ENGINEERING

McMaster University

Final Report – Software Defined Radio

COMPENG 3DY4 – Computer Systems Integration Project

Group – 19

Ranuja Pinnaduwage– 400269531

Hamza Saeed – 400319189

Pragya Khanna - 400336689

Patrick Wang – 400264434

April 10, 2023

Introduction

This project was designed to amalgamate course material learned through the last three years of a computer engineering degree. The task of creating a Software Defined Radio (SDR) utilized programing techniques as well as several mathematical concepts learned in applied and theoretical courses. These concepts include knowledge of data structures, convolution, Fourier transforms, communication systems, and even control systems. Beyond these concepts, this course also taught techniques of applying the previously learned techniques to a real-time system. Concepts such as block processing, multithreading, and code optimization were some of the techniques that were useful in creating the building blocks of this project. This entire course being a project course, each lecture provided information on how the project elements can be implemented for the best performance. This provided an insight into how real-world systems are developed and the thought process behind such projects.

Project Overview

A Software Defined Radio (SDR) is a device that can use a single computing unit to receive and transmit radio signals. This project aimed to tackle the receiving portion of an SDR, where the signals are then processed to produce audio. The project consisted of four main blocks; the RF block, where radio signals are processed into demodulated data. The Mono block is where the mono audio portion of the signal spectrum is extracted for playing. The stereo block, where stereo audio is acquired and processed with mono data for left and right ear playing, and finally, the RDS block. In this block, the radio description, such as the song being played, is extracted.

The system used to run the SDR is a Raspberry Pi 4 Model B, which comes with a quad-core Cortex-A72 Processor and 4GB RAM. The software development was primarily done on a Raspberry Pi virtual machine, where GitHub was used for version control. The Raspberry Pi runs using the Linux operating system and therefore requires the mastery of Linux commands. An RF dongle and antenna receive Radio Frequency signals (IQ samples) and send them to the Pi through a USB connection. Linux commands are used for piping this RF data into the project to be processed, which is how this project is a real-time system.

In the RF Block, the RF signals are acquired through the Unix pipe, where they are down-sampled and low-pass filtered so that there is access to up to 100kHz. The purpose of this is to lower the number of samples that would have to be further processed in the later blocks whilst also only maintaining the useful range of frequency data. The data at this point is still in the form of in-phase and quadrature samples; the data is demodulated to merge the IQ samples into a single data signal. This concludes the work of the RF block, where the purpose is to acquire RF data and transform it into demodulated data for use by the other blocks.

The mono and stereo blocks both conduct similar processes where blocks take the demodulated data and process it into audio signals. The mono signal is stored from the 30Hz to 16kHz range, where a lowpass filter can be used to acquire this signal. The filtered signal will also be down-sampled to the final playing rate. Like mono processing, the stereo signal is stored from the 23KHz to the 53KHz range, except a bandpass filter must be used to acquire this signal. The main difference is that the stereo data requires a carrier signal to recover the original signal. The carrier is stored at 19kHz, where a phased locked loop (PLL) must be used to recover the phase shift in the signal. Once recovered, the stereo signal can be multiplied with the PLL to acquire the final stereo signal, which is also down-sampled to the final playing rate. Since stereo aims to offer a left and right ear playing experience, the stereo final stereo data must be further processed to acquire the left and right signals. The mono signal, defined as (L+R), must be combined with the stereo signal, defined as (L-R), in such a manner as to reproduce the final signal.

The RDS block begins by undertaking a similar process to the stereo block. The signal, which is located from 54KHz to 60KHz, must be extracted by a bandpass filter, and it must be multiplied by a carrier signal. However, the carrier signal must be formed from the extracted RDS signal. This signal is squared to acquire a carrier signal at 114KHz which must be bandpass filtered and then multiplied by the RDS signal. From here, the signal must be lowpass filtered at 3KHz and resampled to the correct number of symbols per second. The signal must then be passed through a root-raised cosine filter, and the data must be converted to binary through clock and data recovery. After utilizing Manchester decoding and frame synchronization, the radio final data can be recovered.

The final aspects of this project include using another Unix pipe to pipe the processed data from the project to aplay to hear the radio. Doing so makes the project fully real-time, from data acquisition from the RF dongle being piped into the project to the output being piped to an audio-playing software. Further improvements to the project can be made through multithreading and other optimization techniques that utilize the computer architecture to its maximum potential.

Implementation Details

The labs provided the basic building blocks for completing the project. Lab one introduced the concept of working with audio data and producing a sound using Python. It taught concepts such as filtering and Fourier transforms and how they can be applied through code. Additionally, the lab introduced a fundamental concept for optimization, block processing. Lab two further explored these concepts by accomplishing the same tasks through C++ instead of Python. This gave the building blocks of refactoring code into a more efficient form since C++, although more complicated to develop, provides a faster result. Finally, lab three required the creation of two blocks used in the final project. This is the RF block and the Mono block; these blocks were developed using Python and a pre-made file of IQ samples. This lab gave an idea as to how one can utilize IQ samples and transform them into sound.

When developing the RF front-end block, mono block, and stereo block, they were first conceptualized in Python, then refactored as C++ code to be tested on a RPI virtual machine. Afterwards, they were optimized for run time by implementing block processing, or the fast method. When the virtual machine successfully ran the aforementioned blocks and produced a desired sound quality, testing was then moved towards the physical RPI, and final adjustments were made to optimize the code to allow live sampling and playing.

For the RF front-end block and mono block, the major functionality was already implemented in C++ from working on the labs, thus most of the work being done was for optimizing the run time and implementing mode 1, 2, and 3. The RF block is composed of a low pass filter, decimator, and a demodulation block, where the demodulation and low pass filter were already implemented. The decimator block was implemented using the fast method, by modifying the low pass filter block so that it also down samples each and performs convolution at the same time to optimize run time.

For the mono block, the fast down sample method was already implemented in the RF block, but for the remaining 3 modes, up sampling needs to be added. Both up sampling and down sampling were done at the same time and implemented in the low pass filter block in the mono block. A new function was implemented to resample blocks of data while doing convolution to optimize run time.

For the stereo block, the fast methods for resampling and down sampling can be reused, so the only new methods that were developed were the Phase Locked loop function, the mixer block to demodulate the stereo signal and the addition block to generate the left and right channel for audio. The mixer block and addition block were simply implemented by doing index wise multiplication and addition respectively. As for the phase locked loop function, the given python code was refactored into C++ and the NCOscale variable was set to 2, in order to turn the 19Khz carrier signal to turn it into a 38khz carrier signal that is in phase with the modulated stereo signal.

After optimizing the run time of our code, usually we run into issues such as segmentation faults, large amounts of noise in the output audio, or the output audio is not playing. Segmentation faults were resolved by using print statements to locate them and they were usually resolved by fixing C++ vector declarations as the segmentation faults were caused by an out of index error for vectors. When no sound played or if the output contained too much noise, we troubleshooted it by printing values in the vectors that stored the audio data as well as plotting the time and frequency domain using GNU plot. Plotting such data also functioned as verification that the data processing was correct, in addition to listening to the output audio.

Analysis and Measurements

Num Taps = 51

In the project we set the number of taps to 51 instead of 101, so the base case for our group would be 51.

Process	Functions	Multiplication/ block	Runtime/block (s)
RF Front end	Filtering and decimation	104448	0.006130
	Demodulation	20480	0.000167
Mono Mode 0	Filtering and Decimation	1044480	0.000488
Mono Mode 1	Filtering and Decimation	522240	0.000488
Mono Mode 2	Filtering, Up sampling and Decimation	3010560	0.001744
Mono Mode 3	Filtering, Up sampling and Decimation	76769280	0.003681
Stereo Mode 0	Carrier Recovery	1044480	0.002845
	Channel Extraction	1044480	0.003052
	PLL	20480	0.0003910
	Mixing	20480	0.0001098
	Carrier Removal (LPF)	20480	0.0005687
	Combiner	20480	0.0002458
Stereo Mode 1	Carrier Recovery	522240	0.002755
	Channel Extraction	522240	0.003152
	PLL	20480	0.0003840
	Mixing	20480	0.00001258
	Carrier Removal	20480	0.0005180
	Combiner	20480	0.00002166

Num Taps = 13

Mono Mode 0: for this mode the number of taps is 13 and the block size is 102400. The samples are decimated by a factor of 5 to produce the audio output, therefore the number of multiplications for this mode is 266240 per block.

Stereo Mode 0: for this mode the number of taps is 13 and the block size is 102400. The same amount of multiplication is done twice due to carrier recovery and channel extraction, this section has more multiplication than mono because there is no decimation.

Process	Functions	Multiplication/block	Runtime/block (s)
RF front end	Filtering and Decimation	26624	0.001678

	Demodulation	5120	0.000269
Mono Mode 0	Filtering and Decimation	266240	0.000239
Stereo Mode 0	Carrier Recovery	266240	0.000289
	Channel Extraction	266240	0.000289
	PLL	5120	0.000986
	Mixing	5120	1e-5
	Carrier Removal	5120	2.1e-5
	Combiner	5120	0.000220

Num Taps = 101

Mono Mode 0: for this mode the number of taps is 101 and the block size is 102400. The samples are decimated by a factor of 5 to produce the audio output, therefore the number of multiplications for this mode is 2068480 per block.

Stereo Mode 0: for this mode the number of taps is 101 and the block size is 102400. The same amount of multiplication is done twice due to carrier recovery and channel extraction, this section has more multiplication than mono because there is no decimation.

Process	**Functions**	**Multiplication/block**	**Runtime/block (s)**
RF front end	Filtering and Decimation	208896	0.010449
	Demodulation	40960	3.34e-4
Mono Mode 0	Filtering and Decimation	2088960	0.000976
Stereo Mode 0	Carrier Recovery	2088960	0.000989
	Channel Extraction	208896-	0.000989
	PLL	40960	0.000378
	Mixing	40960	1e-5
	Carrier Removal	40960	0.00305
	Combiner	40960	0.000340

Num Taps = 301

Mono Mode 0: for this mode the number of taps is 301 and the block size is 102400. The samples are decimated by a factor of 5 to produce the audio output, therefore the number of multiplications for this mode is 1044480 per block.

Stereo Mode 0: for this mode the number of taps is 301 and the block size is 102400. The same amount of multiplication is done twice due to carrier recovery and channel extraction, this section has more multiplication than mono because there is no decimation.

Process	**Functions**	**Multiplication/block**	**Runtime/block (s)**
RF front end	Filtering and Decimation	192184	0.03084
	Demodulation	38092	6.4e-4
Mono Mode 0	Filtering and Decimation	1921840	0.003211
Stereo Mode 0	Carrier Recovery	1921843	0.003042
	Channel Extraction	1921843	0.003042
	PLL	38092	0.000378
	Mixing	38092	5e-6
	Carrier Removal	38092	0.00247
	Combiner	38092	0.000330

Mono Mode 0: for this mode the number of taps is 51 and the block size is 102400. The samples are decimated by a factor of 5 to produce the audio output, therefore the number of multiplications for this mode is 1044480 per block.

Mono Mode 1: for this mode the number of taps is 51 and the block size is 61440. The samples are decimated by a factor of 6 to produce the audio output, therefore the number of multiplications for this mode is 522240.

Mono Mode 2: for this mode the number of taps is 51 and the block size is 16384000. The samples are up sampled by a factor 147 and decimated by a factor of 800 to produce the audio output, therefore the number of multiplications for this mode is 3010560.

Mono Mode 3: for this mode the number of taps is 51 and the block size is 13107200. The samples are up sampled by a factor 147 and decimated by a factor of 1280 to produce the audio output, therefore the number of multiplications for this mode is 76769280.

Stereo Mode 0: for this mode the number of taps is 51 and the block size is 102400. The same amount of multiplication is done twice due to carrier recovery and channel extraction, this section has more multiplication than mono because there is no decimation.

Stereo Mode 1: for this mode the number of taps is 51 and the block size is 102400. The same amount of multiplication is done twice due to carrier recovery and channel extraction, this section has more multiplication than mono because there is no decimation.

Comparing mono and stereo, the processing time for stereo is almost double that of mono, this is due to the number of multiplication and extra steps to require to calculate stereo, proving that the number of multiplications and runtime are linearly dependent.

As we increase the number of taps, the number of multiplications also increase, this increases the time required for extraction of data and processes that involve block convolution. The processes that include subtraction and addition, for example the combiner, stay around the same. As we decrease the number of taps, the number of multiplications decreases, this decreases the time required for extraction of data and processes that involve block processing. The base case was 51 taps, increasing/decreasing the taps increases/decreases the number of multiplications based on the decreasing and increasing factor.

While processing the RF samples, the time increases linearly as the number of taps increase, therefore the overall computational time also increase in a linear manner.

In terms of the audio quality, the audio quality was observed to get distorted when the number of taps were increased too much or decreased too much.

Proposal for Improvements

Voice quality can be improved by adding a de-emphasis filter at the end of the mono block. Though not required in the project, a de-emphasis filter would benefit the users since its purpose is improve the signal to noise ratio for audio signals boosted by pre-emphasis [1]. FM transmitters usually use pre-emphasis before sending the signal data to a receiver, which was noticeable by the trace amounts of noise left over, despite implementing all the required components in the mono block [2].

One method to test and debug the project is to take the Fourier transform of the audio data and plot it in the frequency domain in order to verify its performance as well as its accuracy. This would pertain to portions of the project such as the PLL, where it can be checked if the acquired carrier is the correct frequency. Using Fourier transform to process data then again using the inverse Fourier transform is very inefficient. The Fourier transform can be optimized to use the fast Fourier transform method, which would speed up the time it takes to verify the results in the frequency domain. Even though the function will not be used in processing the input signal, the time complexity of the traditional Fourier transform is $O(n^2)$ while the fast method is $O(n\log n)$. With the faster method, it would be easier to test and verify very large audio samples [3].

Run time can be further improved by modifying the convolution functions. By splitting up the single nested for loop into multiple loops, the run-time can be cut in half for each convolution function call. This is because redundant decisions can be avoided to save time in the overall run. A basic implementation of this idea could be to split the block and state vector usage, where after the outer for-loop index exceeds the impulse response size, then there is no need to check if state is to be used. This is because the state would only be necessary for the point at which the subtraction of the two loop indices becomes negative, which would never happen after the above scenario [4].

Project Activity

	Week 1	Week 2	Week 3	Week 4	Week 5
Ranuja Pinnaduwage	Implemented Mono block mode 0 and RF block processing	Implemented Down and Up Sampling For Modes 1,2,3	Implemented Fast down sampling and resampling Tested and debugged Mono Finished Modes 1, 2,3 Started Stereo Implemented PLL	Started RDS Debugging Stereo Finished Stereo	RDS Carrier Recovery and Demodulatio n RDS Root Raised Cosine filter Multi-threading
Patrick Wang	Implemented Mono block mode 0 and RF block processing.	Helped with Down and Up Sampling and Tested and Debugged for Modes 1,2,3	Helped with fast down sampling and resampling. Tested and debugged mono Finished Modes 1, 2,3 Started Stereo Implemented PLL Implemented C++ audio playing	Debugging Stereo Finished Stereo	RDS Carrier Recovery and Demodulatio n RDS Root Raised Cosine filter Helped with Multi-threading

Pragya Khanna	Worked on pipelining. Mono mode 0,1 implementati on RF front end implementati on	Mono mode 0,1,2,3 implementati on and live testing Started learning about stereo.	Worked on stereo implementation . Tested and debugged mono	Tested mode 1,2,3 for stereo Debugged stereo Helped fixing the low pass filter	Commented the code Helped collecting samples with different RF for testing
Hamza Saeed	Setup the pipelining and the interface RF front end implementati on Collected samples for testing.	Live tested mode 1,2,3 for mono Tested the functions for accuracy	Worked on stereo implementation and implemented aplay combined the files to be played on one run	Tested the mono and stereo mode. Made mono mode 1,2,3 work. Fixed the lpf to compensat e for the gain.	Helped with multithreadin g. Differentiate d modes based on the input parameters Collected samples with different RF for testing

Conclusion

Overall, the 3DY4 project has provided all group members with valuable educational experience. We acquired practical knowledge about commercially available computer systems operating in a space-constrained context through the actual development of a real-time SDR system. Through this project, participants could use their theoretical knowledge of signal processing, FM technology, and digital communication standards while becoming familiar with the Software Development Life Cycle (SDLC) and source-code management procedures. This project has improved the technical proficiency of the team members. However, they have also better understood signal-processing concepts and how to use them in real-world settings. The members also improved their collaboration, project management, and problem-solving ability. The skills and knowledge members acquired because of this project will be extremely helpful to them as they pursue careers in the computing industry in the future.

References

[1] W. Jahn. “De-Emphasis.” Analog Devices. https://wiki.analog.com/resources/tools-software/sigmastudio/toolbox/filters/deemphasis#:~:text=The%20De%2DEmphasis%20filter%20is,during%20recording%20with%20pre%2Demphasis [Accessed April 2, 2023]

[2] “Pre-Emphasis and De-Emphasis in FM Broadcasting | Introduction.” https://www.fmradiobroadcast.com/article/detail/fm-emphasis.html [Accessed April 2, 2023]

[3] K. Cheshmi. (2023). Week6-2. [PDF]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/528090/viewContent/4076679/View

[4] K. Cheshmi. (2023). Week6-1. [PDF]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/528090/viewContent/4073242/View