass filters for stereo *Implemented pointwise multiplication, addition, and subtraction *Created stereo signal flow in C++ | *Ensuring each component worked as expected |
| **Huner Dogra** | *Reviewed project spec | *Understanding project requirements | *Read over sampling slides to brainstorm ideas for implementation *Coded a fast downsampler for front-end and mode 0 & 1 | *Implementing states with downsampling | *Finished coding resampler (identified potential bugs) | *Fixing issue with resampler (state saving not implemented correctly) | *Debugged resampler to identify issue with mono mode 3 (went through the code line by line to ensure that the correct parameters were being passed in) | *Identifying exactly where our code was faulty (necessary for proper operation of radio as a whole (if an earlier stage was wrong it may interfere with stereo or RDS)) |
| **Charlotte Funnekotter** | *Reviewed project spec *Added lab 3 code to project repo | *Understanding project requirements | *Refactored lab 3 Python code to C++ | *Using the correct data types in C++ and working with vectors | *Debugged mono path (identified and fixed small bugs in resampler | *Learning the theory behind resampling to better understand what was expected from the resampling block; it took some time as the concept was new and fairly complex | *Implemented mono all-pass filter (delay block) *Refactored PLL from Python to C++ | *Incorporating state into the PLL, which was essential to facilitate block processing but required a basic understanding of the functionality of the PLL |
| **Sarbjot Ghotra** | *Reviewed project spec | *Understanding project requirements | *Read over sampling slides and coded fast downsampler *Converted downsampler to resampler *Performed calculations for mode 2 & 3 sampling parameters | *How to use upsampling factor U in the conditional statements of the for loops in resampler | *Finished coding resampler | *Fixed for loop iteration within resampler (using upsample factor properly) | *Incorporated the resampler into mono *Mono debugging | *Finding additional ways to debug aside from using print statements (GNU plots, parameter adjustments) |

*Table 7.1: Project Activity, Weeks 0 to 3*

| | Week 4 | | Week 5 | | Week 6 | |
|---|---|---|---|---|---|---|
| | **Tasks** | **Challenges** | **Tasks** | **Challenges** | **Tasks** | **Challenges** |
| **Isaac Thomas** | *Debugged stereo path (plotted waves at every step for stereo debugging, identified PLL discontinuity, created functionality for plotting time and frequency domain; converted frequency bins to frequency in hertz) | *Obtaining a plot that accurately represented the input data (i.e. converting freq bins) | *Debugged mono mode 3 *Added stereo functionality to Python model *Implemented stereo model with PLL, all-pass filter, etc. | *Searching for bugs in mode 3 | *Debugged RDS frontend in python, fixed I and Q components of PLL using constellation plots *Finished the code for the Manchester Encoder, implemented and tested *Added manchester threshold to fix desynchronization issues *Provided support for RDS application layer | *Receiving coherent data was critical in the RDS signal flow process, in order to ensure it could be processed by subsequent steps *Ensuring the constellation plots had a clear separation in I and Q samples, and that most of the power in the In-phase component |
| **Huner Dogra** | *Continued debugging mono mode 3 (used print statements to isolate issue, reverted to slow desampler (identified that issue was prior to resampling)) | *Identifying exactly where our code was faulty (necessary for proper operation of radio as a whole (if an earlier stage was wrong it may interfere with stereo or RDS)) | *Continued debugging mono mode 3 (plotted all stages of audio processing to identify location of issue) *Implemented multi-threading class (created enqueue function using built in queue function, and used the notify_one condition variable) *Created threads in main class to separate front end and back end operations (enqueued and dequeued in appropriate locations) | *Identified and fixed infinite looping error (threading necessary for code to run in real-time, especially once RDS is implemented, infinite looping prevented code from running properly) | *Created Python model for Channel Extraction/Carrier recovery (implemented function to square signal, bandpass filter with new parameters, PLL and NCO with new parameters, and all pass filter to match delay) *Created frame synchronization function for RDS (did matrix calculations to compare with syndrome, implemented 'synced' variable to determine how much to advance sampling window) | *Debugged issue with sampling window never syncing (frame synchronization necessary for application layer to obtain complete data sets) |
| **Charlotte Funnekotter** | *Debugged stereo path (identified and resolved issue with state-saving in PLL) | *Getting a deeper understanding of the functionality of the PLL and what the expected output should look like | *Refactored and cleaned up code to prepare for the implementation of threading and Unix piping (separated frontend and backend, moved loop and mode parameters to main function, added structures for cleaner state saving) *Implemented Unix piping functionality | *Configuring the program to receive input from the terminal; this required a complete reorganization of our code | *Implemented Manchester/differential decoder, pattern finder, and window retrieval *Created frame synchronization function (did matrix calculations to compare with syndrome, implemented 'synced' variable to determine how much to advance sampling window) | *Understanding the flow of RDS processing (cross-referencing the lecture slides, project spec, and online resources) *Debugging issue with sampling window never syncing |
| **Sarbjot Ghotra** | *Continued debugging mono mode 3 (used print statements to isolate issue, reverted to slow desampler (identified that issue was prior to resampling)) | *Continued testing of alternate modes with various parameters/constraints | *Continued debugging mono mode 3 (tested many alternate modes to isolate the issue) *Implemented multi-threading class (created dequeue function using built in queue function and implemented while loop with wait()) *Created threads in main class to separate front end and back end operations (enqueued and dequeued in appropriate locations) | *Identified and fixed infinite looping error | *Created Python model for Channel Extraction/Carrier recovery (implemented function to square signal, bandpass filter with new parameters, PLL and NCO with new parameters, and all pass filter to match delay) *Created Python model for RDS demodulation *Started the code for the Manchester Encoder *Created python model for RDS Application layer | *Determining a correct and effective algorithm to manchester encode *Within RDS Application Layer, determining character position appropriately in the program service name from the decoder control bits |

*Table 7.2: Project Activity, Weeks 4 to 6*

## 8: Conclusion

This project has provided us with a very thorough learning experience. It allowed us to see how our work can be applied in the real-world, and gave us the knowledge necessary to complete a project of this type in the future. We learned the importance of time management as well as how to implement computer engineering concepts in a practical way. We also learned the value of dividing larger tasks into smaller subtasks to assist with completion. By splitting up mono, stereo, multithreading, and RDS, we were able to more easily approach every problem we encountered. Working in a group setting, we determined the usefulness of dividing work based on individual strengths. Overall, we learned about the practical applications of digital filtering, signal processing, and frequency modulation. Throughout this project, we developed skills to effectively identify and solve issues with code, which will be very important for any future engineering endeavors.

## 9: References

Nicola Nicolici, "COE3DY4 Project: Real-time SDR for mono/stereo FM and RDS," McMaster University, Accessed: Feb. 19, 2024. [Online]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/554053/viewContent/4569193/View

“Specification of the radio data system (RDS) for VHF/FM sound broadcasting in the frequency range from 87,5 to 108,0 MHz,” European Standard, April 1998. Accessed: March 26, 2024. [Online]. Available: http://www.interactive-radio-system.com/docs/EN50067_RDS_Standard.pdf