COE3DY4 Project Report

Group 21 & 25
Boxi Liu 400232078 Ridvan Song 400208112
Enyu Liu 400226447 Yinwen Xu 400195279
Apr 6, 2022

Introduction

The following is a detailed document detailing the implementation details and performance analysis of a real-time software system built to interface with a USB radio device to create a software defined radio. The system is based on the Realtek RTL283U chipset, single-board computer Raspberry Pi 4 and the C++ implementation of a software-defined radio (SDR) system. It is designed to receive and process FM mono/stereo audio, the digital data sent through FM broadcast using the radio data system (RDS) protocol with real-time constraints.

Project overview

Our project is based on different blocks as explained below:

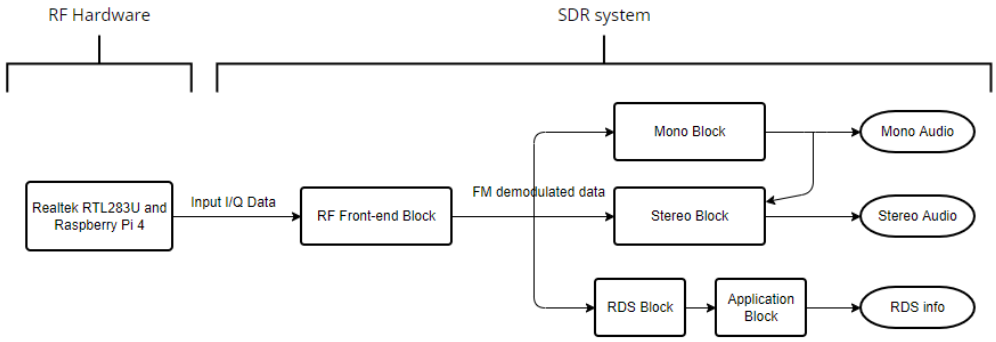

Figure 1: Block diagram of our real-time radio system.

Our real-time radio system consists of a Realtek RTL radio receiver device which can extract radio transmitted signals and convert them to a digitized quadrature FM signal to be directed to the stdout. Figure 1 shows the block diagram explaining the processing steps our software uses to create stereo audio data and RDS information from the received quadrature FM data.

The RF front-end Block first extracts I/Q data from the hardware output then implement the filtering and demodulation to get the FM demodulated data. This data is then used by the Mono audio thread, stereo audio thread, and RDS thread to produce stereo audio and RDS information that is displayed to the user.

The Mono block then extracts the mono channel analog audio (0 to 15kHz) from the FM demodulated data to create mono-channel audio. In practice, idealized filters cannot be implemented, hence finite-impulse response (FIR) filters that guarantee linear phase are commonly used.

The Stereo block extracts the stereo audio signal through band-pass filters and synchronizes the phase of the signal using a PLL on the 19kHz pilot signal. The data collected from the stereo channel is then combined with the mono audio data to produce the left and right audio channels. Phase-locked loop (PLL) are used to synchronize our signal which is achieved using a numerically controlled oscillator (NCO) which is included in the filter block.

Re-sampling including up-sampling and down-sampling methods helped us achieve the desired input and output sampling rates based on our specifications.

The RDS block first recovers the carrier through band-pass filtering then implement the RDS demodulation which involves root-raised cosine filtering and differential decoding based on Manchester decoding. After that the frame synchronization will be applied before passing data to the application block to be processed into application data. The application block will extract the information words based on the RDS protocol and provide data like the service name, service type, and PY codes of the certain radio channel.

Implementation Detail

The process of implementation is modelling by Python before translating to C++. This section is separated into different sub-sections. Since Python is the modelling language, some remarkable bugs we encountered in Python are noted as we go through each section.

Infrastructure

To make full use of the advantage of OOP languages (Python and C++), we create classes for RF front-end, mono, stereo, and RDS processing respectively. In each class, a method named "process" serves as the main function in this class, and frequencies and states are set as the attributes of the class. Creating classes can not only make the code much more readable and easier to debug, it also can hide implementation details such as the state saving information needed across multiple iterations from the main function.

The C++ infrastructure development is quite like the Python one (we won't go through it because its similar to the one below), we wrote different classes and modules in different files which can be referred by the main block (project.cpp). The name of blocks and their corresponding functionality illustrated as below:

Module Name	Functionality
Channel (module)	Provides functions used to implement different operations between two channels including addition, substraction, multiplication, mixing, upscaling.
Filter (module)	Provides functions for different kinds of filters including low-pass filter, band-pass filter, all-pass filter. The fmPLL functions of stereo and RDS processing are also included in this block.
Fourier (module)	Provides necessary functions related to the fourier implementation including DFT&IDFT, FFT, plotting tools etc.
genfunc (module)	Provides functions to generate sinusoid function or random signals, implement addiction between two signals.
Iofunc (module)	Provides functions related to the file I/O operation including printing data, reading data, writing data etc.
Logfunc (module)	Provides functions related to the vector operation.
Sampling (module)	Provides functions for re-sampling operations including up-sampling and down-sampling.
Rf_fe_block (class)	Extracts I/Q data from input data, down-samples them and then implement the FM demodulation.
mono_block (class)	Re-sampling the FM demodulated data and do the low-pass filtering.
stereo_block (class)	Implements the Stereo Carrier Recovery, Stereo Channel Extraction and Stereo Processing.
rds_block (class)	Implement the RDS Channel Extraction, RDS Carrier Recovery and RDS Demodulation, Manchester and differential decoding, Frame Synchronization and Error Detection.
Application (class)	Extracts the information words from the radio signal.
Performance (class)	Provides function to measure the execution time of program.
Project (main)	The main block controls the operation of our SDR system

Table 1: Module structure

Block Processing

Block processing is the key idea which enables real-time processing due to a small latency. A while loop is required in main.py or project.cpp to loop through all the blocks. States are needed for block processing, because a current block requires some latest data from the previous block. These states can be set as attributes of the classes, which will be automatically updated every time it is changed.

Re-sampling

For mode 0 and 1, only down-sampling is needed (i.e., up-sampling factor is 1), whereas for mode 2 and 3, resampling (up-sampling followed by down sampling) is needed. For up-sampling of factor U, zero padding is performed, with U-1 zeros in between two samples. For down-sampling of factor D, D-1 samples are removed after each sample, therefore, the useless D-1 samples can be skipped for calculation for higher efficiency. U and D are calculated such that the ratio of the input sample frequency to the output sample frequency is D/U and D and U are relatively prime. Note that a lowpass filter is needed in between the up-sampler and the down-sampler to filter out the aliasing images created by the up-sampler.

Filtering

There are four kinds of filters: low pass filter, band pass filter, all pass filter and RRC filter. Pseudocode is given for all these filters. However, it is important to note that the filter implementation can be optimized by not calculating the samples which are to be skipped by down-samplers, which saves the time to a large extent.

My_lfilter function is defined to perform the convolution, it can be used for low-pass, band-pass and RRC filtering. The up-sampling ratio and the down-sampling ratio are passed as inputs to the function and the steps of the for loops are these ratios so that useless calculations are avoided from being calculated. States are needed in the function as input and updated as output.

The all-pass filter can be implemented without convolution, as it simply acts as a phase shift, therefore, it is easiest to simple shift the vector by the desired delay.

RF Front-End Processing

8-bit unsigned integer I/Q samples from RF hardware are input to the RF front-end block. I/Q data are first separated from each other before passed through a lowpass filter, down-sampled by the decimator, and demodulated by the FM demodulator. Finally, the IF (intermediate frequency) data is generated which is to be used by mono/stereo/RDS processing. The demodulator is designed such that the arctan function is not used, which increases the efficiency of the program.

Mono Processing

Mono path is quite like the RF front-end path except that resampling is taken into consideration for mono path. Since some of the required resampling ratios were not integer multiples, we had to find the lowest common multiple of the resampling rate, implement an integer up-sampler, aliasing filter and integer down-sampler to achieve the correct sampling rate. We up-sampled our data by introducing zeros in between samples. This results in high frequency images that were produced in the input, research about this topic can be found at (1). We accounted for this by introducing an aliasing filter in conjunction with our mono audio extraction LPF to remove the images. After the filter, the sample was then down sampled to the desired frequency.

At this stage, some of the largest problems in our C++ implementation were related to the allocation of our vectors. It took some understanding to understand how to conserve CPU time by reducing re-instantiations of vectors and allocating capacity for all vectors before block processing.

Stereo Processing

For Stereo Processing, it has 3 sub-blocks, stereo channel extraction, stereo carrier recovery and stereo processing. Stereo channel extraction and resampling in stereo processing are like the mono processing. One thing we need to focus on is the stereo carrier recovery part, where the pilot tone is extracted by a bandpass filter before synchronized by PLL (Phase Locked Loop) and NCO (Numerically Controlled Oscillator). Since the PLL and NCO code is given in Python for single pass, we need to establish the states for block processing. We created a list of states including feedbackI, feedbackQ, integrator, phaseEst, trigOffset and ncoOut, which is passed as an input and returned by the function. The returned state is stored as an attribute of the stereo class object, which can be used by the next block of data.

The stereo part is combined with the mono part to generate the left channel and right channel. However, we encountered a problem that the mono audio and the stereo audio both sounded clear, but there was noise in the left/right channel. We noticed that we had to do a phase shift to mono audio because the stereo audio was passed through one more filter than the mono audio, which caused the phase difference between mono and stereo audio. So, we calculated the delay as the delay was simply the number of taps of the extra filter, which was int(AUDIO_TAPS*self.up_decim/self.down_decim/2). Finally, we used raw files whose left and right channels differed for testing.

RDS Processing

RDS is the most challenging part among the three signal paths. However, with the experience of mono and stereo, some parts of RDS are easy to implement. Channel extraction is straightforward. Then, we did not use the pilot tone for carrier recovery, but instead we used the extracted signal itself for tuning by squaring the extracted signal, bandpass the doubled frequency, and scale to half the frequency and tuning using PLL and NCO. Meanwhile, an all-pass filter is applied to the extracted signal, which aims to synchronize with the carrier against the delay caused by the filter. After that, the delayed extracted signal is mixed with the carrier by pointwise multiplication before the result is resampled. Up until now, RDS is like stereo in terms of coding and understanding. But what follows makes the RDS path challenging.

The mixed signal is first input to the RRC (root raised cosine) filter, whose code is given in Python. But it is important to realize that the signal after the RRC filter has graphically identifying characteristics. It is easy to judge whether the code prior to RRC filter works fine by looking at the output of the RRC filter. This could be a milestone for RDS path. In our case, we found a bug that we did not divide by 2 the length of the state of the all-pass filter.

After that, data is sampled by a clock which is Manchester decoded and then differential decoded, before the bitstream is used for frame synchronization where A, B, C, C’ and D are printed. This could be another milestone because the order of A, B, C, C’ and D is an intuitive criterion for the performance of the code. The decoding part is relatively easy, while the synchronization is more challenging. Two steps are taken for the synchronization part. The first step is to find the first A/B/C/C’/D by shifting the 26-bit window one bit at a time. After we find the first letter, the window starts shifting 26 bits each time. It took us a long time before we got the letters printed like A, B, X, X, A, X, X, D, …., whose order are correct, and the correct letters among which are enough for synchronization. During debugging, we found some barely visible bugs like mistaking variable i for j. Another bug was that we did not implement the states of the differential decoding correctly, which failed to give us the correct first code of each block.

Finally, the application layer is applied to the messages and offsets we just get. Unfortunately, the output is not correct for lack of time, but the main structure of the application layer is implemented.

```
program_id:  []
program type:  Classic rock
program service group:  A0
program service name addresses:  0
program service names:  ['\u202c绔+鈩', '綮\ue020+鈩', '娱괸+獵挠+黯', '擿+', '纂', '獵捊+纂', '獵挠纂
', '獥挠+纂', '獵挸+纂', '獵-']
Finished processing all the blocks from the recorded I/Q samples
```

Figure 2: Output RDS data produced by the Application Layer

C++ Asynchronous Processing

To increase the speed of our C++ program, we must take full advantage of the multi-core capabilities of the Raspberry Pi ARM processor. The following details the specific asynchronous functionality of C++ we used in our program. The benefit of the parallel processing is also that if one thread needs more runtime than available, the slower process will not bottleneck the speed of other more time-sensitive processes.

Thread allocations

To fully take advantage of the 4-core processor of the Raspberry Pi, the optimal number of threads without introducing unnecessary hyper-threading and context switching would be 4 parallel threads. We used the C++ <thread> library for opening and closing threads, and the queue library for storing vector data.

Our team has split our threads by the following table:

Main thread (RF front end producer)	Used for reading the binary data input stream from stdin, performs I/Q separation and demodulation, streams data into audio stream queue and RDS stream queue. Will stop producing if max queue size is reached
Stereo audio consumer child thread	Used for reading the demodulated data from the audio queue, passing it to the mono thread for processing, processing the stereo audio and writing binary data to stdout.
Mono audio consumer child thread	Initialized to process mono data in parallel with the stereo channel for each member of the audio queue, initialized by the stereo queue.
RDS data consumer child thread	Used for reading the demodulated data from the audio and passing it to the RDS thread for processing and passing the binary data to the application layer for interpretation.

Table 2: Threading allocations

Mutex locking

We used the C++ semaphore library as mutexes for locking critical sections of threaded processes to prevent race conditions with other threads. The following table details all the mutexes that we used and their purpose.

Audio_mutex	Mutex for locking audio queue for reading and writing demodulated RF data. Locked when writing by the RF FE thread or reading by the audio thread.
Rds_mutex	Mutex for locking RDS queue for reading and writing demodulated RF data. Locked when writing by the RF FE thread or reading by the RDS thread.
Mono_mutex	Mutex used by the audio thread for activating the mono processing thread when mono processing is needed.
Mono_done	Mutex for the audio thread to wait for the mono thread to finish processing while processing stereo in parallel to synchronize and speed up processing.

Table 3: Mutex allocations

Innovations

We realized, to further speed up stereo processing, we could take advantage of using all 4 processors with 4 threads by splitting mono processing and stereo processing. The mono thread is an independent thread that

listens for the audio thread to activate it and process mono data in parallel with the stereo data. The stereo path will then wait until the mono is complete to mix the audio and create the left and right channels.

We realized we could save CPU time and memory by passing pointers of intermediate data between threads to save the overhead for initializing and copying vectors. This also allowed us to pass data between our mono and stereo block seamlessly without a new queue. However, it is important to note that mutexes must be used (and sparingly) to prevent race conditions in reading and writing data.

Issues

One of the issues we faced was that, when sharing pointers to vectors between threads. We noticed that some data was overwritten prematurely as threads sometimes interfered unintentionally. We solved this by ensuring the separate intermediate states which were different, like the fm_demod vector that was inserted into the queue used a different vector than the fm_demod vector that was removed from the queue. But sometimes, a shared vector pointer was necessary to pass information from one thread to another.

C++ Specific Implementation Details

Beyond the conceptual details implemented in both C++ in python which are similar in both implementations, there are some specific C++ implementations which are detailed below.

Pre-allocation of Vectors

In python, since there are no initialization of vectors or concern for runtime, vector size allocation is not considered. But in our C++ implementation, specific procedures were performed to ensure optimal runtime for vector operations.

As explained in the cpp reference guide (3), and initialization of a vector has a large computational overhead due to the memory pre-allocation steps needed to prepare the vector. Additionally, any additional node added to the list or resize of a vector without re-allocating the capacity of a vector can produce an additional memory allocation overhead. We overcame this limitation by ensuring the procedures below.

1. All vectors for each block (mono, stereo, RDS) were pre-allocated and pre-loaded with the needed values and the required capacity of that vector before the initialization of block processing through the setup_vectors method in each class.

2. The use of any linear time vector operations were limited. Instead of using clear() other linear operations to reset a vector before each computation, we simply overwrote vectors of fixed size with new values without resizing or clearing them.

Infrastructure Innovations

Our team took a unique approach to deal with having so many separate processing steps by separating each processing task into a separate class. This allows us to hide implementation details of states and intermediate vectors within classes and not need to continuously pass vector pointers across functions increasing the chance of a failure. This also allows us to save vector pointers and data needed across multiple blocks without redeclaring or passing them through functions. But rather, they are universally available to all functions within the class, making function calls much easier to write.

Analysis and Measurements

Multiplications and Accumulations

*Note, it is assumed that only the action of multiply and accumulate is counted, individual multiplications for scaling and demodulation are not counted because their significance is small compared to the large size of the multiply and accumulates.

	Input Block_size	I&Q LPF RF_FE	Mono LPF
Mono Mode 0	76800	2*38400/10*101=775680	3840/5*101=77568
Mono Mode 1	245760	2*122880/10*101=2482176	12288/6*101=106848
Mono Mode 2	192000	2*96000/10*101=1939200	9600*147/800*101=178164
Mono Mode 3	92160	2*46080/9*101=1034240	5120*441/2560*101=89082
Stereo Mode 0	76800	2*38400/10*101=775680	3840/5*101=77568
Stereo Mode 1	245760	2*122880/10*101=2482176	12288/6*101=106848
Stereo Mode 2	192000	2*96000/10*101=1939200	9600*147/800*101=178164
Stereo Mode 3	92160	2*46080/9*101=1034240	5120*441/2560*101=89082
RDS Mode 0	76800	2*38400/10*101=775680	
RDS Mode 2	192000	2*96000/10*101=1939200	

	Stereo carrier extract BPF	Pilot extract BPF	Stereo final LPF
Stereo Mode 0	3840*101=387840	3840*101=387840	3840/5*101=77568
Stereo Mode 1	12288*101=1241088	12288*101=1241088	12288/6*101=106848
Stereo Mode 2	9600*101=969600	9600*101=969600	9600*147/800*101=178164
Stereo Mode 3	5120*101=517120	5120*101=517120	5120*441/2560*101=89082

	RDS Channel Extraction BPF	RDS Pilot extraction BPF	Rational re-sampler LPF
RDS Mode 0	3840*101=387840	3840*101=387840	3840*209/1920*101=42218
RDS Mode 2	9600*101=969600	9600*101=969600	9600*437/960*101=441370

	Root raised cosine filter	Total	Mult & Acc per sample (total/block_size)
Mono Mode 0		853248	11.11
Mono Mode 1		2589024	10.53
Mono Mode 2		2117364	11.03
Mono Mode 3		1123322	12.19
Stereo Mode 0		1706496	22.22
Stereo Mode 1		5178048	21.07
Stereo Mode 2		4234728	22.06
Stereo Mode 3		2246644	24.38
RDS Mode 0	3840*209/1920*101=42218	1635796	21.30
RDS Mode 2	9600*437/960*101=441370	4761140	24.80

Table 4: Multiply and Accumulate tally in all modes

Non-Linear Operations

	Input Block_size	Pll_stereo	Pll_rds	Total
Mono Mode 0	76800			0
Mono Mode 1	245760			0
Mono Mode 2	192000			0
Mono Mode 3	92160			0
Stereo Mode 0	76800	4*3840=15360		15360
Stereo Mode 1	245760	4*12288=49152		49152
Stereo Mode 2	192000	4*9600=38400		38400
Stereo Mode 3	92160	4*5120=20480		20480
RDS Mode 0	76800		4*3840=15360	15360
RDS Mode 2	192000		4*9600=38400	38400

Table 5: Non-linear operation tally in all modes

Runtime performance

Note*: Full Mode 0 total times are determined using the 'time' command and is the addition of user and sys times added

Note**: Individual module timing was done using the chrono library, all blocks that were dependencies was included in the time (ex. Stereo will include RF FE, Mono and Stereo calculations)

	Total time for **5s sample 101 taps**	Total time for **5s sample 13 taps**	Total time for **5s sample 301 taps**
Mono Mode 0	1.19299s	0.987968s	2.21426s
Mono Mode 1	1.07028s	0.680446s	2.12242s
Mono Mode 2	3.16516s	3.26342s	3.82069s
Mono Mode 3	3.03377s	2.69992s	4.02794s
Stereo Mode 0	2.12012s	1.58846s	4.35222s
Stereo Mode 1	1.92513s	1.15887s	3.98162s
Stereo Mode 2	4.86369s	5.21057s	6.38042s
Stereo Mode 3	4.65041s	4.36574s	6.53235s
RDS Mode 0	2.61749s	2.32254s	4.22444s
RDS Mode 2	2.25062s	3.28852s	4.65112s
Full Mode 0	4.269s	3.612s	7.073s
Full Mode 1	2.349s	1.624s	4.357s
Full Mode 2	7.863s	8.216s	9.829s
Full Mode 3	5.678s	5.172s	7.267s

Table 6: Runtime measurement table for all modes on RaspPi

Analysis

It can be seen as the number of taps of the filters increase in size, the runtime increases in length in all modes. This is because the number of multiplications scale proportionally to the number of taps directly. Meaning the more taps, the exact ratio accumulates, and multiplications are executed. However, by tripling the number of taps, the runtime does not triple, this is because there are other calculations in between which do not depend on the number of taps which are executed between the convolutions.

The ratio of number of multiplication and accumulates across mono, stereo and RDS scale consistently with the runtime as well. As the number of accumulates increase, the runtime will increase proportionally. However, there are a few inconsistencies with mode 2 and mode 3, this could be due to the extra steps needed during resampling.

As the number of taps increases, it can also be noted that the quality of the pass-band frequencies becomes stronger compared to the stop-band frequencies, leading to a faster and more accurate synchronization for the PLL. The higher quality filtering due to the increase in number of taps will also result in higher quality of audio. Respectively, as you decrease the number of taps, synchronization in the PLL state becomes worse, and the quality of audio will deteriorate. Although, it can be noted the difference in quality between 101 and 301 taps is minimal compared to between 101 and 13 because of the logarithmic nature of the improvement the stop-band frequencies become miniscule.

Proposal for Improvements

Vector size allocation in RDS

In our stereo block, we made intentional efforts to minimize any vector operations that were linear time complexity or worse by minimizing our use of clear(), resize() and using capacity allocations. However, in RDS, because of the complexity Manchester decoding and differential decoding, much of the bitstream vectors used needed to be cleared and appended to in each iteration. Although we made sure to allocate enough capacity to ensure that automatic resizing never occurred. However clear() operations still work in linear time, and so an improvement we could make is to find a way to constantly overwrite previous vectors without clearing them or changing their original size.

Memory optimization

In our program, each intermediate state between operations, such as upsampling, filtering, downsampling, and PLL have individually an assigned vector for holding data, and a vector saving state information for the next iteration. For future improvements, we can improve the memory bandwidth of the program by performing as calculations as possible in place using two alternating vectors. The extra memory needed for saving state information can also be reduced by extending state information at the end or beginning of the data vector. This can also remove extra condition checks required in convolution by removing the need check if the multiply and accumulate was beyond the scope of the vector. This will also improve the improvement in phase shifting in RDS by removing the need to splice together 2 vectors.

Project Activity

	Boxi Liu	Enyu Liu	Ridvan Song	Yinwen Xu
Week1	Read the project Specification	Read the project Specification	-Read the project specification -Setup python class infrastructure	Read the project Specification
Week2	Midterm recess	Midterm recess	Midterm recess	Midterm recess
Week3	Start mono single pass in Python		-Implemented python block processing for mono and RF_FE	Help with debugging the mono part
Week4	Start stereo in Python (all)	- Set up the C++ infrastructue - RF Front-end Block and Mono Block in C++	-C++ implementation for RF FE and Mono -Debug RF_FE and Mono block in C++	Work on translating stereo part from python to C++

Week5	- Debug stereo (all) python - Start RDS in Python (to RRC filter)	RF Front-end & Mono Block's Syntax Error	-Performance optimization in Convolution and vector resource allocation -Mono real-time testing	
Week6	- continue RDS in Python (to frame synchronization)	Stereo Block in C++	-C++ implementation of Stereo -C++ Stereo debugging and optimization -C++ Stereo real-time testing	
Week7	- debug RDS in Python (all)	- Stereo Block syntax check - RDS in C++ (Synchronization to Application)	-C++ implementation of RDS (all) -python application layer RDS -C++ debugging and verification with python for RDS(start – synchronization) -C++ threading and optimization -Real-time testing on raspberry pi	Do raspberry pi set up and help with real-time testing
Week8	Write Python part of implementation detail in report	Introduction, Overview and implementation (C++) in report	-Realtime performance measurement analysis -future improvements report -c++ implementation details, optimization, and threading	Work on the project report stereo processing part and conclusion

Table 7: Task planner and allocation

Conclusion

Over the last four months, we designed and built a full software defined radio system using a completely new language to us. By learning how to implement high level concepts of radio communication such as phase synchronization, FM demodulation, Quadrature decoding, and filtering in real-time, we gained a deep understanding of the intricacies of low-level languages such as C++, and a more thorough understanding of these theoretical concepts of radio communication. Furthermore, by applying software concepts like discrete mathematics, block processing, asynchronous execution, and memory allocation to the project, we were able to develop deep rooted knowledge that will allow us to continue building future systems of digital radio in the future. Ultimately, the experience we gained from learning how to plan, write, debug, and optimize a large-scale project with other team members have taught us crucial technical skills that we will take with us into industry, and skills in working in an agile software team environment.

References

(1)*Frequency domain of upsampling* *[*Online*]. Available:* https://www.projectrhea.org/rhea/index.php/Upsampling_Slecture_Molveraz#:~:text=Upsampling%20is%20the%20process%20of,input%20samples%20of%20the%20signal.

(2)*IQ Sampling* *[*Online*]. Available:* https://pysdr.org/content/sampling.html

(3)*cpp reference* *[*Online*]. Available:* *https://en.cppreference.com/w/cpp/header/vector*

(4)3DY4 course material [Online[. Available: https://avenue.cllmcmaster.ca/d2l/home/414164