COE3DY4 Project Report

Group 41

Linda Meng, Kevin Li, Laura Ju and Kevin Chen

mengy30, lik96, juw3 and chenz231@mcmaster.ca

April 5, 2020

Introduction

The purpose of the project is to experiment and implement a software-defined radio (SDR) system involving front-end radio-frequency(RF) hardware. With this hardware and the correct implementation of the software, the goal is to be able to listen to the real-time reception of frequency modulated (FM) mono/stereo audio. To listen to the live samples, a raspberry pi machine, and RF dongle must be set up and configured. The system can be used in two modes 0 and 1, which collect samples at 2.4MSamples/sec and 2.5MSamples/sec respectively. Additionally, for the proper operation of Mono and Stereo, the system must be able to operate in the frequency of 0-15KHz for the FM subchannel of Mono and 23-53KHz for Stereo.

Project Overview

To understand the project, some key concepts must be understood. Frequency modulation is a modulation technique where the frequency of the carrier wave is changed corresponding to the instantaneous amplitude of the modulating signal. Radios operating in FM use frequency radios, hence FM radios.

Software-defined radios are radios where traditional hardware solutions are replaced with software solutions. Some particular examples of hardware that can be translated to the software include filters, demodulators, and mixers. Through the use of a computer such as a raspberry pi and a C++ compiler or equivalent, and a RF front end which includes the receivers antenna, a SDR can be implemented.

A Finite Impulse Response filter or FIR filter is a digital filter that is used in data signal processing. A finite impulse response includes crucial coefficients that will define the properties of the response such as cutoff frequency, sample frequency, and the number of taps that define the length of the filter which directly corresponds to the ‘quality of the filter.

Since radio stations broadcast in FM or frequency modulation, an FM demodulator is required to recover the original modulating signal. The three sub-channels of interest of the demodulated FM channel are the Mono, Stereo, and RDS sub-channels in this project. In FM, information is stored as variations in frequency and thus in this project, the FM demodulator must be able to take in the In-phase and Quadrature (IQ) data from the receiver and retrieve the original three subchannels.

Phase-locked loops (PLL) is a type of control system which functions as a signal generator in which the output signal is synchronized with the phase of the input signal. Additionally, the goal of PLLs is to produce a filtered output in lock to noisy input. A PLL requires a numerically controlled oscillator which will generate the frequency to match the original carrier frequency. A PLL also requires a phase comparator that compares the output signal’s phase to the phase of the received FM wave.

Resamplers such as upsamplers and downsamplers are also very important in Data Signal Processing(DSP). Downsampling is a form of compression where if a sample is downsampled by M, then only every M sample would be kept. Essentially, recreating a scenario where the signal is sampled at a lower rate. Contrastingly, upsampling is the process where additional data is introduced in between the collected samples. Unlike downsampling, where samples in between can be deleted, numerical samples cannot be inserted in between the original samples so all the inserted samples must have zero-value. Upsampling is a method of artificially increasing the sampling rate.

To recreate an FM receiver in software, the In-phase and Quadrature data must go through a low pass filter to filter out the unwanted frequencies that are irrelevant to this project and also to act as an anti-aliasing filter. Afterward, the data will go through a downsampler and then both I and Q samples must go through a FM demod function which will act as input for the upcoming Mono/Stereo implementations.

Implementation details

RF Front-End Processing

In this project, we have two inputs with different sample rates, 240k and 250k samples per second which corresponds to mode 0 and mode 1.

To extract a positive FM channel, we set the cutoff frequency to be 100kHz. After we read a block of data from the raw file, we split the data into I data and Q data. The even number data is in phase data (I), and the odd number data is the quadrature data (Q). There are two separate paths for I and Q data.

First, we took them as the input for the function from lab3 to calculate the impulse response for both I data and Q data. After this function, we got filter coefficients for both data. Then we used the convolution function in lab3 to take the convolution of the filter coefficients and I data or Q data. Then we used our own FM Demod function to recover the signal. The FM Demod function is based on the function given in the python model. The recovered signal equals to (I[k]*(Q[k]-Q[k-1])-Q[k]*(I[k]-I[k-1]))/(I[k]^2+Q[k]^2). After these processes, RF Front-End Processing is completed.

Mono Processing

For mono mode 0, we set the cutoff frequency to be 16KHz to extract the Mono channel from the FM channel which has a sampling rate of 240k samples/sec, and we reduced the sample rate by 5 because of the final output should have a sample rate of 48k sample/sec. We reduced the sample rate by simply keeping one sample out of five samples. We didn't find many problems in this part except we made a small mistake in the convolution function. We didn't figure it out at the beginning because we followed exactly the same steps we did in python and the python model worked correctly. Then we checked the code again and we found the problem was because the negative index in the array works differently in python and C++. This mistake made us pay specific attention to the array sizing in the later stages of the projects.

After we successfully implemented mode 0, we started working on mode 1. Mode 1 has an input sample frequency of 250KHz. 250KHz cannot be fully divided by 48KHz, so we wrote an upsampler with rate 24 which simply inserts 23 zeros between each sample, and then we extracted the Mono channel as we did in mode 0. Finally, we reduced the sample rate by 125 to produce an output signal with a 48KHz sample rate. After these processes, the mono path is finished.

When mode 0 was running, the audio sounds good and doesn't have any underruns. But when mode 1 was running, there were underruns between blocks. The output audio will have no sound on some period due to the underruns. Therefore, we optimized our convolution function. At first, our convolution function will multiply every sample in the block with the impulse response, which is really inefficient. Thus, we added the downsampling rate as a new input to the convolution function. After the modification, the convolution will only convolute those samples that will be held after the downsampler. This small change increased the convolution speed 125 times faster than before. This change made to convolution was also used in Stereo processing and RDS processing.

Stereo Processing

In stereo processing, we first used two bandpass filters with pass bands of 18.5KHz to 19.5KHz and 22KHz to 54KHz to extract the stereo pilot and the stereo audio. The bandpass filter was implemented very similarly to the low pass filter. The only difference was the impulse response. To calculate the impulse response for the bandpass filter, we defined a normalized center frequency to apply a frequency shift, and we followed the pseudo-code given.

After we extracted the stereo pilot, it will go through the Phase Locked Loop (PLL) and Numerically Controlled Oscillator (NCO) to recover the stereo carrier. We wrote PLL and NCO functions in C++ based on the python code given. In addition, we added five states to save between blocks. Then we did the pointwise multiplication for the recovered stereo carrier and the stereo audio. The mixer will be simply 2 times recovered stereo carrier and then times stereo audio.

In order to have a 48KHz output signal, we did the digital filtering to the signal after mixing which is the same as the mono path. If the program was operated in mode 0, the signal will first go through a 16KHz low pass filter and then reduce the sample rate by 5. If the program was operated in mode 1, the signal would first go through an upsampler with rate 24 and then go through a 16KHz low pass filter and reduce the sample rate by 125. After reducing the sample rate to 48KHz, we combined the stereo audio with the mono audio. The output has both the left audio channel and the right audio channel. Because the Mono audio is equal to the left channel plus the right channel, and the Stereo Audio is equal to the left channel minus the right channel. The left audio channel equals the Stereo audio plus the Mono audio and divided by two, the right audio channel equals the Mono audio channel minus the Stereo channel and divided by two. Finally, we combined these two channels by putting them in an array, the even number indexes save the left channel and the odd number indexes save the right channel. After these processes, stereo processing is finished.

The difficulties we met in stereo processing are the existence of noise and lag in the wav file. We used two approaches to resolve this problem. First, we found that it is inefficient if we set the size of the output array in the beginning rather than increasing it by the size of the output array of one block in each for loop. Therefore, we declared the size of the output array of one block and multiplied it by the number of blocks. We also modify the mono processing in the same way. Second, we optimized the convolution function that will only be used in stereo mode 1. After these modifications, the output audio sounds good and it runs without underrun on a virtual machine, but it still has underrun on Raspberry Pi. Then we added threads for Audio processing and RF front-end processing. Once the threads have been added, the program can run without underruns on Raspberry Pi.

We also plotted the time domain graph of the signal going through the PLL and NCO in the python model, and we saw discontinuities between blocks. Then we checked the PLL function again and found that we forgot the argument of the trigoffset.

RDS Processing

In RDS processing, we first used a bandpass filter with a passband frequency of 54KHz to 60KHz to extract the RBDS and we multiplied samples that we extracted with themselves. Then we used another bandpass filter to extract a 114KHz tone because the squaring has doubled the center frequency of the RDS channel. We used the PLL and NCO to produce the in-phase output and the quadrature output with the center frequency of 57KHz. We modified the fmPLL function from the stereo part. We added a quadrature output which equals sin(trigArg*ncoScale + phaseAdjust) and the quadrature output is exactly 90 degrees out of phase with the in-phase output. The last value in quadrature output also needs to be saved. In RDS the output of NCO has a center frequency of 57KHz which means the ncoScale needs to be 0.5. Then we mixed both in-phase data and quadrature data with the data that was extracted by the 54KHz to 60KHz bandpass filter. Both in-phase data and quadrature data after mixing went through the 3KHz low pass filter and right now the sampling frequency was 240KHz but we want it to be 57KHz after the rational sampler. So we designed an upsampler with the sample rate of 19 and a downsampler with the sample rate of 80. The maximum cutoff frequency of the low pass filter between the upsampler and the downsampler can be determined by the function min{U/D*IF/2, IF/2}. U is upsampling rate and D is downsampling rate, IF is the intermediate frequency. In this filter, U is 19, D is 80 and IF is 240KHz, so the maximum cutoff frequency is 28.5KHz. The cutoff frequency we chose was 19KHz. In order to shape and reduce inter-symbol interference, the RRC filter was introduced. The impulse response of the RRC filter was calculated by the sampling frequency and the number of taps, and we translated the given python code to c++. After the in-phase data and the quadrature data go through the RRC filter, we plotted the time-domain graph and checked if the signal has a width of 24. We found that not all signals have a width of 24, some signals have a width of 48. That was due to the signal not idealized. Then we wrote an algorithm to choose the starting point for clock and data recovery. The algorithm basically finds the position of the smallest magnitude in 24 samples. Because the in-phase data was cosine wave, if the position was less than 23 it means that the width of the first signal is 24, so the starting point is point 0. If the position was larger than 23, it means that the width of the first signal is 48, so the starting point is point 12. Because we process the signal in blocks, sometimes there are still samples left but less than 24, so we saved the number of samples left in the current block and used it to determine the starting point in the next block.

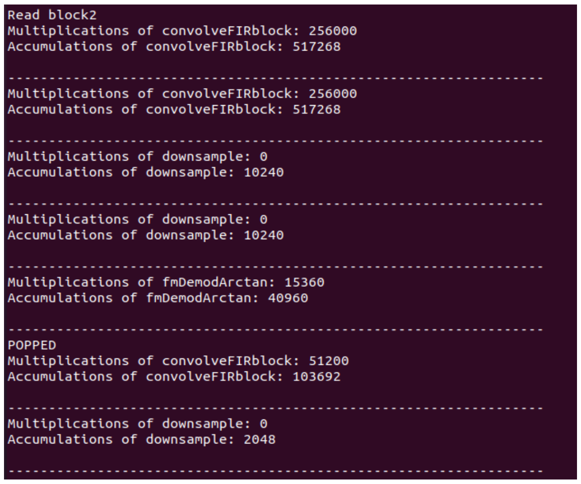
Then we plotted the constellation graph for i and q, but we couldn't see the clear separation between all the sampled H and L symbols. There were always some points set near the center of the graph. When we increased the block size, we could clearly see the two H and L symbols, they were mostly separated but still had some points set near the center of the graph. That might due to the signal not being idealized. Then we processed to Manchester and differential decoding. Manchester was simply taking two samples and if the samples are in the sequence of HL, the output is 1, if the samples are in the sequence of LH, the output is 0. The differential decoding was the same as taking the samples connected to an XOR gate, the mathematical function for the XOR gate is x'y+y'z assuming the inputs are x and y. Then we took 24 samples from the output of differential decoding and multiplied them with a 24x10 matrix. The output was a 1x10 matrix, and we compared it with the syndrome word given. Our outputs were wrong but we didn't have time to figure that out.

Threading

Before adding threading in the project, stereo mode 0 and mode 1 had underruns when testing in real-time. In order to implement threading, a model was made in C++ to run two threads concurrently. At first, the model had read errors and Makefile errors. Both errors were eventually solved by importing the thread plugin and adding the -pthread in the Makefile. To ensure the two threads were indeed working properly, each function would print something to show they are both running. A synchronization queue was later implemented in the model to allow data transfer between the two threads. Then, one function was made to push something in the queue while the other one was made to pop from it in the model. This was proved to be working by having both of them printing "POPPED" or "PUSHED" after the corresponding action. After this stage, RF-frontend and mono-processing were taken out from the main function and put into their own separate files in order to create the two threads. The concept of the model was then applied to the actual program. Segmentation faults occurred when implementing threading to RDS. This error is caused by thread popping the block instantaneously after it reads from it. This means either the mono-stereo or RDS thread reads from the synchronization queue first and pops the block from the queue, and the other thread would not have access to the same block anymore. Either keeping track of whether both threads are done with the block before popping it or having two synchronization queues would solve the problem. Since time had come close to the deadline, the latter approach was taken despite the extra memory and runtime it would create. Although having two queues did solve the segmentation faults, the program would now only run a few blocks before it gets stuck. We believed there were some finer details we needed to fix in the code in order for RDS threading to work properly. Due to the lack of time and help from office hours, RDS threading was left unfinished.

Analysis and measurements

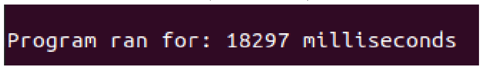
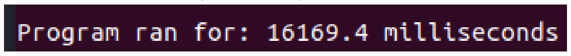
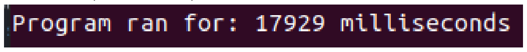
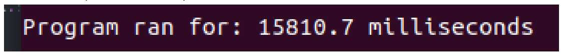
A sample output of the amount of calculations is shown on the right. All possible accumulations/additions and multiplications were added up for each block. Note that divisions were considered to me multiplications since dividing is essentially $\frac{1}{x}$. Afterwards, since each we know the sample size of each block and how many calculations per block, the operations per audio sample could be calculated which is seen below:
**Note* The samples in a block for mode 0 is 2048, and the samples in a block for mode 1 is 1064*



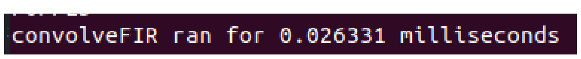
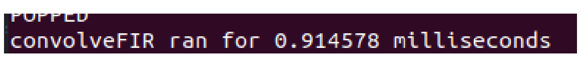
```
Read block2
Multiplications of convolveFIRblock: 256000
Accumulations of convolveFIRblock: 517268

----------------------------------------------------------------
Multiplications of convolveFIRblock: 256000
Accumulations of convolveFIRblock: 517268

----------------------------------------------------------------
Multiplications of downsample: 0
Accumulations of downsample: 10240

----------------------------------------------------------------
Multiplications of downsample: 0
Accumulations of downsample: 10240

----------------------------------------------------------------
Multiplications of fmDemodArctan: 15360
Accumulations of fmDemodArctan: 40960

----------------------------------------------------------------
POPPED
Multiplications of convolveFIRblock: 51200
Accumulations of convolveFIRblock: 103692

----------------------------------------------------------------
Multiplications of downsample: 0
Accumulations of downsample: 2048

----------------------------------------------------------------
```

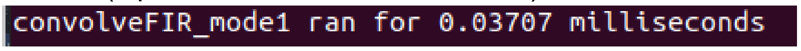
Operations per audio sample

	Mono Mode 0	Mono Mode 1	Stereo Mode 0	Stereo Mode 1
Multiplications	283	550	888	1714
Accumulations	587	1138	1768	3404
Atan2	0	0	3	5
Sin	0	0	3	5
Cos	0	0	5	9

Mono + Stereo (mode 0)  `Program ran for: 18297 milliseconds`	Mono + Stereo(mode 1)  `Program ran for: 16169.4 milliseconds`
Mono (mode 0)  `Program ran for: 17929 milliseconds`	Mono(mode 1)  `Program ran for: 15810.7 milliseconds`

Runtime of stereo is slightly greater than that of mono processing by 2.05% in mode 0. Similarly, the runtime of stereo in mode 1 is greater than that of mono by 2.27%. From the above measurements, we can calculate the mono processing runtime for each block in mode 0 and mode 1 are 8.75ms and 14.9ms, respectively.

Mode 0(Standard Convolution Function)	Mode 1 (Standard Convolution Function)
 `convolveFIR ran for 0.026331 milliseconds`	 `convolveFIR ran for 0.914578 milliseconds`

Mode 1 (Optimized Convolution Function)
 `convolveFIR_mode1 ran for 0.03707 milliseconds`

The standard convolution function ran for much longer in mode 1 due to up-sampling. Runtime was drastically improved by using an optimized convolution function that only convolutes the non-zero data.

Proposal for improvement

Considering the current status of our project, a key feature that we could include for a better user experience would be writing a text-based UI. As of right now, we need to type in commands in the terminal in order to run the program. This method is acceptable in the development stage because we the programmers are quite familiar with how to test the program with command lines. However, someone without a programming background or familiarity with this particular project would probably have no clue how to run the program. This is why we need to create a UI to make the program user-friendly for everyone. We are aware of the fact that GUI programming is incredibly complicated and difficult, which is why making a text-based UI would be a lot more feasible.

One feature that we could add to better our own productivity when improving the project would be putting everything in its own file instead of stuffing everything in the main function. We might not need this feature for this particular project, since the amount of code is still somewhat manageable. As we expand it to a bigger project, however, we believe this feature will drastically increase the productivity of testing and debugging. The main function should only be used to execute the functions, which means features like RF frontend or audio processing should be in separate files outside of the experiment.cpp. This has proved to be a useful and feasible feature because we have already implemented it for some of the functions. Because we put stereo processing and RF front-end in their own files, we were able to know which exact file was causing the error when debugging. This would have been a lot more complicated had we put them all in the main function. We could do the same to the rest of the project such as RDS to make merging easier on Git and improve the productivity in the debugging process.

Runtime performance of our system could be improved by using better techniques in threading. We could pass pointers of the RF demodulated blocks into and out of the synchronization queue instead of the actual blocks of data. Having only one synchronization queue instead of two when we run RDS and stereo processings concurrently would also reduce overruns. In order to achieve this, we would need extra conditional variables to check whether both threads are done with the particular block. Popping would only occur when both threads are finished reading from the block.

Project activity

week 1	-getting familiar with the project(all four members)
week 2	-going back to previous labs to fix block processing(all four members)
week 3	-getting started on RF-frontend and mode 0 of mono processing(Laura) -finished python model for mode 0(Kevin Chen) -hardware setup and raw file collection(Kevin Li) -Validated raw file correctness with given Python code (Linda)
week 4	-Live-testing (Kevin Li) -finished debugging for mono processing in mode 0(Laura) -getting started in mode 1 for mono processing(Kevin Chen) -writing python model for stereo processing(Laura and Kevin Chen) -optimization of RF-frontend(Linda and Laura) -getting started and debugging threading for RF-frontend and mono processing(Linda and Kevin Li)

week 5	-finished debugging mode 0 for stereo processing(Laura) -debugged mode 1 and optimization mono processing(Kevin Chen) -started and debugging of new convolution function(Kevin Li and Linda) -getting started in mode 1 for stereo processing(Kevin Chen) -finished debugging threading for RF-frontend and mono-stereo processing(Linda) -getting started on RDS(Laura)
week 6	-getting started and debugging RDS threading(Linda) -finished new convolution function(Kevin Li and Linda) -debugging RDS(Laura) -optimization of stereo processing(Kevin Li and Linda) -validation and optimization of stereo processing(Laura and Kevin Chen)

Conclusion

Before this course, we all have very limited knowledge of building a computing system interfaced with the real world. During this project, we have consolidated our knowledge on signal processing and we learned how to implement this knowledge in the real world. Besides, we learned many new things about signal processing such as the RRC filter and PLL. We have also developed our programming skills in both C++ and python. For example, we learned how to use the makefile, how to implement multithreading, and how to use the Gnuplot. We also learned more about the design process for the project. We are surprised that python models can help a lot in debugging and troubleshooting, and what significant role that optimization has. This complex team-based project developed our team working skills as well as the analysis and problem-solving skills. This experience will help us to accomplish higher goals and be more successful engineers in the future. This project also helps us to find what areas and fields each of us wants to keep developing in the future.

Please note. RDS with threading was originally implemented but it did not function as intended. This is due to the interference of the synchronization queue where if stereo popped out a block, then RDS could not access it. Therefore two synchronization queues were planned to be implemented but due to a lack of time could not be completed. Therefore, the changes were reverted and to run RDS please comment out all the code above and comment in all the code below.

References

1. Github Code. https://github.com/3dy4-2021/project-group41-thursday. Accessed: 2021-04
2. Project Document. https://avenue.cllmcmaster.ca/d2l/le/content/343598/viewContent/3090007/View. Accessed: 2021-04
3. Course Website. https://avenue.cllmcmaster.ca/d2l/home/343598. Accessed: 2021-04
4. RF Front-end & Mono Process. https://avenue.cllmcmaster.ca/d2l/le/content/343598/viewContent/3092547/View. Accessed: 2021-04
5. fmPLL. https://avenue.cllmcmaster.ca/d2l/le/news/343598/562691/view?ou=343598. Accessed: 2021-04
6. Threadings. https://avenue.cllmcmaster.ca/d2l/le/content/343598/viewContent/3121612/View. Accessed: 2021-04
7. RRC. https://avenue.cllmcmaster.ca/d2l/le/news/343598/566007/view. Accessed: 2021-04
8. RawRecordingResample.https://avenue.cllmcmaster.ca/d2l/le/news/343598/566005/view?ou=343598. Accessed: 2021-04