COMPENG 3DY4 Project Report

Group 12 (L02)

Alisher Rakhimov (rakhimoa, 400311502) // Olukemi Odujinrin (odujinro, 400317700) // Taiwo Rabiu (400331895) // Antheus Aldridge (aldria1, 400339569)

rakhimoa@mcmaster.ca // odujinro@mcmaster.ca // trabiu@mcmaster.ca // aldria1@mcmaster.ca

Submitted: April 10, 2023

Introduction

The main objective of the project is to navigate a complex (industry-level) specification and understand the challenges that must be addressed for a real-time implementation of a computing system that operates in a form factor-constrained environment [1]. The project covers the application of communication concepts in computer engineering design and the practical set up of a software-implemented radio receiver, expected to run in real time.

Project Overview

In this project, radio-frequency (RF) hardware and single-board computers such as the Raspberry Pi 4 are used to implement a software-defined radio (SDR) system programmed in Python and C++ for real-time frequency-modulated (FM) broadcasting for mono and stereo audio and digital data transmitted through FM broadcast using the radio data system (RDS) protocol.

A third-party developed software interface of the RF hardware such as an RF dongle is used to acquire the RF signal through an antenna and is translated into the digital domain by generating an 8-bit sample for the in-phase (I) component and another 8-bit sample for the quadrature (Q) component. The RF front-end is a programmable block from the SDR system that extracts the FM channel using a low-pass (finite impulse response) filter and obtains the intermediate frequency (IF) of the signal through decimation and FM demodulation.

Once FM demodulated data at the expected IF sampling rate is extracted, the SDR system can follow one of three paths. First is the mono path which extracts the mono channel that is between 0 to 15KHz on the FM channel spectrum. The mono path outputs the sum of the left and right audio channels found in the stereo path. The second is the stereo path. This path has a different approach as it extracts the 19KHz pilot tone using a band-pass filter. The pilot tone is synchronized to the extracted stereo channel using a phase-locked loop (PLL) and is put through a mixer with the stereo channel. Stereo down conversion and filtering are applied and finally combined with the mono audio data to produce separated left and right audio channels. Finally, the RDS Path recovers the subcarrier from the RDS channel, which is down-converted using a digital communication approach and is resampled before performing clock and data recovery [1]. The concept of frame synchronization is done here before being able to pass the collected information words to the RDS application layer.

The project overview of extracting the RF signal and recovering either mono/stereo audio or RDS data is achievable through digital signal processing (DSP) with the use of core and basic building blocks such as finite impulse response filters, re-samplers, and PLLs. Low pass, bandpass, and all-pass FIRs can be implemented using the convolution of the filter's impulse response with a discrete time sequence of digital block samples as the input. Re-samplers are implemented by up and down sampling the digital block samples to obtain desired IF and audio frequency (AF) sampling rates depending on the RF sampling rate provided. The purpose of PLLs is to produce a clean output and essentially lock the phase of a periodic and noisy input signal that is then passed to a numerically controlled oscillator (NCO) for scaling. These very important building blocks are chained in a signal-flow graph of the SDR system to implement an FM receiver using software.

Implementation Details

Labs & Building Blocks

The purpose of the labs was to gain some valuable knowledge about the basic building blocks that would be useful in the project. The goal of Lab 1 was to develop an understanding of some basic digital signal processing (DSP) primitives used in Software Defined Radios (SDRs) and implement them in Python [2]. A function from the scipy signal toolbox to easily generate filter impulse responses and use them to filter signals. We would then implement our own LPF impulse response generation function and convolution function that could handle block processing. Block processing was a technique used as it required data to be processed in real time because not all of the data was available to process at once. Block processing support was added in the form of state saving for the convolution function which meant it needed to store the last N_taps elements of the input signal to be used in the next block as the convolution function required previous input data points to compute the current data point.

The goal of Lab 2 was to provide a link between the modeling of (DSP) primitives used in Software Defined Radios (SDRs) in an interpreted and scripting language like Python to their implementation in C++ [3]. This was important because the project needs to satisfy a real-time constraint on limited hardware so choosing a compiled low-level language like C++ was important. The main challenge of converting lab 1 code into C++ was mostly just being mindful of storing and accessing elements in a vector because there are no built-in protections against improper memory access. Another important aspect of C++ was passing by reference rather than returning large vectors of data because this ensures no extra memory is needed to copy large vectors.

Then for Lab 3, we needed to understand how to use signal-flow graphs to model a real-life radio application in software [4]. The building blocks from the previous labs were used to implement the mono mode 0 path in Python. However, a few new functions such as fm_demod() were needed because the functions provided were computationally inefficient as the arctan function is computationally heavy. Optimizing the fm_demod() function was written with an alternate mathematical formula that used simple addition, subtraction, multiplication, and division.

RF Front-end

At the beginning of signal-flow graph, following the RF hardware block (already implemented by the RF dongle) is the RF front-end block. At this stage, the basic implementation consisted of the building blocks from labs 1-3 such as the convolution and FM demodulation and obtaining the IQ samples. IQ samples are extracted and processed in blocks as this technique is needed to avoid both an excessive latency for data acquisition as well as large memory requirements [1]. Vectors were the prime data types in C++ during this project as they were used to store the blocks of RF samples, create filter coefficients to perform convolution, state saving, and processing of these samples. Implemented functions for the RF front-end block include impluseResponeLPF(), readStdinBlockData(), convolveFIR(), and fmDemod(). Both the I and Q samples must pass through their own low pass filters and be decimated and recombined to obtain the FM demodulated signal as shown in Figure 3 below.

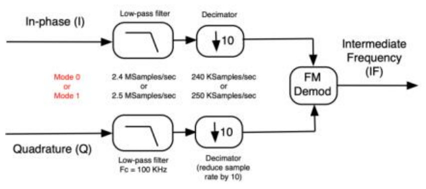
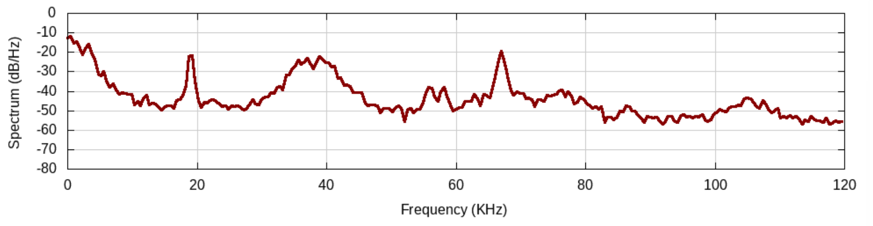

In-phase (I)
Low-pass filter
Decimator
10
Intermediate Frequency (IF)
Mode 0 or Mode 1
2.4 MSamples/sec or 2.5 MSamples/sec
240 KSamples/sec or 250 KSamples/sec
FM Demod
10
Quadrature (Q)
Low-pass filter Fc = 100 KHz
Decimator (reduce sample rate by 10)

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

impulseResponseLPF() function generated the filter coefficients, a sinc function in the time domain but a rectangle window in the frequency domain that only allows low frequencies to a certain cut-off (i.e rf_Fc). The design of the convolution was written in an efficient technique that applies decimation while convolution is being applied. This saved memory and computational time. This was the implementation of downsampling and later in the mono & stereo blocks, resampling was implemented with the same technique. The plot_signals() function was written to help visualize each stage of the signal-flow graph and it served as Group 12’s validation method for the entire project. For example, the output of fmDemod() was plotted as shown below.



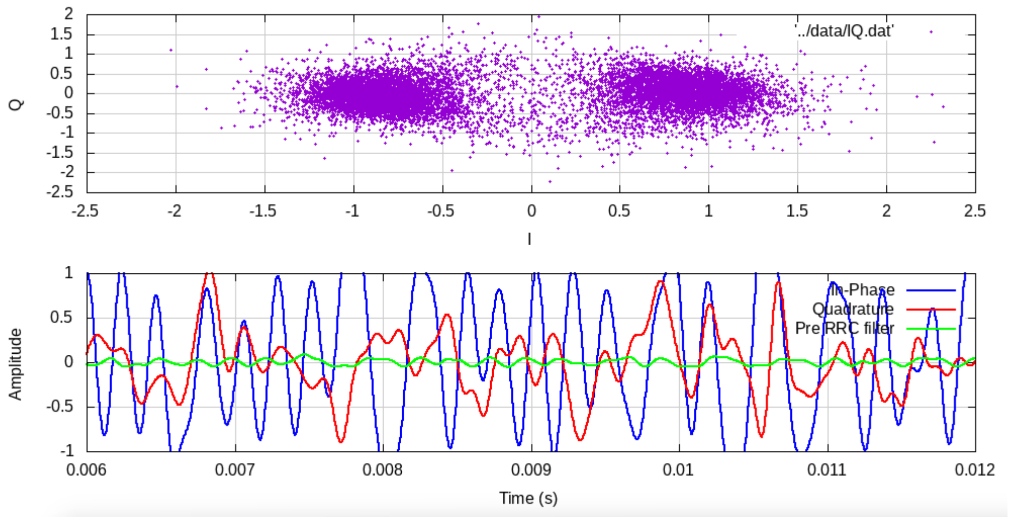
Mono & Stereo

The entire mono path had already been completed in Python for lab 3 which meant the first step was to convert lab 3 code into the C++ project. Other than the front end which was discussed above, the next step was to extract the mono channel, down sample, and pipe out audio data to aplay. In lab 3, down sampling was done after convolution but here the process needed to be optimized by down sampling within the convolution itself, and this was done simply by sliding the impulse response during convolution by the decimation factor instead of by 1 sample each time. This setup was not very difficult to implement but it would only suffice for mode 0 where no up-sampling was required. Other modes required down sampling by non-integer factors. This meant the data needed to be up sampled and then down sampled to achieve the specific resampled frequency. There are up and down sampling factors that are needed to be calculated based on the given and target sampling rates. For example, Group 12 had an IF of 240Ksamples/sec and must obtain a signal at AF = 44.1Ksamples/sec. Since these 2 values aren’t easily divisible, resampling must be applied. Applying some basic math, the up sampling factor ends up being 147 and the down sampling factor is 800. This is accurate because multiplying 240K by 147 then dividing be 800 obtains 44.1K, the desired AF sampling rate. Once 3 modes for mono were working, then the stereo processing path may begin. The stereo path would re-use many of the tools we used previously but just required some new conceptual knowledge to be able to use those tools as needed. The main new function needed was something to generate an impulse response for a bandpass filter where the pseudocode was given so implementation was straightforward. The bandpass filter was needed to extract the DSB-SC modulated stereo channel and the stereo carrier. To demodulate the stereo channel, multiplication with the carrier for the stereo channel is needed to split and for one image to be brought to the baseband and the other image to be filtered out. Even though the carrier is extracted, a PLL is needed to clean up the carrier signal as the extracted signal has noise. The PLL would generate a perfect sinusoid needed to lock on to the phase of the carrier. Since the PLL was a function with feedback, it needed state saving so that there would be continuity between blocks and would maintain its phase lock. To do this, a vector is used to store the different PLL variables that were part of the feedback loop so that data was available for the next block to use. Once a clean signal was obtained, all that was left was to demodulate the stereo channel through pointwise multiplication. Since this demodulation technique produces an image at 2 times the carrier frequency, a filter was needed. Also now that the signal is in the baseband, the frequencies are low enough to convolve and resample down to the output audio sampling rate. The last stage required mixing mono and stereo to produce the individual left and right channels which were done through the addition or subtraction of the 2 channels.

RDS

The start to RDS is quite similar to the mono and stereo paths as it starts with a bandpass filter to extract the RDS channel. The RDS channel is located from 54kHz to 60kHz and is filtered the same way filtered signals are represented in the above sections but without the resampling. Once filtered, the signal needs to be demodulated using DSB-SC techniques like stereo. However, because there is no carrier pilot tone, it must be extracted from within the RDS channel itself by first doing pointwise multiplication with itself. Next, another BPF is used to extract the 114kHz RDS carrier sinusoid which again isn’t resampled because 114kHz is approaching the Nyquist sampling limit 2*114kHz < 240kHz. This carrier then needs to be cleaned up using the PLL and scaled down by 2 as the RDS center frequency is 57kHz. All of these steps have already been seen in the previous paths so implementing this section was fairly straightforward as existing functions previously made were simply called without having to do anything fundamentally new. However, a new all-pass filter was needed to delay the original extracted RDS channel because the RDS carrier gets delayed by the bandpass filter by N_taps number of samples. The impulse response was quite simple as it is just a single impulse time shifted by N_taps/2 and then the channel was convolved the same way as before. Once the channel was DSB-SC demodulated using a mixer it had to be passed through a LPF to remove the image at 2*fc. This LPF would also now resample the signal so that the proper number of samples per symbol or SPS is acquired.

After this stage, there was more difficulty in implementation as brand-new concepts and functions had to be created. The first stage after resampling was the root-raised cosine filter which had to be adapted from the provided Python code There weren’t any issues as it was mostly having the proper math equations. After filtering the signal, next was clock and data recovery, and originally this was implemented by finding the first peak of the first block by finding the largest absolute value within the first SPS samples. This method would prove problematic, and this will be discussed later in this section. Now that there was a starting sample, data collection can begin for IQ constellations. This would first require a quadrature output from the PLL which was done simply by creating a new PLL function that was functionally the same but returned the sine of the argument instead of the cosine. This new set of quadrature data would then follow the same path as the in-phase and use the same functions with the same parameters. Since each block did not contain a lot of symbols, a vector was used to accumulate samples across the runtime of the entire program. To visualize the data, gnuplot was used, and wrote I samples to x and Q samples to y. But it was important to visualize the in-phase and quadrature signals after being passed through the root-raised cosine filter so that the amount of power in each signal can be seen. Looking through the gnuplot documentation was helpful as we didn’t have experience plotting multiple signals on a plot or plotting scatter plots. Regardless, this was simple to find and implement by just changing the parameters in the gnuplot file. The figure below shows the plots during the final stages of implementation but at the start, the IQ constellation was quite off for a few reasons.

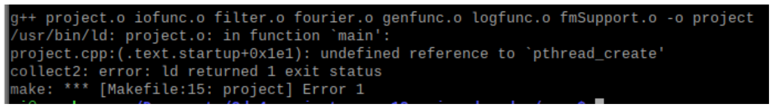


At first, there was a lot of overlap between the 2 clusters which made it look more like 1 long cluster of samples instead of 2 distinct clusters. This was because a poor algorithm was implemented for selecting the samples and it only consider the first set of samples which meant it would also deviate from the optimum as more data came in even if the sampling point was incremented by SPS each time. The new implementation would now take a look at the entire block every time to keep it synchronized. We made a function that would test all starting samples in a block (0-SPS) and then would take samples every SPS sample and record their distance from zero. These were then added together and averaged out to fund the average distance from 0. Whichever starting sample would produce the largest average distance from the origin would then be selected. This algorithm takes into account the possibility that the first symbol in the block was heavily interfered with or that it may lose synchronization over time. However, when the data was plotted, 2 distinct clusters were observed but still had the issue of being slightly rotated across the origin. This meant that the PLL had to be phase tuned and changing around the phase parameter until a desired plot that had the 2 clusters rotated in the orientation seen above was a way to do so. Once the proper samples were obtained, it was time to move to the decoding stage by converting the samples into binary and using Manchester decoding. This would look for the high-low and low-high transitions. Although, during lectures encountering LL or HH pairing was expected, they were invalid and meant a different starting symbol was needed to alleviate this issue. However, it is very difficult if not impossible to know if a LL or HH pairing occurred because it started on the wrong symbol or because that specific data bit was corrupted. Unfortunately, not enough time was available to have an efficient way to deal with this so the first bit was the start of sampling but if the program fails to synchronize in the application layer after a certain number of blocks have been processed, the first symbol can be discarded from that block and start sampling from there. After this was differential decoding and this was quite simple as it just required XORing the current bit with the previous bit. This decoding stage had state saving that was a bit different from the previous sections. When converting the samples into symbols there wouldn’t always be the same number of symbols per block and it needed to be checked if the number of symbols extracted was even or odd because when pairing them, there would be an extra unpaired symbol when an odd number of symbols was extracted. This was stored as an integer that would indicate no bit(-1) zero bit(0) or one bit(1) which would then be read during the processing of the next block and add or not add an additional bit accordingly. Then state saving was required for differential decoding as it required the previous bit for computations, and this was implemented similarly to functions such as fm_demod() or PLL.

Before implementing the main application layer, a function was needed to would check a set of 26 bits to see if those bits correspond to a block type. This was not very difficult as it required to perform matrix multiplication between the 26 input bits and the parity check matrix but instead of taking dot product through addition, it would instead be XORed. Then it would compare the output 10-bit parity with the corresponding expected parities of the block types. At first, frame synchronization was challenging to implement because the input bitstream for the particular block would be smaller than 26 bits (usually 25 or 24). This would require a very unique kind of state saving that would be difficult to implement. So instead, the RDS path would accumulate multiple blocks of data at a time before starting processing. Unfortunately, this is quite inefficient because when it is accumulating blocks the RDS path does nothing but when it is ready to start processing it has to process multiple blocks but there wasn't enough time to optimize this. To actually synchronize with the bitstream, checking windows of 26 bits and continually shifting by 1 until something was found was sufficient. When it synchronized with a block, it would shift the window by 26 bits each time because the next 26 bits very likely belonged to another block unless the previous one was a false positive. In case this does happen, it will just start over and try to re-synchronize. Then once synchronized the lower 16 bits is read for data but more specifically, the 16 data bits were looked at for block A, data bits 5-9 of block B and all 16 data bits of block D. Finally, now that bit information has been extracted, it must be converted into text for the user to comprehend. Each bit was represented as a boolean in a vector. The function binarytoString() took inputs of the PI code and Program Type and first converted them from a vector to a string of 1s and 0s. This made converting the information from bits easier. For the Program Type, 5 bits of 1s and 0s represented a type of music (i.e Classical, Jazz, Rock). Simply creating if statements to compare with the input generated the correct program type. For the PI code, the 16 bit string needed to be converted into hex code. Applying a small function that does this converts a binary code to hex code and produces the expected data output.

Multithreading

To efficiently facilitate the real-time execution of the SDR system, multithreading was implemented, sectioning the signal flow-graph into 3 main components. These components were implemented using threads in C++. First is the producer thread, which is the RF front-end block. The producer thread is used to push blocks of data (i.e IQ samples represented as vectors of floats) into a queue. The first consumer of the queue is the audio thread, and the second consumer is the RDS thread. These threads pop and process data one at a time so long there is a block to process in the queue. The mono & stereo blocks are grouped in the same thread as they depend on each other for their outputs and they both receive and use the same data in similar ways. The RDS thread applies a different implementation process compared to the audio thread and is expected to have its own thread. To implement the synchronization queue, the producer, consumers, a mutex, and conditional variables are used. In addition, atomic variables are used as tools for navigating the threads and timing their execution. Multithreading is a new concept in C++ and as expected there were debugging errors while implementing. Below is an error that occurred when pthread wasn’t defined in the project. This prevented the std::mutex, std::atomic, and std::condition_variable data types from functioning in the 3 threads. The solution involved updating the main header file to include pthread and multithreading was successfully implemented.



```
g++ project.o iofunc.o filter.o fourier.o genfunc.o logfunc.o fmSupport.o -o project
/usr/bin/ld: project.o: in function `main':
project.cpp:(.text.startup+0x1e1): undefined reference to `pthread_create'
collect2: error: ld returned 1 exit status
make: *** [Makefile:15: project] Error 1
```

Analysis and Measurements

We calculated the following values using the following block size:

*1024 * rf decimation factor * audio decimation factor * 2 /audio up sample factor*

And N_taps = 101

Processing path	Multiplications and accumulations per bit	Non-linear operations per bit
Mono 0	1111	0
Mono 1	1111	0
Mono 2	1200	0
Mono 3	1861	0
Stereo 0	2121 per LR pair (mono not included)	20
Stereo 1	2121	20
Stereo 2	2300	21.77
Stereo 3	3622	34.86
RDS 0	104 436.64	819.2
RDS 2	111 401.6	819.2

Blocks of Code				Running Times (ms)			
				Mode 0	Mode 1	Mode 2	Mode 3
IQ Samples	10.04	8.75	10.82	15.84			
FM Demod	0.0555	0.0593	0.05	0.09			
PLL	1.025	1.12	1.12	1.81			
Convolution	1.8	1.77	8.14	7.713			
RDS Decode	0.0045		0.013				
Frame Synchronization	1.084		0.209				

Path	Modes	Running Times (ms) (13 taps)	Running Times (ms) (301taps)
Mono Mode 0 & 1:1024 Mode 2 & 3:1023	Mode 0	2.226 (0.00217)	30.236 (0.0295)
	Mode 1	2.061 (0.002127)	29.94 (0.0292)
	Mode 2	3.035 (0.002967)	33.43 (0.0326)
	Mode 3	4.526 (0.00442)	48.80 (0.0477)
Stereo Mode 0 & 1:1024 Mode 2 & 3:1023	Mode 0	4.4 (0.00430)	55.80 (0.0545)
	Mode 1	4.5 (0.00439)	56.13 (0.0548)
	Mode 2	5.66 (0.0055)	69.2 (0.0676)
	Mode 3	8.24 (0.00805)	99.37 (0.0971)
RDS	Mode 0	4.6 (0.184)	69.75 (2.79)
	Mode 2	5.54 (0.2216)	93.64 (3.75)

*times in brackets represent runtime per sample/bit while number outside of brackets are runtimes per block

Total number of seconds for which the I/Q samples have been acquired: 0 (2.331s), 1 (0.984s) 2 (2.331s), 3 (1.145s)

In order to make mode 2 work, the block size had to be kept to the modified value of 6. This in result makes the running time large so to maintain consistency with the other modes the running time was divided by 6.

Observation at 13 N_{taps} – Audio was not clear and a bit of static existed. The RDS implementation was unfortunately unable to sync.

Observation at 301 N_{taps} – Audio was clearer, however the audio was too slow to process creating underrun. The RDS implementation was unfortunately unable to sync. It is expected that RDS should sync properly for these number of N_{taps} however a possible reason to the results of increasing the N_{taps} is that there is delay in the phase tunning stage and proper readjustment is required to accommodate for this delay.

Comparison between theoretical and experimental results:

For Mono and Stereo, the change in running times between modes were relatively similar between the theoretical calculations and experimental results in. For example, for mono, the theoretical results between modes changes from 1111 to 1111 to 1200 to 1861. These factor changes are 1, 1.08 and 1.55. For the experimental results, the modes change from 0.00217 to 0.002127 to 0.002967 to 0.00442. The factor changes are 0.98, 1.39 and 1.49. Although not perfect, theses ratio factors are closely similar and shows the implementation of the SDR system to be efficient. The large ratio between modes 0 & 1 to 2 & 3 are possible due to the difference between the AF sampling rates. More computational operations were being implemented on a smaller set of samples in modes 2 & 3 and is a possible explanation to why their running times are larger relative to modes 1 & 2.

For RDS, there were very similar runtimes per block with stereo but was slightly more computationally heavy which resulted in a slightly larger runtime. When normalizing all the results on a per bit/sample basis, a comparison of the results to the theoretical calculations can be applied. Since RDS produces so many fewer bits relative to the number of audio samples produced, it can be seen that the runtimes per bit grows significantly relative to the audio runtimes per sample. This was seen with the theoretical results as well with very similar relative changes between the different processing paths.

Proposal for Improvement

All systems aren't created perfectly, and all projects have room for improvement both user-wise and productivity-wise. A feature that would benefit the user of the SDR system, is implementing FM channel changing interface. During this project, FM 87.5 was the source for real-time FM demodulation. Generating a simple and small interface to switch to another radio channel such FM 99.5 can improve the overall user experience and make for a more efficient and powerful system. Another feature that would better the productivity of the project when expanding/validating/improving is creating separate branches on GitHub to work on different sections of the program and merge when completed. At times, during the project, certain members of the group wrote functions or blocks of code in a file outside of the repository to avoid conflict with properly working pieces of code and work done by other group members. The feature to create 4 branches for all members, keeps records of all code written by all members for quick recall and keeps the main branch intact. Only when the group agrees that the new changes positively progressed the project, will a push to the main branch be encouraged.

Optimization of the runtime performance of a system is always encouraged. The faster, the better, and for this project, the runtime was an important factor to optimize. Since the system is based on real-time data, the runtime of the SDR must be able to "keep up" with the data input. To further improve the runtime performance, placing more processes in parallel with others can be deemed useful and efficient. Multithreading was a key technique for improving the running time however, it does have its limitations. Creating functions that perform more than one task at once serves as a good solution to improving the running time. As mentioned previously, resampling is the combination of up-sampling and down-sampling but all at once. Instead of performing these two operations separately, simply doing them together saved a lot of computational power and reduces the running time. In similar ways, we can apply this technique, for example, by generating a filter coefficient as well as applying convolution to an input can be a place to apply such a technique.

The current stage of our project is work that has been put in for over a month. Given more time and resources, the features and improvements mentioned above can upgrade the functionality of the SDR and place the results of the project at an industry-level system.

Project Activity

Below is the project activity beginning after the midterm recess and the dates of the lab sections attended during that week. Work on the project was done outside lab and class hours as added contribution.

week	
1&2	**Alisher -** Completed lab 3 and Read over project documentation **Kemi –** Completed lab 3 and read over project documentation **Taiwo -** Completed lab 3 **Antheus -** Completed lab 3
3	**Alisher -** added support for piping in data and writing to aplay, implementation of mode 0 mono path **Kemi -** fmDemod py to c++ conversion **Taiwo -** fmDemod py to c++ conversion debugging **Antheus** – working on mono implementation
4	**Alisher -** added efficient resampling convolution function, added functionality to easily plot signals (and their PSD's) of certain blocks, extraction of stereo components(pre-demodulation) **Kemi -** fmPll py to c++ conversion + state saving, mono modes if statement implementation **Taiwo -** stereo left & right channel extraction, implemented bandpass filter **Antheus -** helped fmPll c++ conversion and state saving
5	**Alisher -** fixed pll + fm_demod state saving issues, helped with mixing, channel extraction and stereo writing out **Kemi -** mixing stereo channel & stereo carrier, mode implementation for writing channels (mono or stereo), debugging bandpass filter **Taiwo -** tested and debugged stereo audio with headphones to hear segmentation of channels **Antheus -** testing static in different modes, helped channel extraction, pll troubleshooting
6	**Alisher -** completed all rds demodulation up to RRC filter including carrier extraction, DSBSC demodulation, all-pass filter, constellation plots, phase tuning, resampling for proper SPS, and symbol extraction, added support for 2nd consumer thread(rds) **Kemi -** overall multithreading implementation **Taiwo -** helped with understanding multithreading concept **Antheus -** debugging stereo modes

7	**Alisher -** proper clock and data recovery (w/constellations), Manchester + differential decoding, frame synchronization with block detection and data extraction **Kemi -** focused on RF & audio multithreading, debugging mono & stereo static bugs, binary to string conversion function for PI code and PTY **Taiwo -** debugging mono & stereo modes **Antheus** – testing static in stereo and mono modes
8	**Alisher -** final presentation & cross examination, analysis and measurements (calculations and measurement code), project activity, implementation details(RDS, labs/building blocks, mono/stereo) **Kemi -** final presentation & cross examination, introduction, project overview, project activity, conclusion, implementation details(Multithreading, RF frontend, labs/building blocks) **Taiwo -** final presentation & cross examination, final report (proposal for improvement) **Antheus -** final presentation & cross examination

*note: *Alisher's commits were done using 2 github users named: "alisher3" and "Amandine-Malo"*

Conclusion

The 3DY4 project presented the opportunity to navigate an industry-level specification that utilizes radio-frequency (RF) hardware and single-board computers that were used to implement a software-defined radio (SDR) system for real-time implementation of a computing system that operates in a form factor-constrained environment. Overall, this project was a practical example of industry-level computer engineering projects, and it applies fundamental communications knowledge learned in communication system courses such as ELECENG 3TR4. The learning experience was positive and beneficial to partake in. Concepts about convolution, filters, FM signals, mono/stereo, and RDS were key to improving the real-world application system that relates to radios, frequency carriers, and more. What was equally appreciated about the experience was learning more tools that can be used in both Python and C++. Multithreading and parallelism were new programming concepts and they proved useful in this project. This experience showed that it is still a very powerful tool that should be explored and used in other real-world computer engineering design systems. Prior to the course, applications of Python and C++ were limited in terms of real-world interactions, however, this project experience bridged a way between software and real-time systems. In addition to the implementation of the project, the opportunity to collaborate in a group and present what had been accomplished and learned prepared each group member to represent the work that had been put forward and stand by it. Each group brought a unique skill, whether it was troubleshooting, problem-solving, software debugging, or hardware implementation, they all contributed to the completion and success of the project. To conclude, the project centered around real-time SDR for mono/stereo FM and RDS strongly covers communications systems, hardware, and software in a well-blended structure and is recommended for all electrical and computer engineering students.

References

[1] COMPENG 3DY4, "COE3DY4 Project", *GitHub Classroom* [Online].

[2] COMPENG 3DY4, "COE3DY4 Lab #1", *GitHub Classroom* [Online].

[3] COMPENG 3DY4, "COE3DY4 Lab #2", *GitHub Classroom* [Online].

[4] COMPENG 3DY4, “COE3DY4 Lab #3”, *GitHub Classroom* [Online].