COE3DY4 Project Report
Group 26

Zhuochen Wu, Hengbo Haung, Rutvi Patel, Priya Patel
wuz147@mcmaster.ca, huanh3@mcmaster.ca, pater52@mcmaster.ca, patep70@mcmaster.ca
April 8, 2022

1 Introduction

The purpose of this project is to implement a software-defined radio (SDR) system in real-time with several different paths which include: mono, stereo, and RDS. The input signal is processed using front-end radio-frequency (RF) hardware, like RF dongles based on the Realtek RTL2832U chipset. For each path, there are four different modes that have different values of RF_Fs, IF_Fs, and Audio_Fs. The overall objective of the project is to consolidate and build on the knowledge acquired in previous courses.

2 Project Overview

2.1 Software Defined Radios (SDRs)

Software Defined Radio (SDR) system is a radio communication system that is used for real-time reception of frequency modulated (FM) mono/stereo audio and the reception of digital data from RDS. The data that feeds into the SDR system is driven by an FM station and the FM channel is extracted by the RF front-end block. The SDR consists of the RF front-end block which produces the input for either the mono path, stereo path, or RDS path.

2.2 Frequency Modulation (FM)

Frequency modulation (FM) is a process of encoding information of a signal by changing the frequency of the input signal and occupying 200 kHz of the FM band for each FM channel. For positive frequencies (0 to 100 kHz), there are three sub-channels: mono (0 to 15 kHz), stereo (23 to 53 kHz), and RDS (54 to 60 kHz).

2.3 Sub-channels: Mono, Stereo, and RDS

The Mono processing path involves a low-pass filter, downsampling, and resampling. On the other hand, the Stereo processing path consists of Stereo Channel Extraction, Carrier Recovery, and Stereo Processing. Although the input and output format is the same for both paths, the mono channel is the sum of the left and right audio channels, while the stereo channel is the difference between the left and right audio channels. Furthermore, the Radio Data System (RDS) extracts the RDS channel (54 to 60 kHz) with signal processing to obtain the digital data. The input to the RDS path is the same as the mono and stereo paths; however, the output is given in bits and words of radio data. The RDS processing path involves RDS Channel Extraction, Carrier Recovery, Demodulation, and Data Processing.

2.4 Finite Impulse Response (FIR) Filters

Finite Impulse Response (FIR) Filters are the building blocks of signal processing specifically, low pass filters and bandpass filters used for the project to guarantee linear phase. Filters can be implemented using convolution for given sample rate, num taps, and cutoff frequency. The FM audio data is processed in blocks to avoid extensive latency for data acquisition and large memory usage.

2.5 FM Demodulators and Phase-Locked Loops (PLLs)

A demodulator is used to recover the data from a modulated signal. After the input FM audio data is passed through a low pass filter and a downsampler, demodulation is performed at the intermediate frequency (IF) signal which is then used as an input to the mono, stereo, and RDS path. Phase-Locked Loops (PLLs) are phase tracking devices used to produce a clean output by filtering or amplifying a noisy input signal.

2.6 Re-samplers and Down-samplers

The purpose of re-samplers and down-samplers is to achieve the desired output sample rate. This can be done by adding zeros between two input samples or removing a certain number of samples from a sequence. For modes 2 and 3, an upsampler followed by a downsampler is needed to implement a fractional resampler to downsample the IF data to the audio data.

3 Implementation Details

3.1 Labs

To begin the project, previous knowledge and functions determined in the labs were used. In lab 1, the impulse response low-pass filter was implemented in Python using the given pseudocode. To verify the function was written correctly, the function was compared with the built-in method from Scipy (Firwin) by inspecting the frequency response of both methods. Furthermore, a convolution function was written in Python to replace the functionality of the built-in lfilter function. The convolution was computed using the input data and the impulse response coefficient with block processing. One of the challenges encountered while implementing convolution was separating the process for state saving of left and right channels which was resolved by using two separate variables for the previous states. To ensure that this was working correctly the gdb debugger was used to check if the output for the convolution matched the expected values.

In lab 2, the code from lab 1 was refactored into C++ for the impulse response coefficient and convolution. One of the difficulties encountered while working with C++ for the first time was understanding how to implement vectors correctly and ensuring that the correct data types are defined. The code from lab 2 was used as a building block for the functions used in the project.

Lab 3 provided the initial understanding for using signal-flow graphs to model a real-life radio application and processing RF data to produce mono audio. The code from lab 3 was used as a model for RF front-end processing and mono mode 0 as shown in the figure below. The input audio is split into I and Q samples that are passed to a front-end low-pass filter using the impulse response coefficient, a sampling rate of 2.4 MSamples/sec, and a cutoff frequency of 100 kHz. Further, for the number of taps, it was determined that 151 was an ideal value that produced accurate results. After

filtering, the data for each of the processing channels are downsampled by a scale factor of 10 to produce a signal at 240 KSamples/sec. This data is then used as an input to the FM demodulator to generate the demodulated signal. In lab 3, the fmDemodArctan function from fmSupportLib.py was replaced with a more computationally friendly method. The method implemented used the equation provided, in which there was a substantial difference between the runtimes. After completing the RF front-end process, the intermediate frequency (IF) signal is obtained in python. The FM demodulated data at the IF frequency is passed to three channels: mono, stereo, and RDS. For lab 3, it was observed that for mono processing, the IF frequency signal is passed through a low-pass filter followed by downsampling to obtain the mono audio for mode 0.

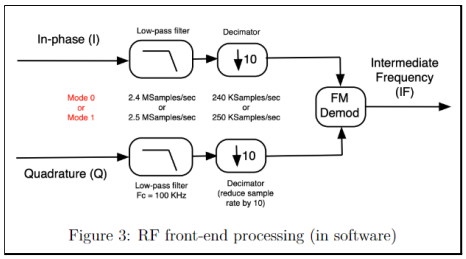
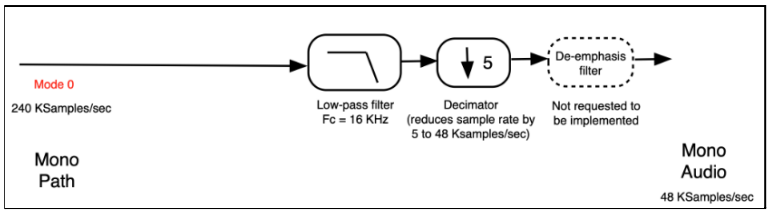


Figure 3: RF front-end processing (in software)



3.2 Mono Processing Path: Modes 0 and 1

To begin, the code used in Lab 3 was refactored into C++ for both RF front-end processing and Mono mode 0. To ensure the code is refactored correctly in C++, estimatePSD function was used to plot and compare the results obtained from python. Additionally, the audio quality was also confirmed with the python model. Then block processing was implemented; however, the I and Q data vectors were initialized to the size of the block instead of half the size. This resulted in problems related to block processing which was solved by tracing back in the code and seeing where the error first arose.

Additionally, to optimize the code written, the RF front-end processing described in Lab 3 was combined into one function called ‘convolutionBlockIQ’. This function was implemented such that it filters the I and Q data simultaneously and downsampled by a scale factor of 10. Furthermore, a similar approach was used for the mono processing of modes 0 and 1 to combine the filtering and downsampling into a single function called ‘convolutionBlock’. Both the functions were implemented such that the filtering only computes the samples that are kept after decimation to avoid unnecessary calculations being computed for convolution achieving optimization.

The parameters given for mono mode 0 include the front-end input sample rate (RF Fs) of 2400 Ksamples/sec, the intermediate frequency sample rate (IF Fs) of 240 Ksamples/sec, and the audio output sample rate (Audio Fs) of 48 Ksamples/sec. These settings are then used to calculate the other parameters. The audio decimation factor is the ratio between IF Fs and Audio Fs, which is given as 240/48 = 5. The block size was determined to be 102400 by using the following equation: 1024*2*rf_decim*audio_decim. Similarly, for mode 1 the decimation factor is 6 and a block size of 98304 is determined using the given custom settings. These settings are then used with the process explained above to get the output for mono modes 0 and 1.

3.3 Mono Processing Path: Modes 2 and 3

Mode 2,3 and Mode 0,1 differ in that IF in mode 2,3 need to do upsampling. For the first version of our code, we got the signal after fm demodulation as input and then upsampled it by the value of expander which is 147 for mode 2 and 441 for mode 3, since in mode 3 we couldn't find a least common divisor. After the signal was upsampled and padded zeros, we used the convolutionBlock function used in mode 0 and 1, which contains the function of convolution and downsampling. For mode 2, first the input 240 kHz frequency was upsampling by 147 to get 35280 kHz, then followed by an LPF and downsampling it by 800 to get output Audio Fs 44.1 kHz. For mode 3, it was similar to mode 2, by changing the value of the expander to 441 and decimator to 3200, the input IF Fs was 141120 kHz (320000 *441).

When we tested the code we found that since it calculated plenty of multiplication, it took time to complete, underruns happened and it was hard to meet the real-time requirement. So we optimized it to a fast version. We merged the up and down sampling as resampling. To reduce the multiplication, we only found indexes that need to be computed, which means we only need to find the index of non-zero in the padded array. So there was a problem finding the corresponding index of y, x, and h. We used two for loops for the convolution. The first one with variable i and increased step value is the upsampling factor. The second with variable j and increased step value is 1.The value of i and j in the loop was the index after the resampling. Through these two for loop, we could use decim * y index - h index to find the index of x before resampling. The index of is j and index of y is i. After implementation, we didn't need to add zero to array and reduced execution time.

In addition, we also found that since the signal needs to expand, the volume of block size needed to increase as well, and as a result, the output quality decreased as the number of taps was small. So we increased the number of taps to 151*147 for mode 2, and to 200*441 to mode 3 to reduce the noise and increase the volume.

3.4 Stereo Processing Path

The stereo signal has three main components: Stereo Channel Extraction, Carrier Recovery, and Stereo Processing as shown in the image below. For the Stereo Carrier Recovery, the Band-pass filter, Phase Locked Loop, and Numerically Controlled Oscillator were implemented. Initially, for the band-pass filter, the pseudocode provided in the project was used to write the function 'impulseResponseBPF'. The number of filter taps was determined to be 151 such that the two band-pass filters have the same delay. The filter parameters f_b, f_e, and f_s were used as defined in the image below. Then the PLL/NCO code provided in Python was refactored into C++. The band-pass filter is used to extract the 19 kHz pilot tone which is then synchronized using PLL to get a clean output and multiplied by a ncoScale factor of 2. For the Stereo Channel Extraction, a band-pass filter was used to extract the stereo sub-channel from 22kHz to 54kHz. The recovered stereo carrier and channel extraction output are then passed for mixing.

Furthermore, for Stereo Processing there are three processes: mixer, digital filtering, and stereo combiner. The mixer and digital filtering are performed through the multiplication of the inputs followed by downconversion in a low-pass filter using the function 'convolutionBlock' for optimization. The stereo data at the audio sample rate is combined with the mono audio by adding to produce the left audio channel and subtracting to produce the right audio channel. Next, block

processing with state saving was implemented in the PLL function so that optimization can be achieved. A structure was used to create a class that saved all the states for the internal operations like the integrator, phaseEst, feedbackI, feedbackQ, ncoLast, and trigOffset. After performing the PLL process the internal operation parameters were saved inside the class. Once the code for the stereo model in python was completed, it was then refactored into C++ while directly implementing block processing.

A challenge that arose during the refactoring of the stereo process in C++ was a segmentation fault which through debugging processes was solved. To resolve the issue, the python model was first compared with the refactored code where it was noticed that during the initialization of the function, the block size was not set up properly. Due to this, when initializing vector size in C++, some of the vectors appeared to be smaller than the size of incoming data, also causing the number of samples during resampling progress to have non-integer numbers that triggered logical errors in the later convolution progress. Also while debugging the optimized version of convolution with resampling, the data generated by C++ code does not correspond with the data generated by the python model. Therefore, GDB is deployed, using breakpoint to run each loop of the optimized function to check how the data changed after each iteration.

In addition, threading was implemented to resolve the performance issue when running the program in real-time. To achieve this, we deploy a producer-consumer design in our program. When the front-end producer thread pulls data from stdin, it processes the given raw data into demodulated data, and the demodulated data is loaded onto the data queue. The data is then being pulled by the stereo consumer thread for further processing. To prevent overload and underload situations, we use a mutex to control the running status of each thread. For underload situations, if there is no data currently on the queue, we use a mutex to lock the stereo consumer thread to wait for the producer thread until data is loaded onto the queue; for overload situations, we check the current number of data blocks on the queue. If we currently have 5 data blocks on the queue, use mutex to lock the producer thread until the number of data blocks on the queue is less than 5. Our design can support 2 separate processes running at the same time but may become more complicated if we implement RDS signal processing and frame decoding into the thread structure.

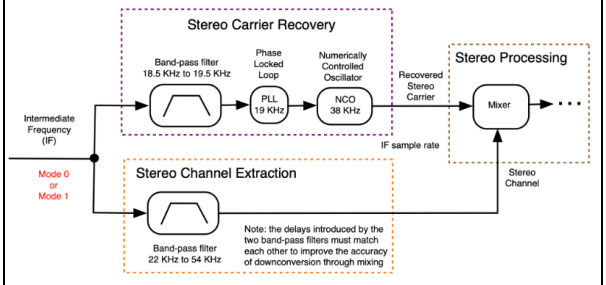


Figure 6: Stereo channel extraction and carrier recovery

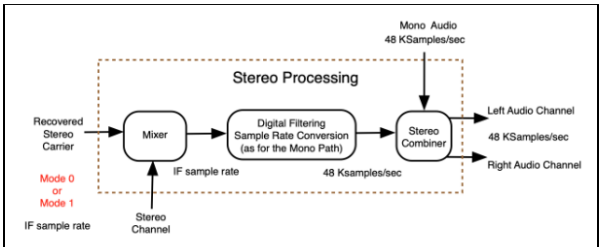


Figure 8: Stereo downconversion, filtering and combining

3.5 RDS Path

The RDS path has four main components: RDS Channel Extraction, RDS Carrier Recovery, RDS Demodulation, and RDS Data Processing as shown in the figure below. From the image, it can be seen that the RDS Channel Extraction requires a band-pass filter to extract the RDS channel from 54 kHz to 60 kHz. For the RDS Carrier Recovery, squaring non-linearity, band-pass filter, PLL, and NCO was implemented. To implement squaring non-linearity, the extracted RDS signal was multiplied by itself. The result is then used as an input to a narrow band-pass filter which was used to extract a 114 kHz tone. Then, the PLL and NCO are used to output a 57 kHz signal used for mixing to perform downconversion in the RDS demodulator.

RDS Demodulation process includes mixer, low-pass filter, rational resampler, root-raised cosine filter (RRC), and clock and data recovery. In the RDS Demodulation process, optimization is achieved by combining the mixer with the low pass filter so that the calculations are only being performed for the values that are needed after downsampling. The data was mixed and convolution was performed, then resampling resulted in a multiple of 2375 symbols per second. For modes 0 and 2, the expected frequencies are 23 (Samples Per Symbol, SPS) * 2375 = 54625Hz and 34 (SPS) * 2375 = 80750Hz. The given intermediate frequency is 240KHz, to reach the expected frequency by modes 0 and 2, the corresponding rational upsampling factors and downsampling factors are 437, 1920, and 323, 960 respectively. Then we passed this through an RRC impulse response to achieve the RRC coefficient which was then passed through the filter. The threading for RDS works similarly to the threading done in stereo.

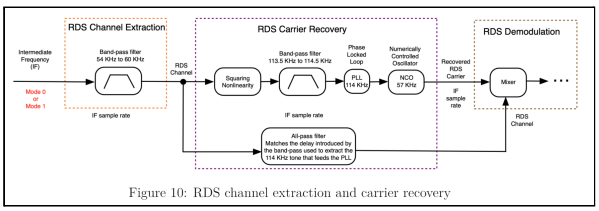


Figure 10: RDS channel extraction and carrier recovery

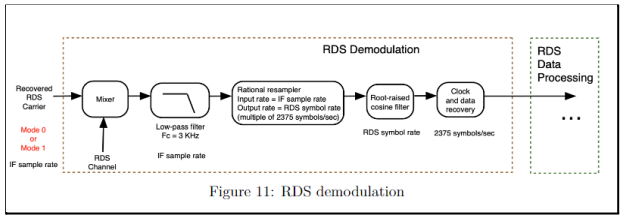


Figure 11: RDS demodulation

4 Analysis and Measurements

Mode 0:

Process	Runtimes per block (ms)	Multiplications per block	**Number of taps:** 151 **RF Downsample factor:** 10 **IF Downsample factor:** 5 **Number of taps:** 151 **Block size:** 1024 * 10 * 5 * 2 = 102400
Filtering (convolution) and decimation for I and Q data	7.998	77312	
Demodulation	0.151	5120	
Mono filtering and decimation	0.549	20480	
Carrier recovery	2.702	773120	

FmPLL	1.629	5120	
Channel extraction	2.17	773120	
Mixing	0.018	5120	
LPF after mixing	1.648	1024	
Combiner	9.667	1024	

Mode 1:

Process	Runtimes per block (ms)	Multiplications per block	
Filtering (convolution) and decimation for I and Q data	9.594	115968	**Number of taps:** 151 **RF Downsample factor:** 8 **IF Downsample factor:** 6 **Number of taps:** 151 **Block size:** 1024 * 8 * 6 * 2 = 98304
Demodulation	0.185	6144	
Mono filtering and decimation	0.536	24576	
Carrier recovery	3.369	927744	
FmPLL	1.872	6144	
Channel extraction	2.619	927744	
Mixing	0.021	6144	
LPF after mixing	2.426	1024	
Combiner	11.463	1024	

Mode 2:

Process	Runtimes per block (ms)	Multiplications per block	
Filtering (convolution) and decimation for I and Q data	218.376	3010560	**Number of taps:** 151 * 147 = 22197 **RF Downsample factor:** 10 IF Upsample factor: 147 **IF Downsample factor:** 800 **Block size:** 1024 * 147 * 2 = 3010560 **Accumulation:** 301056 (in combiner)
Demodulation	4.463	301056	

Mono resampling	185.928	3010560	
Carrier recovery	78.024	1638400	
FmPLL	48.150	301056	
Channel extraction	63.856	1638400	
Mixing	0.612	301056	
LPF after mixing	44.056	2048	
Combiner	126.333	2048	

Mode 3:

Process	Runtimes per block (ms)	Multiplications per block	**Number of taps:** 200 * 441 = 22197 **RF Downsample factor:** 3 IF Upsample factor: 441 **IF Downsample factor:** 3200 **Block size:** 1024 * 442 * 3 * 2 = 2709504 **Accumulation:** 903168
Filtering (convolution) and decimation for I and Q data	652.351	2709504	
Demodulation	12.697	903168	
Mono resampling	160.828	4675100	
Carrier recovery	233.782	2470400	
FmPLL	139.880	467510	
Channel extraction	189.600	2470400	
Mixing	1.705	903168	
LPF after mixing	46.018	2048	
Combiner	258.351	2048	

Non-linear function:
Cos function: 2 * 151 = 302
Sin function: 151

5 Proposal for Improvement

To further improve the performance of the system, one possible solution is to implement the pointers in threading functions. For example, instead of pushing variable fm_demod to the data queue, using pointers that point directly to the address that stores this value could improve the function performance by eliminating the time for the machine to find the physical address of the variable. Such improvement would be significant if we need to handle a larger size of the data block making it more efficient.

For user experience, if the performance of the program could be improved, we can utilize a larger number of taps to improve the audio quality, without underrun while running the program in real-time. Furthermore, a de-emphasis filter could be implemented at the end of the stereo thread. During transmission of rf data, the built-in pre-emphasis filters are utilized to attenuate certain frequencies, which cause the downgrade to the audio quality. By implementing the de-emphasis filter we can recover the audio data before the pre-emphasis filter, therefore, improving audio quality.

6 Project Activity

Week	Project Progress	Contribution
Week 1 - Feb 14	Project specifications released	
Week 2 - Feb 21	Midterm recess	
Week 3 - Feb 28	Reviewed the project document	Group
Week 4 - Mar 7	Modifications to Lab 3 code in python	Group
	Refactored the fmDemod from lab 3 to C++	Rutvi, Priya
	Refactored the code for mono mode 0 from lab 3 to C++	Rutvi, Priya, Zhuochen
Week 5 - Mar 14	Mono Mode 0 completed	Rutvi, Priya, Zhuochen
	Mono mode 1 completed	Rutvi, Priya
	Implemented the model code for stereo in python	Zhuochen, Hengbo
	Slow Resampling	Hengbo
Week 6 - Mar 21	Stereo model C++	Rutvi, Priya
	fmPLL to C++	Zhuochen, Rutvi, Priya
	Fast Resampling function	Zhuochen, Hengbo
	Mono mode 2 and 3 completed	Zhuochen, Hengbo
	Combining mode 0 and 1	Rutvi, Priya

	Mode select completed	Zhuochen
Week 7 - Mar 28	Threading	Zhuochen
	RF front end refactor C++	Rutvi, Priya
	RDS model python	Zhuochen
	RDS model C++	Rutvi, Priya
	Finalizing source code	Group
Week 8 - Apr 4	Presentation	Group
	Project Cross-Examination	Group
	Report: Introduction/ Conclusion/ Project Overview	Rutvi, Priya
	Report: Implementation Details (Labs, Mono Processing Path Modes 0 & 1, RDS Path)	Rutvi, Priya
	Report: Implementation Details (Stereo Processing Path)	Zhuochen, Rutvi, Priya
	Report: Implementation Details (Mono Modes 2 & 3)	Hengbo
	Report: Analysis and Measurements	Zhuochen, Hengbo
	Report: Proposal for Improvement	Zhuochen, Rutvi, Priya

7 Conclusion

In conclusion, this project has been a great learning experience and allowed us to consolidate some of the course concepts to better apply them to a real-time implementation of a computing system. Further working as part of a large group, we have improved our time management skills and collaboration which can be assets for future projects. Even though the project had many different layers, we learned that dividing the project into smaller tasks (i.e. mono, stereo, etc.) allowed us to progress further as a group. This project has been a great first-time experience to explore using real-time implementation techniques and we have grasped new skills that can be used in the future. Overall, the project was challenging but as a group, we were able to tackle and move past the difficulties.

8 References

- Project Description
- Lectures and Notes
- Lab 1, Lab 2, and Lab 3