COE3DY4 Project Report

Group 46
Guotong Miao, Ruiyan Guo, Xinyu Chen, Zhaohan Wang
miaog@mcmaster.ca, guor28@mcmaster.ca,
chenx237@mcmaster.ca, wangz394@mcmaster.ca

April 8, 2022

1 Introduction

The goal of the project is to develop and implement a Software Defined Radio (SDR) system to receive real-time Frequency modulated (FM) mono/stereo audio and digital data using the radio data system (RDS) protocol from a front-end radio-frequency (RF) hardware. This project is set up on the Raspberry Pi 4 with an RF dongle based on the Realtek RTL2832U chipset. The user is expected to interact with the SDR through the command line.

2 Project Overview

Frequency modulation (FM): The main object this project deals with is the FM signals, encoding of information in a carrier wave by varying the instantaneous frequency of the wave. Each FM channel occupies 200 KHz of the FM band and it is symmetric around its center frequency that can range from 88.1 MHz to 107.9 MHz in Canada.

FM Demodulator: A demodulator performs the demodulation operation to extract the original information-bearing signal from a carrier wave. It can be an electronic circuit or a computer program in a SDR. This project focuses on the positive frequencies of the demodulated FM channel (0 to 100 KHz) and there are three different sub-channels that are of interest. The mono sub-channel is from 0 to 15 KHz; the stereo sub-channel is from 23 to 53 KHz and the RDS sub-channel is from 54 to 60 KHz.

Software-defined radios (SDR): SDR is a radio communication system where components that have been traditionally implemented in hardware (e.g.mixers, filters, amplifiers, modulators/demodulators, detectors, etc.) are instead implemented by means of software on a personal computer or embedded system.

Finite impulse response (FIR): An ideal filter assumes infinite response, while in practice, the filters based on convolution are limited in size.

Phase locked loop (PLL): A PLL is a control system that generates an output signal whose phase is related to the phase of an input signal. It is crucial to make the phases in the lock when performing demodulation and synchronization.

Re-samplers: When converting the signals between different sampling rates, an up-sample and a down-sample are needed in general. A re-sampler combines them to acquire the efficiency needed in real-time execution.

Radio data system (RDS): RDS is a communications protocol standard for embedding small amounts of digital information in conventional FM radio broadcasts.

3 Implementation details

In order to implement the FM Radio processing, the project is divided into 5 sections which include RF front-end processing, mono processing, stereo processing, multi-threading, and RDS processing. The code from the previous lab is also taken advantage of in the project for implementation.

3.1 Labs

The main purpose of the labs is to implement some utility functions for the project and to gain an initial understanding of digital signal processing.

In lab 1, the main tasks were to understand the basic DSP primitives used in software-defined radios and implement them in python. There were four functions implemented: the impulse response of the LPF to replace the build-in firwin function, the actual filtering process (convolution) to replace the build-in filter, and Fourier transforms function to verify the design. The impulse response and the basic single-pass filter were relatively easy to realize since the implementation process just needed to follow the math functions and the given pseudo-codes. There were challenges when proceeding to the block processing implementation, it took some time to build an understanding of how the convolution process works and what is the state saving really about. The state can be treated as part of current block data concatenated at the start and unifying their index by using the fact that a negative index means to start from the end of a list. Once the theory became clear, the coding came out smoothly.

In lab 2, the task was to reprogram the python model into C++, the process was relatively straightforward. The only challenge was to get familiar with the programming language itself such as the usage of vectors.

In lab 3, all the work is done as a preparation for the project. The source code for the RF front-end is initialized in this phase.

3.2 RF front-end

According to lab3, the information transmitted and received by the RF dongle are the I/Q samples. The input is split into I and Q data by the function `split_iq_data()` which is used to store the data of even and odd indexes separately, where I stands for the in-phase component and Q stands for the quadrature component of an RF signal. Then, I and Q are convoluted separately to filter out the FM channel by the function `RF_front_end()`. The demodulation function takes the processed I/Q data as the input and combines them into the demodulated data with the IF sample rate.

3.3 Mono Processing

For this part, resampling and convolution are combined into a single function. Firstly, the impulse response generation function from previous labs is used here to compute the impulse response "h" based on the sinc function. In lab 3, the signal is only downsampled by a factor of 10. In order to process all the modes of operation generally, it is required to use both downsample and upsample operations during the process of convolution. However, doing them separately is prohibitively slow to run in real-time. Therefore, we follow the instruction in the lecture to combine the upsampling and downsampling into the convolution function. The main idea is to only calculate necessary components. We need to calculate input_size*U/D elements and during each element calculation, skip the 0s inserted in the upsampling by tweaking the index instead of really doing the upsampling. We also created a function to automatically change each frequency parameter and each mono expander U = AF/std::_gcd(IF,AF), front-end decimator D0 = RF/IF, and mono decimator D1 = IF * U / AF, so the code has extensibility to expand the modes.

Then, we test for different taps numbers and finally use the number in the lab, 151, which is a trade-off as more taps mean a good quality of sound but also cause more calculations. And for block size, we choose to use block_size = 1024 * D0 *D1 * 2 in order to have integer size during processing. However, when verifying modes 2 and 3, it is better to lower the block_size by changing the params.D1 to a small number like 50, as the mono decimator params.D1 in these two cases are all huge (800 and 400) in our constraints.

During the debugging process, we use the tools prepared in labs like Gnuplot and utilities to generate data files for plotting. At first, we did not increase the sampling frequency when generating the coefficients for the resampler and the audio is just noise and the plot for demodulation data indicates the signals are not filtered properly. We then locate the problem and fix the mono path.

3.4 Stereo Processing

The stereo processing implementation is generally very similar to the mono part, but it also brings in some new concepts such as PLL and BPF.As shown in the project guideline, our development process is also divided into three branches, carrier recovery, channel extraction, and the signal combining phase.

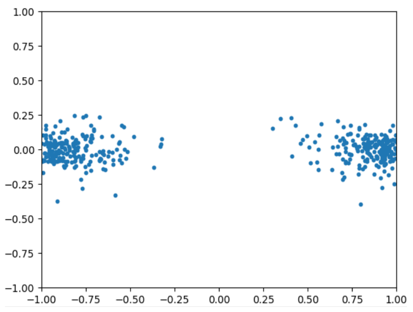
For the python model, everything came out smoothly, we implemented the bandpass filter by generating the impulse response according to the pseudo-code provided, for the pll, we modified it by adding a pll state saving component, except for ncoScale is set to be 2.0, all other variables remain as the default values. Then, at the stereo processing stage, the pointwise multiplication, low pass filtering, down-conversion, and channel concatenation are implemented in the same order.

For the C++ implementation, we encountered some problems when translating the code for pll, we originally stored all its properties in a map which is the C++ version of the library, but soon it is replaced by struct since it is found to be more convenient when embedded it into the pll function to modify its values. During the stereo testing stage, and we test for the new stereo testing raw file in week 5, we find the stereo cannot separate the left channel and right channel and start to debug. According to the lectures, we know we need to add an all-pass filter to delay the mono data in order to combine the mono and stereo data synchronically. However, after we added the all-pass filter, the stereo still did not work. We spent almost half a week going through our code again, checking each function that we wrote, plotting out all the data, for example, I and Q data and spectrum wave. Finally, we found the typo in the bandpass filter by unit testing the bandpass filter. This brought our attention to the unit test of the block even as small as typing formulas.

3.5 RDS Processing

The RDS processing is generally the trickiest part of the whole project. Although the overall data flow shares some similarities with the stereo part, it introduces a lot of new concepts, which need to be carefully designed.

For the python model, the data flow before the RDS demodulation block was carried out by modifying the stereo processing's code, the parameters for the bandpass filter as well as the pll properties were changed accordingly, some extra steps were added such as the squaring nonlinearity and allpass filter. For the demodulation block, the rational resampler is coded with the same logic flow as discussed in the lecture. For data recovery and manchester decoding, it is worth mentioning that the design is guaranteed to work for a strong input signal, we first used a peak detector to find the peak value for each symbol, then the following sampling points are found by adding the step SPS.



The algorithm for the peak detector is that we suppose that the magnitude of the middle position is relatively high or low. If the two adjacent points are of the same signs, the error = abs(middle point value - 0), else if the two adjacent points are of different signs, the error = abs(middle point value - (value point1 + value point2)/2). And then we iterate the error since our first mode of SPS is 19, which means 19 samples in a block. Therefore, we tested 19 different positions to be the start point and followed our detector peak method to find the I and Q data after running all of the blocks each time, and then we got 19 different iteration errors. Finally, we choose the starting point of the sample which gives us the smallest iteration error. By testing the correctness of the data, we also draw the peak point preCDR_I and preCDR_Q cluster plot to tune to an accurate phaseAdjust in python code. The phaseAdjust is a parameter to control the phase, therefore, we directly use what we got the phaseAdjust from the python test to enhance the accuracy of our CDR, and also draw the I and Q wave data plot to double-check whether the data is meaningful or not. After that, we highly improved the running time for the python model by changing all of our own functions (conv and my_lowpass) to the firwin() and lfilter(). Even lfilter still can not deal with large blocks as the memory limit is hit, so we implement the resampler in python to speed up the process.

During the pairing process, block 0 is specially treated. It tries and compares two different pairing methods to decide the best, the first method is to pair start from sample 0, if any HH/LL is detected, the process starts over from the sample 1, and in this case, the following block needs to pair the state from the previous block. This block 0 processing branch is critical since the rest of the blocks rely on the pattern found in block 0. After the differential decoding, where each bit

is XORed, the data is multiplied with the parity check matrix and it keeps looking for the first check word A to appear, after the first A is detected, the code regularly checks if the following check words appear in a consecutive manner, if an error arises, the particular block would be sent back to CDR function for repairing/resynchronization.
Due to the complexity of the RDS implementation, we encountered a lot of problems when testing this python model. One of the most representative issues we discovered was that we did not clear the state saving vector for CDR in the case of resynchronization. Resynchronization shares the same code as the block 0 processing branch mentioned above, originally, we considered that this branch only serves the very first block, and we did not address the fact that resynchronization will cause this branch to be re-introduced during the later processing period. The error, as a result, was whenever there is resynchronization needed, the next adjacent block always fails to detect any syndrome. Due to the lack of pass by reference mechanism for integers in Python, we use a list containing one flag variable to act as a semaphore between the CDR and frame synchronization.
For the C++ implementation, the logic of implementation is the same as that in python. Due to the tight schedule, it took a lot of time to debug and troubleshoot out C++ code which is written in a short time. The most difficult part of converting is finding the equivalent functions in C++ to that in python. For example, a python function `bitstream()` is used in the python file to transfer the binary strings to binary bits. We tried `std::bitset()` [1] at the beginning but then we found the size of the bitset is fixed, which means that it can't be resized. The sizes of most of the vectors in our code are decided by input data, therefore fixed size would not be taken advantage of well to store data. After abandoning the use of `bitset()`, we chose to use boolean vectors which can be defined as `std::vector <bool>` [2]. This type of vector is similar to a float vector, but the data stored in it only contains two types including `true` and `false`. However, after learning and testing the boolean type, we found it was allowed to use 0 and 1 directly as the representations of `false` and `true` separately.
In order to obtain the index of the data we want from the input or processed vectors, it is critical to transfer the binary numbers stored in strings to decimal integers. And in python, it is easy to convert the integers in the string to the integers with `int` type when making use of `.unit`. However, it is necessary to find another function with the equivalent functionality from C++ library. After searching and comparing, we selected `std::stoi()` [3] function to do the conversion. Because all numbers we processed are in base 2, we use 2 as the third argument every time the `std::stoi()` is used. The result generated by this function is an integer with the base of 10 which can be used as the index number.
There are still some issues that haven't been solved in our C++ code.

3.6 Multi-Threading
According to the 3DY4 lectures and operating system knowledge, we first learned, understood, and reviewed the principle of the multi-thread. Rather than the normal steady-sized queue to create multi-thread, we chose the circular queue that would be more convenient and efficient,

because it would automatically decide the actual time when consumer and producer threads work without conflicting or using a mutex to lock the data inside each thread. If we use the normal queue, we have to mention whether the parts of the thread locked by mutex are necessary or not. After deciding on the type of method we used, we started to follow the method mentioned in the lecture(March 17th) to create a queue with corporate steady size and separate two different types of threads, creating the RF_front_end thread that becomes a producer to produce the processed data and let the Frequency Modulated, FM, mono/stereo audio thread, and digital data, RDS thread consume the data from the queues simultaneously. Then inside each thread, we set the queue, write_offset, and read_offset. Using while(1) to decide when to produce and consume. When checking if the queue is empty, we use while (write_offset.load() <= read_offset.load()) , and set the waiting time 5-10ms for each checking to avoid the busy waiting loop and the need for threads on the CPU. To avoid conflict between the audio and RDS threads, we use two read_offsets for audio and RDS path. And the RF_front_end needs to check the slower offset to ensure the useful block will not be erased.

4 Analysis and Measurements

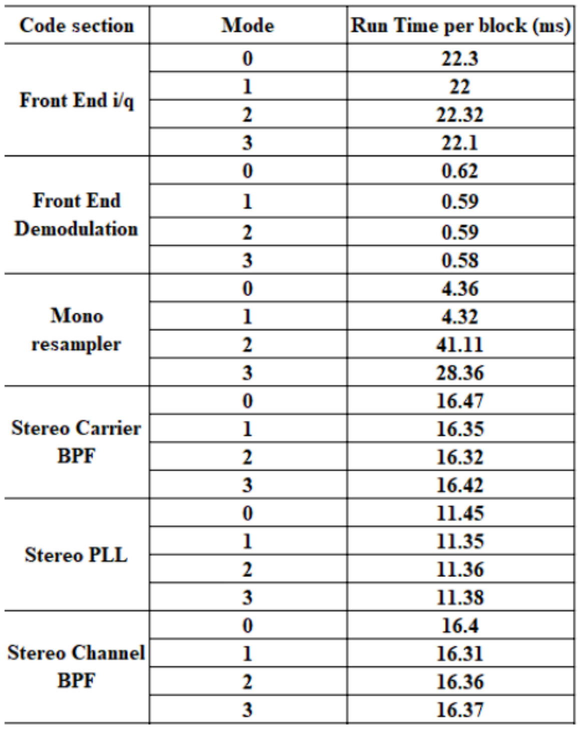
Assumption: block size = 1024 * front end decimator * 50 * 2; taps = 101

Analysis of computation load:

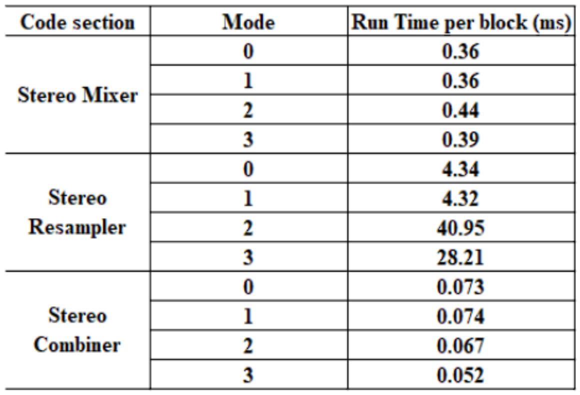
Path	Multiplications and accumulations per sample or bit	Non-linear operations per sample or bit
Mono mode 0	1111	0
Mono mode 1	1111	0
Mono mode 2	1200	0
Mono mode 3	1750	0
Stereo mode 0	2121	20
Stereo mode 1	2121	20
Stereo mode 2	2299.6	21.8
Stereo mode 3	3983.3	32.7
RDS mode 0	89326.5	1010.5
RDS mode 2	98618.5	1010.5

Runtime Measurements:

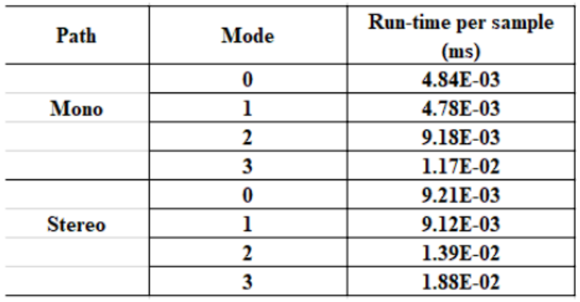
The result is an average of 100 blocks.   TAPS = 101:



Code section	Mode	Run Time per block (ms)
Front End i/q	0	22.3
	1	22
	2	22.32
	3	22.1
Front End Demodulation	0	0.62
	1	0.59
	2	0.59
	3	0.58
Mono resampler	0	4.36
	1	4.32
	2	41.11
	3	28.36
Stereo Carrier BPF	0	16.47
	1	16.35
	2	16.32
	3	16.42
Stereo PLL	0	11.45
	1	11.35
	2	11.36
	3	11.38
Stereo Channel BPF	0	16.4
	1	16.31
	2	16.36
	3	16.37



Code section	Mode	Run Time per block (ms)
Stereo Mixer	0	0.36
	1	0.36
	2	0.44
	3	0.39
Stereo Resampler	0	4.34
	1	4.32
	2	40.95
	3	28.21
Stereo Combiner	0	0.073
	1	0.074
	2	0.067
	3	0.052

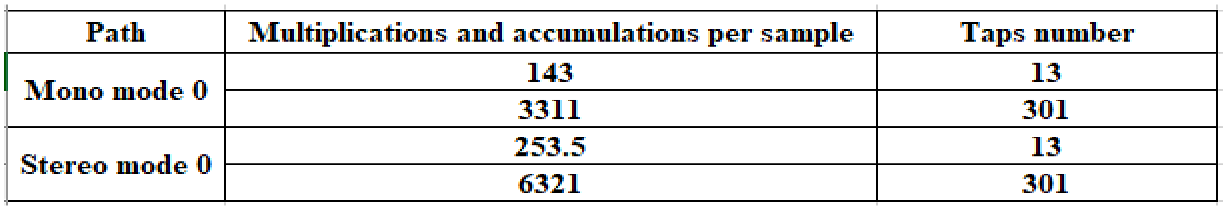


Path	Mode	Run-time per sample (ms)
Mono	0	4.84E-03
	1	4.78E-03
	2	9.18E-03
	3	1.17E-02
Stereo	0	9.21E-03
	1	9.12E-03
	2	1.39E-02
	3	1.88E-02

From the tables above, the analysis agrees with the measurements. The runtime per sample has the same trend as predicted by the number of multiplications and accumulations and the number of non-linear operations. Mode 0 and mode 1 have similar runtime per sample, and modes 2 and 3 have increasing runtime. The runtime of the stereo path is approximately 2X the runtime of the mono path. The ratio between the runtime of mono mode 2 and mode 1 is a little surprising at first that it should be about 1.2X by analysis but about 2X in measurement. This makes sense due to the extra 2X multiplications compared to the normal convolution used to calculate the indices in the resampler block. Among the building blocks, the filter blocks dominate the run time as convolution operation is the bottleneck of this project.

Impact of filter taps number:

Analysis:



Path	Multiplications and accumulations per sample	Taps number
Mono mode 0	143	13
	3311	301
Stereo mode 0	253.5	13
	6321	301

Measurements (average in 100 blocks):

Code section	Runtime per block (ms)	Taps number
Front End i/q	3.09	13
Front End Demodulation	0.59	
Mono resampler	0.6	
Stereo Carrier BPF	2.26	
Stereo PLL	11.61	
Stereo Channel BPF	2.29	
Stereo Mixer	0.49	
Stereo Resampler	0.61	
Stereo Combiner	0.069	
Front End i/q	63.1	301
Front End Demodulation	0.58	
Mono resampler	12.61	
Stereo Carrier BPF	47.35	
Stereo PLL	11.41	
Stereo Channel BPF	47.35	
Stereo Mixer	0.38	
Stereo Resampler	12.58	
Stereo Combiner	0.061	

Path	Runtime per sample (ms)	Taps number
Mono mode 0	7.20E-04	13
	1.36E-02	301
Stereo mode 0	2.35E-03	13
	2.40E-02	301

From the above tables, by varying the taps number, the computation load will vary proportionally as predicted by the analytical analysis. Comparing the 13, 101, and 301 taps, the runtime per sample increases, and the audio quality gets better. When using 13 taps, the audio has a lot of high-frequency noise and distortion, while using 301 brings the crystal clear audio quality.

5 Proposal for Improvement

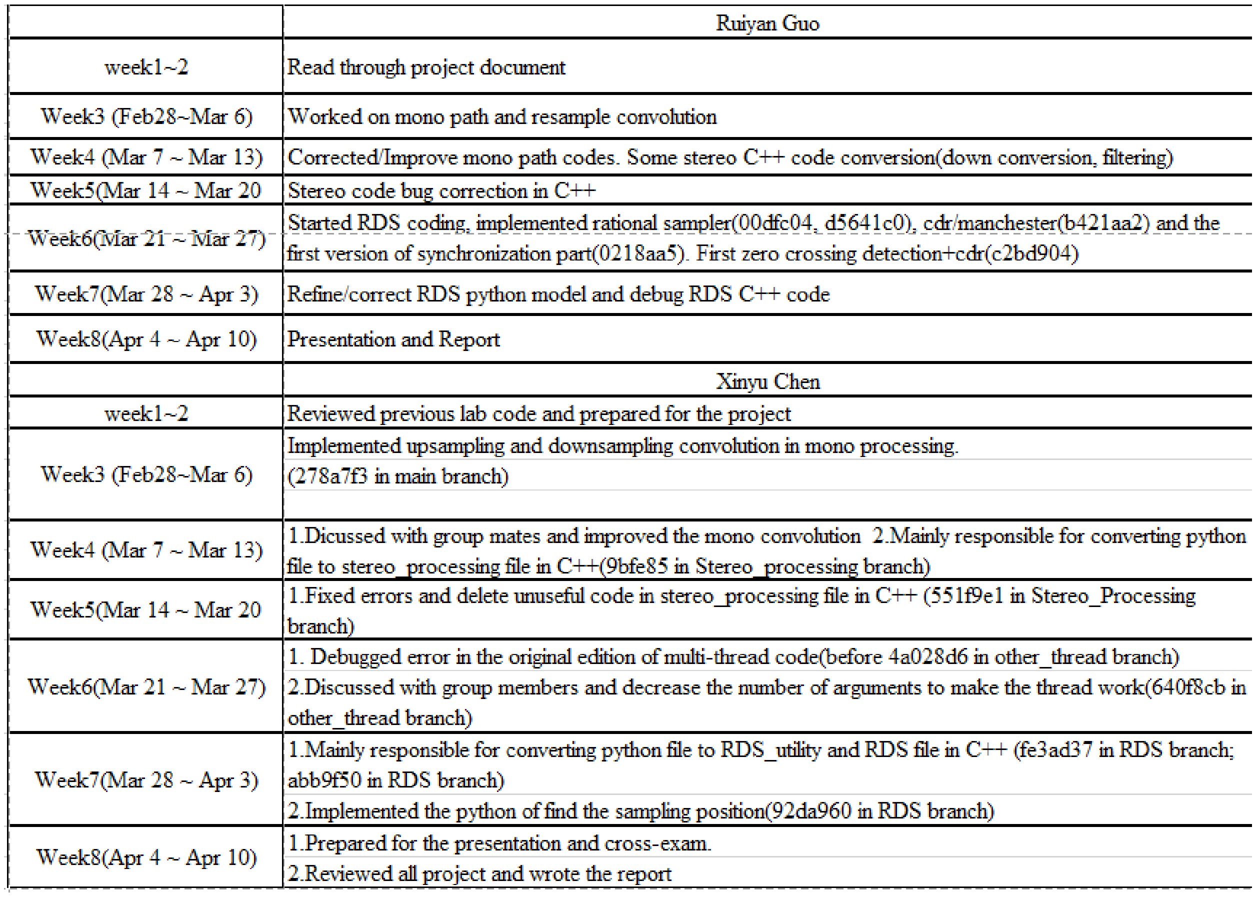
1. In order to provide a better user experience, we could package all of the code and create an app that can insert the input, build an interface to display the name of fm station, and then build another sub-window to show the RDS syntax and the waveform or the spectrum of the voice simultaneously.
2. When testing the cluster data for the scatter plot, we could use the cluster model to calculate the Kmeans so that we can test the exact value of a bunch of data to decide which data is better.
3. To improve productivity, we could exert the test framework and write unit tests for each building block we built to make sure the work done is reliable and facilitate the test for larger blocks later.

To improve the runtime performance, we could add one more thread for RDS thread to split the functions of timing recovery and frame synchronization, so that these two blocks can benefit from pipelining to improve the runtime per bit.

6 Project Activity

	Guotong Miao
week1~2	Reviewed lab codes and read project doc
Week3 (Feb28~Mar 6)	Built the main structure. Prepared header files. Finished basic workflow of mono path.
Week4 (Mar 7 ~ Mar 13)	Implemented resampling algorithm discussed in lecture. Corrected sampling frequency used by resampler. Finished front-end demodulation. Debugged, fixed state saving related issues and finished mono path. Built structure for stereo path.
Week5(Mar 14 ~ Mar 20)	Fixed state saving of pll. Finished I/O structure for stereo path. Refactored coefficients initialization and debugged stereo path.
Week6(Mar 21 ~ Mar 27)	Added delay on mono path to match with stereo path. Applied gain after mixer.Corrected typo in bandpass filter. Finished stereo path. Started frame synchronization for RDS path model in Python. Modified pll to output q component. Prepared RDS thread in C++
Week7(Mar 28 ~ Apr 3)	Built basic logic for message parsing in the RDS model. Tuned pll parameter. Discussed sampling position algorithm for clock data recovery. Fixed state saving issue in RDS model. Refactored CDR function. Implemented resampler in Python. Fixed message combining logic. Refactored and implemented RDS in C++.
Week8(Apr 4 ~ Apr 10)	Wrote report section 4, participated in other sections, finally go through the whole report. Reflected on project experience.

Work Table	Zhaohan Wang
week1~2	Review labs code, read and understood the project document.
Week3 (Feb28~Mar 6)	1. Wrote an algorithm to calculate the parameter for each mode. (mode.cpp) (85f75a3) 2. Finish the filter.cpp (e6f8e40) 3. Implement the RF_front_end from python to c++. (3c839c2)
Week4(Mar 7 ~ Mar 13)	1. Finish RF_front_end without state block. (6a29ccb) 2. Improve the initialization issue of block state to increase the sound quality.
Week5(Mar 14 ~ Mar 20)	1. Writing code in python, create the mixer and output the stereo(efbb569) 2. After new testing raw file upload, I found the problem after test, recheck the code what we wrote 3. debug the c++ code with Guotong(5bb8037) (006593e) change the state saving and I/O structure for stereo
Week6(Mar 21 ~ Mar 27)	1. Still debug stereo and add all pass filter at (08aceee) work with Guotong just after the Monday lecture, but the stereo still worked not correctly, I add the break point and plot graphs for wave and recheck the grammar of code to verify but did not commit. 3. write RRC and implement in c++.(ee9580e) 4. built the frame of thread for whole project. 5.used circular queue to avoid mutex and pass the test, more clear explanation wrote in report section 3.6 multi-thread. First version of thread (4284951) final version of thread (db7672f) (640f8cb)
Week7(Mar 28 ~ Apr 1)	1. testing the RDS and debug in python, change our own conv and filter function by firwin() and lfilter to highly increase the efficiency of the code. (0ec35ce) (also Test the running time before and after, huge time difference) 2. Create peak detector. (1aec312) (include another method in f5ef483) 3. test and draw the I q wave plot and draw the peak point preCDR_I and preCDR_Q cluster plot to tune to a preparate phaseAdjust in python code. The clear explanation is what I wrote at the beginning of section 3.5 RDS processing in the report. Tunning pll parameter. 4. CDR implement on c++ and c++ debug, change the typo(f5ef483)
Week 8(Apr 4 ~ Apr 8)	wrote report in section 1,2,7 part of 3 and 5, participated in others sections, change the format and typo, add relative plot



	Ruiyan Guo
week1~2	Read through project document
Week3 (Feb28~Mar 6)	Worked on mono path and resample convolution
Week4 (Mar 7 ~ Mar 13)	Corrected/Improve mono path codes. Some stereo C++ code conversion(down conversion, filtering)
Week5(Mar 14 ~ Mar 20	Stereo code bug correction in C++
Week6(Mar 21 ~ Mar 27)	Started RDS coding, implemented rational sampler(00dfc04, d5641c0), cdr/manchester(b421aa2) and the first version of synchronization part(0218aa5). First zero crossing detection+cdr(c2bd904)
Week7(Mar 28 ~ Apr 3)	Refine/correct RDS python model and debug RDS C++ code
Week8(Apr 4 ~ Apr 10)	Presentation and Report
	Xinyu Chen
week1~2	Reviewed previous lab code and prepared for the project
Week3 (Feb28~Mar 6)	Implemented upsampling and downsampling convolution in mono processing. (278a7f3 in main branch)
Week4 (Mar 7 ~ Mar 13)	1.Dicussed with group mates and improved the mono convolution 2.Mainly responsible for converting python file to stereo_processing file in C++(9bfe85 in Stereo_processing branch)
Week5(Mar 14 ~ Mar 20	1.Fixed errors and delete unuseful code in stereo_processing file in C++ (551f9e1 in Stereo_Processing branch)
Week6(Mar 21 ~ Mar 27)	1. Debugged error in the original edition of multi-thread code(before 4a028d6 in other_thread branch) 2.Discussed with group members and decrease the number of arguments to make the thread work(640f8cb in other_thread branch)
Week7(Mar 28 ~ Apr 3)	1.Mainly responsible for converting python file to RDS_utility and RDS file in C++ (fe3ad37 in RDS branch; abb9f50 in RDS branch) 2.Implemented the python of find the sampling position(92da960 in RDS branch)
Week8(Apr 4 ~ Apr 10)	1.Prepared for the presentation and cross-exam. 2.Reviewed all project and wrote the report

7 Conclusion

This project combines different types of knowledge related to Fourier transform, operating system, control system, and RF principle, which highly develop our theoretical knowledge and improve our debugging skills within python and c++. All of the members in our group enhanced their teamwork, time management, and self-learning skills. During the cooperation, we were much more familiar with git to share and exchange our progress efficiently, which helps our programming skills as a team a lot. The project provides a strong connection between the knowledge that we learned before and the real-life system which gives us a good understanding and real motivation to improve ourselves in the future.

8 References

1. https://www.cplusplus.com/reference/bitset/bitset/
2. https://en.cppreference.com/w/cpp/container/vector_bool
3. `std::stoi()`. C++ Reference. Available from:
4. All 3DY4 lectures and slides on A2L.
5. https://www.embedded.com/dsp-tricks-frequency-demodulation-algorithms/
6. c++ - Concatenating two std::vectors - Stack Overflow
7. vector element-wise product with C++ and STL - Stack Overflow