COE3DY4 – Project Report

Group 51
Vito Xie
Evan Liang
Yuvraj Bal
Thomas Yoo
xiev1@mcmaster.ca
liange7@mcmaster.ca
baly@mcmaster.ca
yoot@mcmaster.ca
April 8th, 2022

Introduction

The goal of this project is to create a software-defined radio(SDR) system for real-time reception of frequency modulated(FM) mono and stereo audio. Fundamental concepts of signal processing are used to process the FM data which is played live while new data is being received and processed. Radio frequency(RF) dongle based on the Realtek RTL2832U chipset and Raspberry Pi 4 are used for real time reception of frequency.

Project Overview

This project combines a Radio Frequency dongle and a Raspberry Pi 4 for the implementation of a Software Defined Radio system for real-time reception of frequency-modulated mono/stereo audio, as well as the reception of digital data sent through radio data system protocol. This was accomplished using software written in C++ compiled for the Raspberry Pi 4 along with some bash commands and other utilities for reading data from the RF dongle and playing audio from the device.

The software defined radio receives real-time FM data and digital data sent through radio system protocol. The signal flow graph of the FM receiver has four major parts: RF front end, mono path, stereo path and RDS. RF front end acquires the RF signal and translates it into 8 bit I/Q samples. It extracts the FM demodulated signal using a low pass filtering on both I and Q channels, decimation to reduce the sample rate followed by fm demodulation. The demodulated signal is sent to the three paths(Mono,audio,Stereo). FM channel bandwidth is 100 Khz. The mono subchannel is from 0-15 Khz, stereo subchannel is from 23-53 KHz and the RDS subchannel is from 54-60 KHz. In accordance with Nyquist theorem a 100 KHz low pass filter is used to extract the required channel. Finite impulse response(FIR) filter has its own filter

coefficients depending on the cutoff frequency, sampling rate, number of taps. For mono processing the filter coefficients are different because the cutoff frequency is 16 Khz to extract the mono audio. Data is filtered in blocks for faster computation. Demodulation is carried out using the derivatives of I and Q which is faster compared to arctan.

Sub channels use the demodulated data at the intermediate frequency which depends on the type of mode used. The SDR extracts the mono audio channel in 0-15 KHz of the FM spectrum. After passing through a low pass filter, for modes 0 and 1, the data is decimated to get the required sampling frequency at the output. However for modes 2 and 3 , a resampler is used because demodulated data needs to be upsampled to get the exact audio sampling frequency at the output after decimation. Phase-locked loops(PLLs) are phase tracking devices used to produce a clean output. In the stereo carrier recovery PLL is used to extract the 19 KHz stereo pilot tone.

Implementation Details

Prior to the project, some components were already completed as parts of the labs from earlier on in the course, such as impulse response generation and digital filtering . An implementation of the Mono Path in mode 0 was already complete in Python along with some C++ helper functions for filtering.

There are a total of 4 different modes of operation that each have their own custom front end, intermediate frequency, and audio sample rates. The mode can be chosen by including a command argument, with no arguments indicating a default mode of 0.

Working through the project involved first translating the existing python code into C++ code, and then concurrently developing the other modes for the Mono implementation, some components of the Stereo Path, and designing the software threading.

RF Front End:

In the RF Front End-block, an input is taken in the form of a .raw file, either directly from the raspberry pi, or from a previously given file in order to test the functionality in non-real time on a virtual machine. The input would have to go through a low-pass filter, a decimator, then an FM Demodulator, where the output would then go on to be passed as an input to the other paths. In order to implement this, we used functions that we have previously created in prior labs as the base of our design. Impulse response generation and digital filtering through convolution functions were already created, so we imported them from our labs. When the decimator was first designed, we did all the computations but only stored the necessary ones, which was acceptable for testing, but not efficient enough for real-time. This led us to alter our convolution function to include the decimator functionality, which meant that only necessary computations were done, instead of all of them. The FM Demodulator function was implemented in Python in lab 3, so it had to be translated in C++ code. In order to validate that the RF Front end was done correctly, we plotted a graph using estimatePSD, and compared the result to the graph that was created in lab 3. Another issue that we encountered was that the implementation

we used to split the audio into I and Q samples were incorrect, causing all our values to also be incorrect. We eventually traced the problem back and created a new working implementation.

Mono Path:

For the implementation of the Mono path, the python code from fmMonoBlock.py in lab 3 was translated into C++, as the result should essentially be mode 0. For mode 0, only a low-pass filter and decimator have to be implemented, which can be taken from the same function that was used in the RF front end block, as the only difference are the input values and the amount of decimation. The implementation of this mode should theoretically take very little time considering the resources we already had, but due to numerous small errors created while translating from python to C++, the process took longer than expected. One such error includes wrong sizing of arrays in C++ such as the block size vector, which was checked by comparing the sizes of the code to what we know the sizes should be. Mode 1 had different sample rates, but the code we had developed so far still worked due to the nature of how the sample rates could still be divisible into whole numbers. Modes 2 and 3 however were different, and required a new resampler to be created that has a much faster implementation than how it was done in modes 0 and 1. A resampler has to be created due to the division of the intermediate frequency and the audio frequency sample rates not being a whole number, therefore the sample rate must first be increased using an expander, then pass through a low-pass filter, then be decimated so that the end result is the desired frequency. However, this process must be done efficiently so that the real-time implementation works. This was done using a resampler function that does the upsampling, convolution and decimation functionality all at once, but it was done while skipping the intermediate steps, allowing the system to be within the real-time constraints. As shown in the analysis and measurements section, the multiplication and accumulations for the mode 2 are less than mode 0, thus mode 2 runs faster than mode 0, due to the fact that the audio sampling frequency is 44.1 KSamples/sec for mode 2 and 48 KSamples/sec for mode 0.

Stereo Path:

For the implementation of the Stereo path, we tried to implement it directly in C++ and skip modeling it in Python. We initially decided to skip modeling in Python due to us thinking that we do not have sufficient time to feasibly model in Python and later reformat it into C++ code, but after reflecting upon this decision, we realized that we should still have modeled it in Python first, which would make the implementation in C++ much easier if we knew that the Python logic worked.

In the stereo path, the FM Demodulated data initially goes through two paths, the Stereo carrier recovery and Stereo channel extraction. In the Stereo Carrier Extraction path, the input data is passed through a band-pass filter and the output is then ready to be used. Simultaneously, the Stereo carrier recovery path also starts with a band-pass filter, which then passes through a Phase Locked Loop and Numerically Controlled Oscillator. The PLL code was given to us in Python, which we had to refactor into C++ and add state saving. The output of the Stereo carrier recovery is then mixed together with the stereo channel by first performing pointwise multiplication on them. The data then has to go through digital filtering similar to the Mono path, which can be done using our convolution function. After the digital filtering, the stereo data is

then combined with the mono audio data by adding/subtracting the two, to get the left and right audio channels. Our stereo path did not work as we did not have enough time to complete it, especially without any guidance when we ran into conceptual problems.

Threading:

To enable our application to be threaded, our software was separated into 4 main functions:

- Main function
- RF Thread
- Audio Thread
- RDS Thread

Our main function processed command line arguments and initialized values to be used throughout the program. It would then call the RF, Audio, and RDS functions in separate threads and allow them to run indefinitely. Initially in order to allow both consumer threads to access the demodulated data, a class was designed to store the data and implement functionality that would allow access to the same data from a single queue. The queue would only pop the data from the queue once the object indicated that both threads had accessed it. However, to provide a more simple implementation, 2 separate queues were used to store the data where each thread would access its own independent copy of the data. The RF thread would process the input data and once demodulated would send 2 copies of the data to separate queues. The Audio and RDS threads would read data separately from their respective queues. The queues were locked individually when any thread was reading or writing to the queues.

All 3 thread functions consisted of 2 sections, a section for one-time initialization of data structures and other components such as filters and PLLs, and a looping main section that just performed the computations on an individual block. As the other components of the software were completed, design decisions needed to be made regarding which parts of the functions were to be placed in which section. This was important for optimizing the code to ensure time was not wasted on repeated computations of unchanging components such as filters. However it had to be ensured that all of the computations that must be performed on each block remain in the looped section to ensure correctness.

Analysis and Measurements

Mono Path Analysis

	Total Multiplications	Total Accumulations	Total non linear operations
Mode 0	2245919	1137866	202
Mode 1	2197791	1120628	202
Mode 2	2340654	1144208	14948
Mode 3	2504042	1150147	44642

*Assuming the default of 101 filter taps and for a single block of data

Mono Path Runtime Measurements

Mode	Code Section	Runtime on local host (ms)	Runtime on Raspberry Pi (ms)
Mode 0	Filtering and Decimating I values	0.672969	5.10952
	Filtering and Decimating Q values	0.673335	5.33404
	FM Demodulation	0.022959	0.081647
	Filtering and Decimating Demodulated Data	0.316763	0.967861
	Total Time for Sections	1.686026	11.493068
Mode 1	Filtering and Decimating I values	0.662115	5.15736
	Filtering and Decimating Q values	0.65852	5.0622
	FM Demodulation	0.020399	0.076851
	Filtering and Decimating Demodulated Data	0.256053	0.807826
	Total Time for Sections	1.597087	11.104237
Mode 2	Filtering and Decimating I values	0.683016	5.11209
	Filtering and Decimating Q values	0.726385	5.12011
	FM Demodulation	0.021323	0.078389
	Resampling and Filtering Demodulated Data	0.287726	3.19307
	Total Time for Sections	1.71845	13.503659
Mode 3	Filtering and Decimating I values	0.722772	5.11605
	Filtering and Decimating Q values	0.716843	5.08937
	FM Demodulation	0.22939	0.077203
	Resampling and Filtering Demodulated Data	0.203773	2.79763

	Total Time for Sections	1.872778	13.080253

*Recorded I/Q samples for close to 5 seconds on the Raspberry Pi (5.36s, 5.12s, 5.36s, 5.29s) for better comparison as the raw file used for measurements on local host is approx. 5 seconds

From the measurement tests, it appears that modes 2 and 3 are slower than modes 0 and 1, which goes against intuition because the audio output sample rates of modes 2 and 3 are less than those of modes 0 and 1, which means there should be less samples to process at the end. This could be a result of the use of the resampler for all modes which made modes 0 and 1 more efficient. Doing this results in a few unneeded multiplications for these modes but improves reusability of code.

Analytically, this does make sense because modes 2 and 3 have a far greater number of non-linear operations as well as a greater number of multiplications and accumulations, resulting in longer runtime. The majority of these operations take place when generating impulse coefficients which would be far greater in modes 2 and 3 because of the upsampled number of filter taps in their mono paths.

The total multiplications for all modes are similar in magnitude but the greatest relative difference comes from the mono path where modes 2 and 3 have a reduced number of multiplications while using the resampler relative to modes 0 and 1. The significant difference of multiplications in this path comes from generating coefficients and increasing their values by the large upsample rates in modes 2 and 3. This is expected because a greatly increased number of coefficients are generated in the code and this operation is not written in a computationally optimized way like with the resampler. Ultimately, this is most likely why there is such a surprising result in runtime.

However, modes 2 and 3 are still able to run in real time on the Raspberry Pi, indicating that the runtime is sufficiently fast for the desired function.

Filter Tap Investigation

Impact of Taps on Signal Quality and Runtime in Mono Mode 0 on the Raspberry Pi

Taps	Filtering and Decimating I values	Filtering and Decimating Q values	FM Demodulation	Filtering and Decimating Demodulated Data
13	0.70086 ms	0.603232 ms	0.081481 ms	0.130535 ms
301	15.5644 ms	15.4788 ms	0.079165 ms	3.02474 ms

By changing the number of taps, it is seen that in this case a filter with 13 taps greatly decreases the overall runtime of the code and improves signal quality. When the filter uses 301 taps, the

runtime increases drastically in comparison and the audio cuts out from time to time. The same recorded I/Q samples are used for this comparison.

The difference in runtime is a result of the increased frequency resolution that allows for a steeper transition band and better attenuation, which increases calculation time. The increased number of taps increases the number of coefficients which increases the number of multiplications and accumulations, therefore making the runtime longer.The difference in audio quality is likely an unexpected side effect of aliasing.

Threading Analysis

```
Operating in default mode 0
Starting FE thread
starting audiothread
Playing raw data 'stdin' : Signed 16 bit Little Endian, Rate 48000 Hz, Mono
Audio thread starts at: 120.744 ms
RF Thread starts at: 120.97 ms
Audio thread ends at: 121.357 ms
RF Thread ends at: 125.7 ms
```

To prove the functionality of our threading, we took time stamps in the separate threads synchronized to the same start point defined in the main function. The time stamps indicate the start and end time of the processing of a block by the separate threads and define a time interval which they were operating on. Since the 2 intervals defined by the start and end times overlap, it proves that both threads were running concurrently.

Code snippets for timing. start_time, defined in the main function, is passed into rfFrontEnd and monoStereo as parameters.

```
int main(int argc, char* argv[])
{

    auto start_time = std::chrono::high_resolution_clock::now();

    int mode = 0;
    if(argc < 2){…
```

```
if(block_id == 21){

    auto stop_time = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double,std::milli> process_run_time = stop_time - start_time;
    std::cerr << "RF Thread starts at: " << process_run_time.count() << " ms" << "\n";

    convolveFIRinBlocks(I, I_block, h, i_state, BLOCK_SIZE/2, rf_decim);
    convolveFIRinBlocks(Q, Q_block, h, q_state, BLOCK_SIZE/2, rf_decim);
    fm_demod.clear();fm_demod.resize(I.size(),0.0);
    demod(fm_demod,I,Q,prev_state);

    auto stop_time1 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double,std::milli> process_run_time1 = stop_time1 - start_time;
    std::cerr << "RF Thread ends at: " << process_run_time1.count() << " ms" << "\n";

}
```

```
if(measureCounter == 21){

    auto stop_time = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double,std::milli> process_run_time = stop_time - start_time;
    std::cerr << "Audio thread starts at: " << process_run_time.count() << " ms" << "\n";

    resampler(mono_data, fm_demod, h2, state, audio_decim, US);

    auto stop_time2 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double,std::milli> process_run_time2 = stop_time2 - start_time;
    std::cerr << "Audio thread ends at: " << process_run_time2.count() << " ms" << "\n";

}
```

Proposal for Improvement

Extra Features:

Improving the software could start with improving the usability of the software. Currently, to run the software, you must use a bash script to pipe the input from a file into the standard character input of the main program. To improve this, a graphic user interface front end could be developed, or even in a simpler fashion, an executable bash script could be designed to make the experience easier. Providing the user with a prompt explaining what arguments to provide the program along with removing the need to know the commands to pipe the input file into the program would improve usability.

In addition to this, adding functionality that allowed the user to record radio information, pipe it into the program, and have the program run all at once could improve our productivity and provide a more complete product. This is technically feasible as other Software Defined Radios already exhibit this functionality, and could likely be accomplished using a bash script that combines the two separate commands we already use for separately recording and data and running our program. This functionality would improve our productivity by making it easier to sort our samples and ensure that input and output sample rates match every time.

Better Performance:

While the Raspberry Pi 4 has 4 physical cores and the ability to run 4 hardware threads, our current implementation of the program only utilizes 3 main threads. Only running 3 threads concurrently is inherently less efficient than some implementation that makes use of all 4 cores. However, the actual performance of the program is limited by the input and output sample rates, where the software cannot process more data than is required, and if it runs without any breaks

then there would be no more room for improvement through the use of an additional hardware thread. This however may become beneficial when running our program on lesser hardware where performance may be an issue in its current state.

Project Activity

	Progress	Evan	Thomas	Vito	Yuvraj
Week of Feb 14	Project Spec Released				
Week of Feb 21	Reading Week				
Week of Feb 28					
Week of Mar 7	Read and understand the project specifications	Read through project document	Added existing components from labs to project	Reviewed project document and previous labs	Read through project document
Week of Mar 14	Refactoring of lab 3 code, block processing	Worked on efficient front end filter and front end debugging	Worked on mono path	Worked/Debugged RF frontend	Debugging R/F frontend
Week of Mar 21	Mono mode 0 complete	Debugging Mono 0/1	Started threading, single queue version	Worked/Debugged mono 0/1	Upsampler for mode 2,3
Week of Mar 28	Stereo C++ Model, fast resampler for real-time	Improved Resampler, debugging Mono 2/3	Switched threading to double queue version. Merged branches, optimized code	Started Stereo path, debugged mono 2/3	Fast Resampler
Week of Apr 4	Report Complete	Worked on Report, Runtime Measurement	Worked on Report	Worked on report	Worked on report

		s			

Conclusion

Throughout the project we consolidated knowledge on signal processing such as convolution, upsampling, downsampling and FIR filters. Python was used to model the SDR to check correctness. Refactoring the code to C++, debugging and optimizing code to run in real-time increased our technical and problem solving skills.

References

[1]Cpp Reference, https://en.cppreference.com/w/ , Accessed: 2022-04

[2]N. Nicolici, Class Slides, Accessed: 2022-04

[3]N. Nicolici, “3DY4-project-2022” , Accessed: 2022-04