COE3DY4 Project Report
Group 15

Andrew Gurges gurgea1@mcmaster.ca
Ben Miles milesb3@mcmaster.ca
Justin Covach covachj@mcmaster.ca

April 10, 2023

1 Introduction

The main objective of this project was to build a software defined radio (SDR) capable of running live on a Raspberry Pi 4 with an RF dongle based on the Realtek RTL2832U chipset [1]. The SDR was to output mono/stereo audio and RDS information live from the FM band. This meant learning about communications fundamentals, such as demodulators and finite impulse response filters, and implementing these in software to process the incoming FM data. These processes also had to be optimized such that the limited resources on the Raspberry Pi 4 could process incoming FM data fast enough for live output.

2 Project Overview

In this section, the main process of the project will be explained. This will be done by following the data flow and exploring the necessary computational blocks for proper success. First, FM radio waves are intercepted by the RF hardware, which is simply attached to the Raspberry Pi 4 and outside of our responsibility and control. The RF hardware converts the analog radio signal into 8-bit digital samples representing the in-phase (I) and quadrature (Q) components [1]. The FM band occupies frequencies 0 to 100 kHz. When someone broadcasts FM data, it is first modulated into channels of bandwidth 200 kHz following industry standards [1]. This allows people to broadcast FM band's of similar structure simultaneously, since the channels can be extracted and processed separately. The IQ samples output from the RF hardware, which pertain to a certain FM channel, are fed into the FR front-end block that performs the demodulation. At the demodulation, the data is also downsampled. This is common throughout any part of the project which contains a filter The frequency leaving the RF front-end is the intermediate frequency, IF.

After exiting the RF front-end, the data travels through three parallel paths: mono, stereo, and RDS. The mono path starts by extracting the mono signal from the FM band by using a low-pass filter with a cutoff frequency of 16 kHz. This filter is also responsible for converting the sample frequency of the data from the IF to that of the audio sample frequency. There are four distinct modes our SDR can run within which define different IF and audio sample frequencies. The output of this block is mono data, audio which contains both the right and left channel.

The stereo path is very similar to that of mono, except for some additional computations. First, unlike mono, stereo does not sit at the baseband but instead is modulated by a 38 KHz carrier. To begin, the stereo channel is extracted using a bandpass filter. For the purpose of demodulation, a stereo carrier recovery exists at 19 kHz. It is to be extracted by a bandpass filter, then passed through a Phase Locked Loop (PLL) and Numerically Controlled Oscillator (NCO). Next, the stereo channel is mixed with the stereo carrier to demodulate the signal, and the sample frequency is adjusted to match that of mono. Lastly, the stereo channel holds the left channel

minus the right channel audio and so it must be added to mono to retrieve the left channel, and subtracted from the mono audio to retrieve the right channel.

The third path is the RDS path, which holds data about the broadcast that can be displayed on digital screens. Like stereo, it is modulated and must be demodulated. However, unlike stereo there is no dedicated carrier recovery, so instead the extracted channel is squared, bandpass filtered, and sent through the PLL and NCO to create a 57 kHz carrier. Once the carrier and extracted channel are mixed, further processing is done to prepare the output RDS waveform to be decoded. Once this is done, the bitstream is extracted by reading the Manchester encoding, synchronizing to the data frame, and finally converting the bits to characters.

3 Implementation details

RF Front-end

In order to demodulate the in-phase and quadrature components of the incoming signal we will perform RF front-end processing to convert the signal to an Intermediate Frequency (IF) which will be fed into all other signal paths to extract audio and digital data. To perform this we take the in-phase and quadrature components and pass them through a low pass filter with the cutoff frequency of 100kHz. It is then passed through a decimator to reduce the amount of incoming samples to analyze. In modes where the IF sample rate is not a factor of the RF sample rate we will need to perform resampling to upsample the rate first, then downsample to get the desired IF rate. To do this we perform a resampling convolution to optimize the amount of computations that are needed to convolve the upsampled filter then downsample the result. This process takes into account the resulting values after convolution and only computes the results that are kept.

In the design process we first created the process for mode 0 and created a simple convolution to apply the filter, then downsampled the values that resulted for the desired sample rate. Once this was verified we moved on to make the resampling convolution for the modes which would not be able to simply be downsampled.

During this process we found some challenges in creating the algorithm for the resampling convolution. We had difficulty in visualizing the process and implementing it in the most optimized way. Ultimately we were able to design a working algorithm that resampled the signals effectively.

Mono Path

In order to extract the mono audio channel, first a lowpass filter coefficient array is created with the same function used in the RF front-end section. This filter will have a cutoff frequency of 16 kHz to extract just the mono audio channel. These filter coefficients are then convolved with the signal received from the RF front-end block in order to extract the mono audio channel. The first iteration of this implementation in python was done with a full convolution, with downsampling of the resulting signal done after the convolution. Once the fast downsampling convolution function was developed, it was integrated into the convolution on this path. For modes 2 and 3, the downsample factor was not an even number, so resampling was required. After the downsampling convolution was verified, the resampling convolution was

created and compared to the downsampling version as well as the original convolution to test that the timing of the resampling function was faster than that of the normal convolution.

We had issues with the resampling convolution function, as the filter coefficient taps and sample frequency had to be scaled by the upsample factor. We also had trouble verifying the timing of the function, but after analyzing the number of iterations of the convolution and comparing it to the expected size of the output, we could improve on the design of the function.

There were no major issues transferring this section from Python to C++. In order to verify the C++ design, we wrote the output of the C++ process to a binary file, and plotted it with a python script using the same format as the python implementation. This was so the output of the python model and the C++ implementation could be compared more accurately.

Stereo Path

Stereo processing has two main components, the stereo channel and the stereo carrier. The stereo channel extraction is very similar to the mono channel extraction, but it uses a modified version of the previously defined lowpass filter, which accepts a frequency for the beginning and the end of the passband in order to create a band-pass filter. For the stereo channel extraction, the bandpass filter was set to 22 kHz to 54 kHz. The bandpass filter coefficients are then convolved with the output from the RF front-end, to be later mixed with the carrier.

For the carrier recovery, a bandpass filter with frequency range 18.5 kHz to 19.5 kHz was convolved with the RF front-end signal, in order to extract the carrier signal. Then, a phase-locked loop was synchronized to the carrier at 19 kHz with a numerically-controlled oscillator to double the frequency to 38 kHz.

When both the carrier and the stereo channel are extracted, they are then mixed using pointwise multiplication. This mixed signal was filtered using the same filter coefficients as in the mono path, using the resampling convolution function, and then the stereo and mono channels were combined with addition and subtraction to create the left and right audio channels.

In order to verify the correctness of the left and right channels, we used an audio file with distinct left and right channel recordings and listened to the audio being output. At this stage, we learned of the need for synchronizing the stereo and mono paths in order to create a perfect separation of the left and right channels. We implemented this mono channel delay by passing the mono channel through an all-pass filter (filter coefficient array filled with zeros with a single 1). The delay caused by this all-pass filter could be tuned by shifting the location of the 1 in the array. We found that in an array with the length of the number of filter taps for the mono and stereo channels, putting the 1 halfway through this array caused the best delay.

RDS Path

The RDS path consists of 3 main stages, RDS carrier recovery, RDS demodulation, and RDS data processing. Before we get into these stages we pass a bandpass filter on the desired frequency range(54-60kHz) to extract the RDS signal. On this signal we then perform carrier recovery which consists of squaring nonlinearity, a bandpass filter(113.5-114.5kHz) and a PLL with an NCO to recover the carrier wave. The resulting signal is then mixed with the extracted channel from earlier that has been passed through an all-pass filter to match the delay introduced in the carrier recovery portion of the process.

Now the signal is ready for RDS demodulation. For this section we pass the signal through a lowpass filter to keep only the RDS data that we require. We then need to pass it

through a rational resampler that uses the same resampling techniques we applied in the other sections of processing. The only change is the upsampling and downsampling factors we will use to match the samples per symbol rate required for the two modes. We determine the resulting sampling frequency by multiplying our sample per symbol rate by the symbols per second rate. This allows us to change the frequency to the desired rate. We then pass the signal through a root-raise cosine filter which essentially increases the magnitude of the resulting signal. This signal then goes through the process of data recovery to extract the values of each symbol.

With these symbols we would enter data processing but we will only briefly touch on this topic because we were not able to proceed to this point in the path due to some issues we will talk about in the next section. For this block we use the extracted symbols and perform manchester decoding on them. We then perform frame synchronization, error detection and pass it into an application layer and we have successfully extracted the radio data.

For the design process the carrier recovery is very similar to the stereo carrier recovery. The only difference is the squaring nonlinearity which was simple to implement due to it being a pointwise multiplication. The changes in filter cutoffs and PLL frequency was also simple as we designed our system with these parameters in mind.

We then moved into the demodulation section which uses previously used tools such as a low pass filter and a resampler. The new change was the root-raised cosine filter which was implemented for us. After this stage we ran into a few issues.

The first issue we found was the magnitude of our signal after the RRCF. Our magnitude for the signal was extremely low so we began tuning the all pass filter to lower the quadrature phase magnitude and recover our signal. This allowed us to discover that our data had been corrupted somewhere along the processing path as our constellation diagram showed many values near the 0 value. We were ultimately unable to determine the cause of this corruption but believe it could be an issue with the resampling method causing the issue. The magnitude of the signal was also resolved by multiplying the signal by the upsample factor after resampling.

Clock and data recovery was simple to implement as we used an algorithm that checked the minimum value and found the nearest peak to determine the symbol offset. Then it would extract the values. We were unable to verify its validity due to corrupted data.

Multithreading

Multithreading is an important part of this system's processing speed and real-time operation. By implementing multithreading we would be able to optimize the separate paths so that the processing could be done in parallel. Our initial design implemented RF frontend on its own thread pushing blocks it processed into a queue which mono and stereo extracted for use in their computations in a separate thread. RDS also had its own thread running where it would compute its values using the same block. To ensure proper synchronization for this, a mutex lock was used to ensure no intermediate data was being processed before the previous data had been fully processed. Our second implementation attempted to improve the speed of the previous system by also separating the mono and stereo path. The mono path would also use a queue to send its final processed data to stereo for mixing. This process was synchronized using the same queue and mutex combination as in the previous section. Overall we found some issues in debugging the different thread synchronization and were unable to fully implement our final desired outcome to fully optimize our design.

4 Analysis and measurements

Analysis of Linear Computations

Table 1 shows the total multiplication and accumulations needed per audio sample (for mono and stereo) and needed per symbol, since we only finished up to the CDR.

Table 1. Number of linear computations for Stereo with num taps = 101

Signal Flow Block	Mode 0	Mode 1	Mode 2	Mode 3
Mono	1,111	1,111	1,200	1,750
Stereo	1621	1621	1755	2583
RDS	48185	-	115929	-

Analysis of Non-linear Computations

For the signal processing path in the RF frontend and mono path there are no required non-linear operations done on the signal. This allows us to do the analysis of these computations for the Stereo and RDS paths.

As we can see from Table 2 in the appendix, the change in non-linear computations throughout the stereo process does not change through the different modes. Although the interpretation of the LPF and BPF can be interpreted to be preliminary computation to the processing path, it is essential to complete these computations before processing the signals. These processes are completely reliant on the number of taps being used for the processing. The PLL however does not depend on the number of taps and requires 4 non-linear computations for each sample.

We can also see in Table 3 for the RDS computations that a similar result is perceived. For the computations of the LPF and BPF the amounts do not change. The PLL depends on the amount of samples and the RRCF depends on the number of samples per symbol. These are the only 2 factors in the RDS signal processing up to the CDR process.

Table 2. Number of non-linear computations for Stereo with num taps = 101

Signal Flow Block	Mode 0	Mode 1	Mode 2	Mode 3
LPF coeff	201	201	201	201
BPF coeff (x2)	302	302	302	302
PLL	4	4	4	4
Total	507	507	507	507

Table 3. Number of non-linear computations for RDS (Up to CDR) with num taps = 101

Signal Flow Block	Mode 0	Mode 2

LPF coeff (x2)	201	201
BPF coeff(x2)	302	302
PLL	4	4
RRCF	36	82
Total	543	589

Runtime Measurements and Analysis per Function

Runtime measurements of all completed functions can be found in the **Appendix**. From these runtime measurements, we are able to determine which functions are the most computationally heavy, and which may be causing bottlenecks. Table *5* of the **Appendix** shows running times per processed block of data for all functions in all modes. From this data, we see that the most computationally heavy functions are the filters and rational resampler, the functions which require convolution. Also, these are the functions which scale upwards in running time as resampling is needed in modes other than Mode 0. The critical path of this project is thus the resampling convolution.

The impact of the number of filter taps was explored experimentally by varying the number of filter taps from 13 to 301 and listening to the output from Mode 0. Table 6 in the **Appendix** summarizes the running times for this part of the experiment. The great increase in running times from 13 taps to 301 taps can be observed in the table, but it could also be heard. Mode 0 running at 13 taps had no underruns or moments where the audio cut out due to the audio not processing fast enough, however the audio quality was very poor. There was static and it sounded compressed. On the other extreme, Mode 0 with 301 taps sounded very good, but it could not process the audio very quickly so there were constant cut outs where the audio processing had to catch up to the incoming radio data.

5 Proposals for improvement

To improve on the user experience that the project provides, we can add a few features to this SDR. One feature we could add is a GUI that would make it easy for the user to run the program to listen to the radio, as well as an ability to play and pause and being able to change the station. Adding a GUI could be done with python, with code included to run the C++ code with a given radio frequency. Instead of the GUI, we could instead connect push buttons and a potentiometer for playing/stopping the radio and for tuning the radio station. This could be done with the raspberry pi's on-board pins and an ADC converter, which can run the C++ code for this project when given commands by the physical inputs. A feature that could improve the groups ability to verify the work done at each stage could be unit testing in order to verify each individual function to ensure correctness at each stage of development. Another way the program could be improved is with more modularization of the different processing streams of the program. Moving each stream to its own file could make the system as a whole more readable and more easily appended to in the future if need be.

To improve runtime and performance we have to consider a few things. First, we will be unable to optimize the system by processing multiple blocks at a time due to the requirement of

state saving for block processing. Second, due to the nature of signal processing, most processes need the previous stage to be completed before it may proceed with its processing. This gives us a few optimization paths to take. Our main option is to break down the processes into further threads when the signal processing can be done in different streams. For example, in RDS signal processing we apply an all-pass filter on the same data that also runs through carrier recovery. If we run these two processes in separate threads we will find some optimization in the system runtime. This principle can be applied in an area of the signal processing where the separate processes do not rely on the result of the other. Another area for optimization could be the algorithms we use. Optimizing the issues and potential throttling of certain processes could ensure faster runtimes. The one throttle for this system is the convolution process. Finding an optimized way to perform this convolution in every situation would significantly increase the runtime and performance of the system.

6 Project activity

Table 4. Summary of Group Members' Activities Over the Course of the Project

Week	Progress	Contributions **A** - Andrew **B** - Ben **J** - Justin
Feb 20	Midterm Recess, project released, group members took time to read the project document.	**A, J, B** - Read project doc
Feb 27	More reading of project documentation, and review of lab work to prepare for the transition to the project. Organizing team for roles in project	**A, J, B-** read project doc and review lab work
Mar 6	Adds Lab 3 work to project, creates faster downsampling function, begins work on resampling function. Adds stereo channel/carrier extraction. Creates a BPF coefficient generator.	**A -** worked on down/resampling and stereo channel **J -** Started channel recovery and Squaring Nonlinearity for RDS **B** - worked on down/resampling and stereo carrier recovery
Mar 13	Adds PLL to stereo carrier extraction, adds state saving to PLL. Adds mixing for stereo channel and carrier, and mixing of stereo and mono for L/R audio.	**A -** Implements PLL state saving **J -** Implemented Band-pass filter, PLL, NCO and all-pass filter for RDS **B** - Completed stereo channel extraction, worked on mixing mono and stereo channels
Mar 20	Improves upon resampling function and adds delay to mono path for synchronization. Creates a python testing script for C++ output. Converts some filtering functions in python to C++	**A -** Works on resampling and python testing script, as well as conversion to C++ **J -** Worked on RDS demodulation **B** - Conversion of some mono/stereo functions from Python to C++

Mar 27	Adds remaining python functions to C++. Adds RDS, mono, and stereo path to C++. Then debugging, verification, and testing with a Python test file. Then tuning the mono delay filter for L/R channels. Adds mode switching to C++. Code testing with APlay and pipes done after verification. Code is refactored to implement multithreading. Final adjustments optimizations done after multithreading implementation to validate real-time running.	**A** - Conversion from python to C++ (RDS, mono, stereo), debugging, testing, mode switching. **J -** Debugging and Tuning RDS. Multithreading structure implemented. Final debugging as well. **B** - Refactors code to work with piping in raw iq samples and piping out to aplay
Apr 3	Work done on this report to prepare for submission in the following week.	**A, J, B** - Works on report
Apr 10	Submission of the final report.	**A, J, B -** Finalize report

7 Conclusions

Overall, in this project we learned about the process of working through a long term self guided project, as well as gaining understanding on software defined radio systems. We learned about the interplay between modeling and implementation through the use of Python to help gain an understanding about the process being performed, and the use of C++ to optimize the implementation in order to work with the limited resources available to us. This also showed us that many programming languages have their own benefits and downsides, where Python is very easy to prototype and iterate over designs due to its ease of use, but it is not nearly as efficient as C++ can be. We learned more about how optimization is not just at the level of the compiler, and that it is extremely important to do algorithm optimization to speed up computations.

8 References

[1] “COE3DY4 Project: Real-time SDR for mono/stereo FM and RDS,” class notes for COMPENG 3DY4, Department of Electrical and Computer Engineering, McMaster University, Winter, 2023.

[2] “3DY4 Project - Custom Settings for Group 15,” class notes for COMPENG 3DY4, Department of Electrical and Computer Engineering, McMaster University, Winter, 2023.

9 Appendix

Tables 5 and 6 summarize runtime measurements taken on the Raspberry Pi 4.

Table 5. Average Runtime per Block (in ms) of All Modes with num taps = 101

Signal Flow Block	Mode 0	Mode 1	Mode 2	Mode 3
Total IQ Sample Intake Time (for	4538.93	4501.61	4461.81	5559.57

500 Blocks)				
In-Phase Samples Filter	2.77509	2.69241	2.75975	4.28084
Quadrature Samples Filter	2.7894	2.70557	2.75312	4.27362
RF Frontend Demodulation	0.022766	0.0218671	0.0246038	0.0354309
Mono LPF	0.495599	0.494677	1.73267	1.93944
Mono Delay Filter	0.0970506	0.0973163	0.0805688	0.0540015
Stereo Channel Extraction BPF	2.65069	2.67712	2.65283	4.25859
Stereo Carrier Extraction BPF	2.67836	2.67791	2.66965	4.28236
Stereo Carrier PLL	0.450001	0.449197	0.483465	0.718144
Stereo Carrier & Channel Mixing	0.501879	0.503036	1.68224	1.95073
Stereo & Mono Mixing	0.00490428	0.00395627	0.0040878	0.00409344
RDS Channel Extraction BPF	2.67988		2.66766	
RDS Squaring Nonlinearity	0.0104591		0.010368	
RDS Carrier BPF	2.66824		2.67075	
RDS Carrier PLL	0.435825		0.472161	
RDS Channel Delay Filter	0.480284		0.47373	
RDS Carrier & Channel Mixing	0.0117747		0.011838	
Demodulated RDS LPF	2.66736		2.67366	
Rational Resampler	1.85555		3.7114	
RRC Filter	0.0866065		0.187664	
CDR	0.00433076		0.0047092	

Table 6. Average Runtime per Block (in ms) of Mode 0 with num taps = 13 and 301

	Number of taps	
Signal Flow Block	13	301
Total IQ Sample Intake Time (500 Blocks)	4259.59	9193.66

In-Phase Samples Filter	0.300101	8.20354
Quadrature Samples Filter	0.308065	8.06188
RF Frontend Demodulation	0.0230332	0.0234108
Mono LPF	0.0728101	1.52376
Mono Delay Filter	0.0105678	0.338062
Stereo Channel Extraction BPF	0.291411	8.14666
Stereo Carrier Extraction BPF	0.295056	8.29664
Stereo Carrier PLL	0.504835	0.472569
Stereo Carrier & Channel Mixing	0.519847	1.50709
Stereo & Mono Mixing	0.00423579	0.00527754
RDS Channel Extraction BPF	0.289093	8.02545
RDS Squaring Nonlinearity	0.00998295	0.0110964
RDS Carrier BPF	0.298899	7.9779
RDS Carrier PLL	0.473082	0.445907
RDS Channel Delay Filter	0.472788	0.471563
RDS Carrier & Channel Mixing	0.0115954	0.0123318
Demodulated RDS LPF	0.287033	8.03803
Rational Resampler	0.25304	2.24393
RRC Filter	0.0849338	0.087149
CDR	0.00297933	0.00473632