COE3DY4 Project Report

Group 21
Mazen Maamon, Samer Rafidi, Tamer Rafidi, Guangyi Wang
maamonm@mcmaster.ca, rafidis@mcmaster.ca,
rafidit@mcmaster.ca, wangg75@mcmaster.ca
2023-04-10

Introduction

In this project as a group, we are working on executing a Software Defined Radio (SDR) system. This system is for real time reception of FM (Frequency Modulated) mono/stereo audio. Also, digital data utilizing the RDS (radio data system) protocol from the RF hardware. Using the Raspberry Pi 4 with RF dongle we can implement this project and collect samples from the radio through the SDR command line.

Project Overview

Frequency modulation (FM) is a method of encoding information onto a carrier wave by varying the frequency of the wave in accordance with the modulating signal. Every FM channel occupies 200 kHz of the FM band which has symmetry in the center frequency; the range in Canada is between 88.1Mhz and 107.9MHz. A frequency demodulator is used to extract the original modulating signal from a carrier wave that has been frequency modulated. It converts the frequency variations of the carrier wave back into their original amplitude variations, allowing the modulating signal to be retrieved. In our case it is a computer program in a SDR but in different implementations it can be an electronic circuit. In this project we concentrate specifically on the FM demodulated channel from 0 – 100kHz. The three sub channels which were the focus of the project were mono, stereo, and RDS. Mono sub channels ranged between 0-15KHz, the stereo pilot at 19 kHz, the stereo 23-53kHz and RDS 54-60kHz.

Software-defined radio (SDR) refers to a radio communication system where traditional hardware components, such as mixers and amplifiers, are replaced by software, computers, and embedded systems. This helps with the flexibility and reconfigurability that is in the radio system. Additionally, being able to implement a range of communication protocols and standards on the same platform.

A finite impulse response (FIR) filter is a type of digital filter that uses a finite and non-recursive impulse response to filter a signal. However, an ideal filter can suppose infinite response but filters which are done by convolution are limited when it comes to size. FIR filters have linear phase response which can be helpful and utilized in audio processing and digital signal processing.

A PLL (phase control system) is a control system which is utilized for generating an output signal whose phase is in relation with the input signals phase. Which is an important aspect when performing demodulation. Additionally, the PLL is utilized for producing a clean output which is done by filtering or amplifying if needed.

Resampler is used for combining the execution of upsampling and downsampling which is needed in conversion between sampling rates. This is needed between conversions of RF, IF

and output audio. Using a resample is more efficient and better to use rather than doing it individually.

Radio Data System (RDS) is a communication protocol which is utilized for embedding small amounts of digital information in FM radio broadcasts. This can be used to display information like program identification, station name, song title, traffic information, and more.

Implementation Details

Labs

The purpose of these labs was to implement some functions and understand why they are used. This is to help us implement them into our final project. In lab 1, we first created a function for Fourier transforms in python and tested it to ensure it works. We also implemented our own low-pass impulse response and compared it with the built-in methods in firwin. We also implemented audio filtering using block processing and replacing firwin. Implementing the functions weren't too difficult as we were given pseudo code. The struggle came from block processing. We had some wrong outputs, but we solved them by plotting data and seeing where the errors came from. In lab 2, we converted all the functions from lab 1 and wrote them in C++. It was a straightforward task but having to brush our knowledge in C++ made it a bit difficult at the start. Lastly in lab 3, we worked on processing in blocks or single pass. We also learned to use PSD plots to check if our implementations worked. We read I and Q samples which were 8 bits and alternating we then normalized it -1 and 1 to 32-bit float format. Then the FM channel was extracted by using a low pass filter (LPF) then downsampled, then applied FM demodulation. We then extracted mono data by using another LPF.

RF front-end

The process initially starts by the RF hardware collecting RF signal by using the antenna which is translated to the digital domain providing in sample (I) and quadrature (Q) 8-bit sample. The stream of sample alternates so one I sample then Q sample etc. This input is split into I and Q data which is done through a function in the code which lets us split the data. In this split function it allows us to store even and odd indices data separately. The RF-front end block in the SDR system then extracts the FM channel by using a low pass filter with a 100k cut off frequency. Following that it goes through decimation (downsample) then it gets demodulated to which we get the mono, stereo, and RDS sub channels. This demodulated function takes in both I and Q data as it's input and the combines them to get demodulated with a sample rate IF based on the mode we are operating in.

Mono Processing

The process for mono processing starts by recognizing what the sample rate of the audio being given is, then deciding the best measure to sample it down to 48 Ksamples/sec. For example, in mode 0, the data did not need to be resampled and was able to be adjusted by the low pass filter. However, for mode 1, it had to be up-sampled by a factor of 24 before passing it through the low-pass filter. The factors were deciding by finding the greatest common divisor between the input and output sample rates. Since the goal was to extract the mono channel, a low

pass filter at cut-off frequency 16 KHz was used. While it was recommended to use 2 separate convolution functions to perform either the down sampling or up sampling, we decided to implement the resample function that is used later in stereo processing for the mono processing stage. Once all stages were implemented, we had issues regarding the output of the audio. After many tests, we recognized that the issue was due to the block size. To fix that, we used different break points in the code to print out the block size and made necessary changes to the code to ensure the code was using the correct block size whenever needed.

Stereo Processing

The process for stereo processing is like the implementation of mono, with a few extra components that have been added to ensure the data is being processed correctly with no noise. In the project guideline, the process for stereo can be seen to be split up into 3 components. Those three include 'Stereo Carrier Recover', 'Stereo Channel Extraction' and 'Stereo Processing.' Furthermore, the use of a phase-locked loop (PLL) and a band-pass filter (BPF) is introduced in stereo processing [1].

In the stereo carrier recovery phase, a bandpass filter is used from a starting frequency of 18.5 kHz to 19.5 kHz. This aims to extract the 19 kHz pilot tone, which is then sent through the PLL to synchronize it to produce a recovered stereo carrier that can be used for mixing. This is also done by multiplying the pilot tone frequency by a factor of 2, which is indicated by the numerically controlled oscillator (NCO) that is in the PLL.

In the stereo channel extraction phase, a band-pass filter is used to extract the left and right stereo channel, which are within the range of 22 KHz to 54 KHz. These two channels will then be mixed in the future component with the pilot tone.

In the final component 'Stereo Processing', the stereo channel is mixed with the recovered stereo carrier obtained in the first stage. This is completed by a pointwise multiplication via a simple for loop. Due to the mixing, another resample must be implemented afterwards to ensure that the data is still 48 Ksamples/sec. Finally, before mixing the mono audio with the stereo audio, we need to ensure that a delay is placed onto the mono-audio so that it mixes how we expected. We implemented the delay by copying the array to a new array by a for loop. Another way to do this is by using an all-pass filter, however that implementation is more resource intensive as it requires multiplication, while the array-copy method is more efficient as its only performing assignment operations. The data is now ready to be combined to create the left and right audio channels that are fed.

The main issue we had with stereo was that there was some error with the block size within the code, which caused mono and stereo to not output proper audio after stereo was implemented. To fix this issue, we placed multiple breakpoints in the code to analyze where the block size gets adjusted and to what value it gets changed to. After multiple iterations, we successfully fixed all issues with the block size, which allowed the mono and stereo audio to play out how we expected it to.

RDS processing

The process initially starts by receiving the IF sample rate. We use a band-pass filter to filter out most of the frequency samples that we do not need. However, there might be some more frequency samples that made it through which are not important, so we use a low-pass filter to process that data and remove the remaining samples that are not needed. This data is then processed to the RDS carrier recovery.

The first step in the RDS carrier recovery is to square the input linearly. This is done because modulating the RDS signal initially shifts the RDS data which results in a frequency deviation. We then square the signal to achieve accurate carrier frequency recovery. However, we lose the phase information when we do so. We then use a band-pass filter to remove any other frequency samples that are not needed. Once we added the band-pass filter, a delay was created in the time it takes to sample the data. To make sure the phases are matched up, we use a PLL with a ncoscale of 0.5 which helps match the phase as we need a shift of 0 or 180 and reduce the frequency by half to return it back to its original carrier frequency. For mixing to occur, we need to ensure that both signals have the same phase which is why we put our RDS channel data through an all-pass filter. This ensures that both the recovered RDS carrier and the RDS channel have the same phase which allows for mixing.

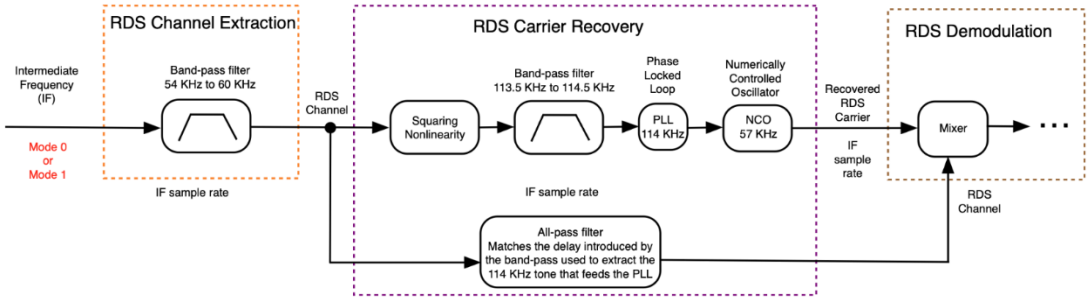


Figure 1. RDS channel extraction and carrier recovery

When we mix the signals, the first step of RDS demodulation is to put our signal through a low-pass filter at 3 kHz. The reasoning for this is to remove any unwanted high-frequency signals which is needed to have an accurate RDS demodulation. The choice of 3 kHz comes because it provides with a sufficient bandwidth to pass the carrier frequency as that is centered at 57 KHz.

We then get to the rational resampler which expects us to take the IF sample rate and change it to a sample rate that is a multiple of 2375 symbols/sec. Our values were 11 and 46 for modes 0 and 2 respectively. This meant that for mode 0, our recommended resamples output would be $11 \times 2375 = 26.125$ KSamples/sec and $46 \times 2375 = 109.25$ KSamples/sec. We used std::__gcd to ensure that our resampler is performing at an efficient and accurate rate.

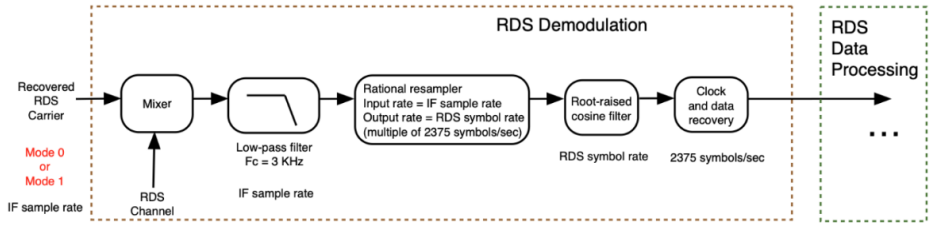
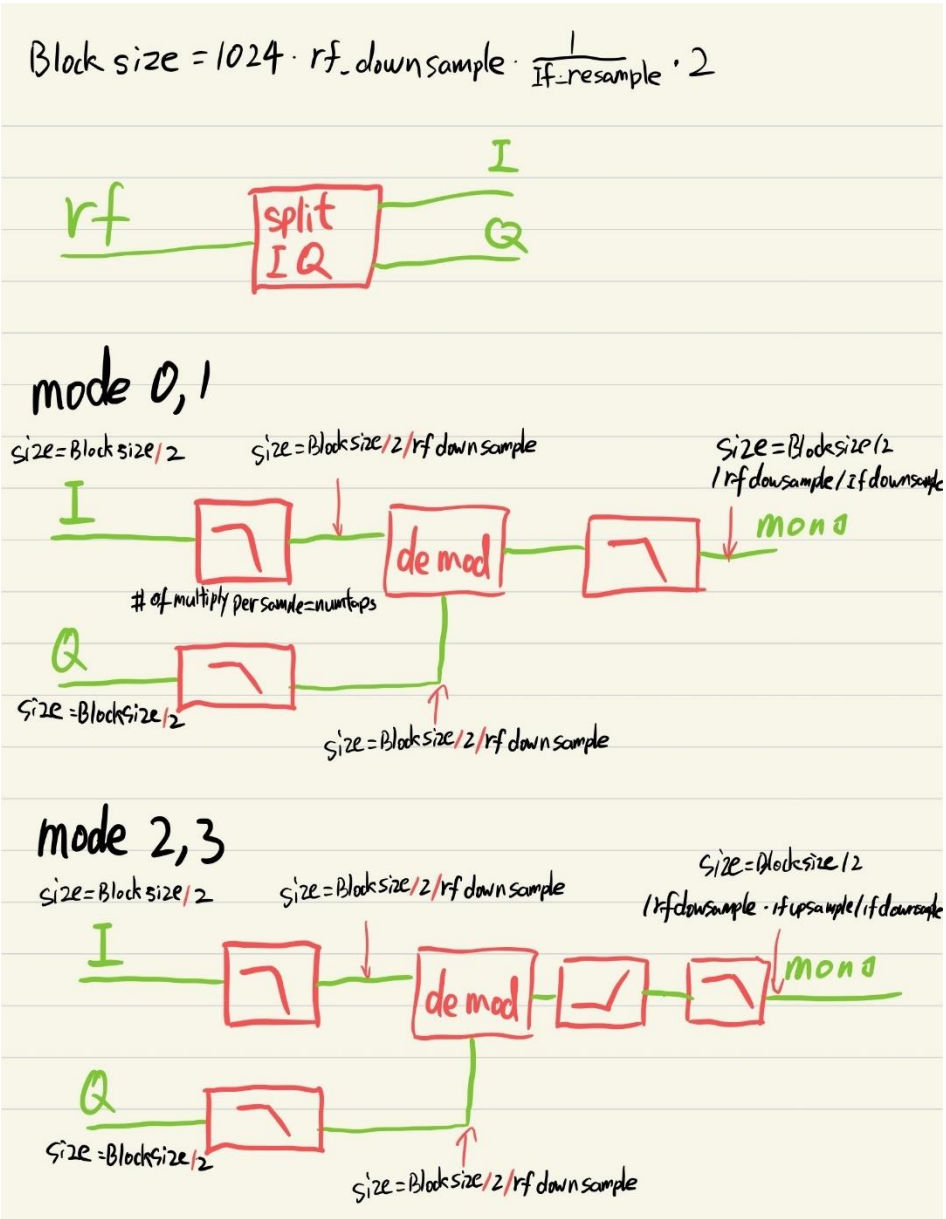


Figure 2. RDS Demodulation

Analysis and Measurements

Mono



Formula used for # of multiplications needed per output sample

Mode	RF_Down	IF_Up	IF_Down	Block size	OutputSize	Multiplication per sample
0	10	1	5	51200	1024	=((2*F3/C3)+G3)*101/G3

Justification for the formula

$$\#\ Multiplication\ per\ sample = \frac{2 \times \#\ of\ samples\ at\ If\ input \times 101 + \#\ of\ samples\ at\ output \times 101}{\#\ of\ samples\ at\ output}$$

The number of multiplications before If input needs to be timed by 2 because filters are used for both I and Q stream.

# of multiplications needed per output sample for all mono modes

Mode	RF_Down	IF_Up	IF_Down	Block size	OutputSize	Multiplication per sample
0	10	1	5	51200	1024	1111
1	10	1	6	61440	1024	1313
2	10	147	800	55728	1024	1200
3	9	441	2560	53499	1024	1274

Mode	Function Runtime(ms) (Output Size = 1024)				
	impluseResponseLPF	resample(RF)	fm_demod	resample(IF)	Per output sample
0	0.0524	7.5739	0.2278	1.4285	0.009065039
1	0.0524	7.4052	0.3695	1.3923	0.00900332
2	0.0524	8.7399	0.3201	2.2257	0.011072363
3	0.0524	10.0842	0.6114	3.1175	0.013540527

Looking at the number of multiplications needed and run time per sample, we can see that the total time is directly proportional to the number of multiplications needed.

Stereo

Stereo being a continuation of mono, simply adds more steps directly on top of the mono path. Since no resampling is needed when extracting the stereo channel, the number of multiplications for the convolution would simply be the number of taps or 101. Fmpll() performs one single multiplication to shift the stereo path on top of the mono path, therefore, adding a total of 102 more multiplications to each sample. The runtime for the resample function for extracting the stereo audio channel and fmpll takes 6.4267ms, 6.4183, 9.8521 and 10.2853 milliseconds for mode 0, 1,2 and 3, respectively. The number of multiplications for the 4 modes would be 1213,

1415, 1302 and 1376, respectively. The band pass filter used to generate the coefficients is performed outside the for loop, but it would take 101 multiplications.

Investigating the effects of num taps on the audio quality showed that using a value of 13 in mode 0 resulted in a bad quality audio which is what was expected as it decreases the resolution of the filter. While increasing the num taps to 301 returned a segmentation fault for reasons that we couldn't figure out. But the resulting audio should be higher quality although might not be receivable by the human eat when compared to a num tap value of 101 or 151. This will come at the cost of increased run time.

Proposal for Improvements

A feature that we could improve to benefit the users of our system is to work on improving the user interface. If we were able to complete the RDS, one way to better improve the user experience would be to have more functionality to our interface. We can add more information for the user that can be helpful. Some features could be showing how strong the signal is to the radio, add any extra information that will help the users easily access what they want about a certain song. Another feature we could have added to improve the productivity would have been automated test can help us in the debugging process a lot and make it much simpler. Since it would find errors in the early stages of development. For continuous integration, it makes sure that the code can be integrated and has been tested. This will make sure that no new bugs are added to the code which would make the debugging process more challenging. Both automated testing and continuous integration is feasible for use.

To improve running time, we could add multi-threading. This helps us run many different processes simultaneously which will make our program run faster which can be done by creating threads which can execute different sections of code. There were times in stereo and RDS when there can be two things ran at once. For example, the all-pass filter and the RDS carrier recovery can be ran simultaneously. This helps improve overall performance of our program by utilizing available processing power. Since multi-threading was part of the project, another change we could've made to improve our running time is to avoid unnecessary memory allocation. We had many vectors which we could've avoided. For example, we created extra vectors to stay organized; these variables could have been removed at the end of the project to improve our running time and taking less space in our memory. One example is the block_data vector which held the value of block_size. Instead, we could have called block_size directly into the functions rather than initializing a new variable for it.

Project Activity

	Samer Rafidi	Tamer Rafidi	Mazen Maamon	Guangyi Wang
Week 1 (Feb 14)	Cross examination + collected device (Raspberry pi)	Lab cross examination	Lab cross examination	Lab cross examination

Week 2 (Feb 21)	Reading week, reviewed lab 3 and briefly skimmed project module	Brief review on past three labs	Briefly read through project module	Went through project module
Week 3 (Feb 28)	Watched the SDR project video and tested lab 3 code with pip	Implemented lab 3 with pip, rather than sending in the data beforehand	Went over the project document and tested lab 3 with project	Tested the contents of lab 3 using real world samples collected using the rtl_sdr command
Week 4 (March 7)	Read through project module and implemented some lab functions into our project	Read through project module and dive deeper into the lab work to understand how things needed to be modified	Started trying to implement the collection of data and turning into FM channel	Read project manual on the implementation of mono and consulted TAs
Week 5 (March 14)	Read through mono module and helped implement some functions in filter.cpp.	Assisted in understanding what mono-processing was asking from us, to begin starting the project	Worked on Mono processing	Finished the first implementation of mono path and stated debugging
Week 6 (March 21)	Read about PLL and band-pass filter and implemented them. Read and understood how we do stereo channel extraction and pilot recovery	Read through the stereo module to understand all components that were involved. Finished recovery of stereo pilot	Debugged segmentation error in Mono processing	Consulted TA for poor mono audio and made sure all vectors are of the correct size
Week 7 (March 28)	Read through RDS and implemented RDS channel extraction /	Completed remaining sections in stereo processing. Assisted in bug	Implemented stereo and debugged block processing issue.	Finished the remaining stereo path and debugged stereo. Added mono

	carrier recovery and started on RDS demodulation	fixes needed. Began to look at RDS processing and translated any necessary python code to C++	Tested live sound play.	delay to remove cross contamination
Week 8 (April 10)	Worked on final report	Worked on final report	Worked on the final report	Worked on the analysis and measurements part on the final report

Conclusion

In this project we used the different things we learned throughout the course and implemented it in a real-life project. As a group we gained a lot of knowledge when it comes to SDR and FM. We gained a lot of knowledge in C++ and python when coding the project and throughout the debugging process which was one of the main processes in this project. We gained more experience in collaborating on a large project like this and how to use git to our advantage. Additionally, we improved our time management skills since we would need to set out self deadlines to follow so when the project deadline comes, we are in a good place in the project to submit.

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 21” Bachelor of Engineering, McMaster University [Online].

[3] 3DY4 Lecture, Documentation, “COMP ENG 3DY4: Computer Systems Integration Project. Stereo Mode, PLL, and Multi-threading” Bachelor of Engineering, McMaster University [Online].

[4] 3DY4 Lecture, Documentation, “COE 3DY4 Lecture 19-20 – RDS Receiver Overview, Demodulation and Decoding”, Bachelor of Engineering, McMaster University [Online].