COE3DY4 - Project Report
Group 46, Group 47 - Thursday
Declan Kinsella, kinselld, 400179789 Yuxin Zhou, zhouy213, 400187135
Hongwei Niu, niuh4, 400191208 Haoran Qi, qih12, 400180773
April 9, 2021

1 Introduction

This project focuses on implementing a real-time implementation of a software FM receiver that follows industry-level specification and operates in a form factor-constrained environment. The project leverages front-end radio-frequency (RF) hardware, single-board computers (Raspberry PI 4), and the reception of digital data sent through FM broadcast using radio data system (RDS) protocol.

2 Project Overview

The whole project was divided into four parts and were completed in order. The **RF front-end processing** uses a low-pass filter to extract the FM channel, followed by decimation to get intermediate frequency (IF) which is then FM demodulated. The **Mono path** uses floating point format IF samples as input and reduces the sample rate to 16-bit signed integers with 48 KSamples/sec, which can be output to an audio player. The **Stereo path** uses the same IF samples as the mono path as input, which is then passed into a bandpass filter and PLL implementation to extract the 19 KHz pilot tone. The pilot tone is then filtered and combined with mono audio data to produce the left and right audio channels. The **RDS path** recovers the subcarrier, implements down-conversions and resamplers, performs clock and data recovery, generates the bitstream and performs frame synchronization with IF samples.

The project has two modes of operation, which could be chosen by the user by inputting different mode parameters in the command line: **2.4 MSamples/sec** input rate for **mode 0** and **2.5M Samples/sec** input rate for **mode 1**. After decimating by a factor of 10, the IF sample rate is 240 KSamples/sec for mode 0 and 250 KSamples/sec for mode 1 (RF thread). Mono path and Stereo path use floating point format IF samples as input and reduce the sample rate to 16-bit signed integers with 48 KSamples/sec.
The **mono path** was combined with the **stereo path** to create the audio thread. The most significant hurdle as implementing the audio thread was designing efficient block convolution. To use the RF thread and audio thread in real-time, the number of computations in convolution had to be decreased significantly. The final design used was to jump over the indexes in which data would not be used as part of the outputs.

The main challenges for the **RDS path** were to ensure the PLL had locked the data correctly, and to analyze the logic of the RDS data processing block. Since the index of choosing the first data in clock and data recovery was critical, we decided to choose the index which would lead one block to have the less high-high or low-low cases as the “maximum index”.

All four parts would be first built in Python model, then be refactored to C code to be further optimized. The code would be tested in real time on the Raspberry PI 4.

3 Implementation

Multi-Threading

Three threads are created to run the main SDR functionalities and two synchronization queues are created for storing the data after the FM demodulation, which will be further used by audio and RDS threads. The first queue is used for data synchronization between RF front-end thread and audio thread, which includes mono path and stereo path. A data type called ***ToAudio*** is declared to store necessary data to pass into the audio thread and RDS thread. The second queue, ***rds_queue,*** is used for data synchronization between RF front-end thread and RDS thread. Since only demodulated data is passed

into the RDS thread, we define the data type of each position in the queue as a float. Mutex and conditional variables are created for each thread to ensure shared sources are mutually exclusive and avoid deadlocks. The threads are defined as ***front_thread***, ***audio_thread*** and ***rds_thread*** using std::thread() from the thread class, and the join() function is used to execute them. ***Thread_demod*** is used to implement details of the RF front-end processing path, ***thread_rds*** to implement the RDS path and ***thread_audio*** for mono and stereo paths. In ***thread_demod***, a struct *ToAudio collection* is used to collect the data of audio_coeff_mode, fm_demod and fm_demod_1 after demodulation and push them into the queues. A unique lock is created for blocking ***audio_thread*** when the data in the queue reaches the maximum number (QUEUE_BLOCKS) of 5. At the end of this function, we use notify() to wake up the **audio_thread** and **rds_thread** as *my_queue* and *rds_queue* become not empty and not full. In ***thread_audio***, we first check if my_queue is empty or not. If it is empty, we create a unique lock and block audio_thread until there is new data put into the queue through the execution of ***thread_demod***. After that, data is extracted from the collection, popped from the queue and audio thread is run. At the end of ***thread_audio***, we unlock the mutex variable and wake up other blocked threads, allowing processing to continue. The same logic is applied to ***thread_rds***, as both ***thread_rds*** and ***thread_audio*** will receive the data from ***thread_demod*** and do their respective processing.

RF Front-End

The design process of the RF front-end is similar to the Python model in Lab 3, we refactorized the Python model into C++ codes. The goal of the RF front-end is generating the I/Q samples for the future use. This is done by filtering, downsampling and demodulating the 8-bit unsigned integers input acquired by RF hardware.

thread_demod():

The 8-bit unsigned integers input are divided into blocks, the size of each block is 51200 samples (1024 * rf_decim * audio_decim) for mode 0, and 50000 samples (1000 * rf_decim * audio_decim) for mode 1. Each block is split into in-phase (I) and quadrature (Q) samples, and run through a block convolution function to apply a low-pass filter with a 100 KHz cutoff frequency. Since the coefficients of the impulse response in LPF with the same size and same cut-off frequency Fc should keep the same, this calculation is done at the beginning of execution and reused across blocks. The coefficients are passed as an array by reference into the functions ***demod()*** and ***thread_audio()***.

demod():

The first difference between the Python model and the C++ codes in this part is to reduce the amount of calculations in C++. To achieve the real-time implementation, the application of the decimator and the LPF were combined, meaning it was only necessary to do the convolutions one time for every tenth sample.

The second difference between the Python model and C++ is that our own arctan function was needed instead of directly using the built-in atan2() from Python. FM Demod is the part needed to design according to the formula for the frequency demodulator algorithms. In the function ***demodArctan()***, each value of fm_demod is calculated by the following formula: $\Delta\theta = \frac{I[k]*(Q[k]-Q[k-1])-Q[k]*(I[k]-I[k-1])}{I[n]^2+Q[n]^2}$. To ensure continuity between blocks, the last value of I[k] and Q[k] in the current block is buffered as buf_I and buf_Q, and used as Q[k-1] and I[k-1] at the start of the next block.

Mono Path

The mono path has two modes, which use the IF samples (240 KSamples/sec for mode 0 and 250 KSamples/sec for mode 1) as inputs. This path filters, downsamples and demodulates the input IF samples to generate the audio output with a sample rate of 48 KSamples/sec.

audio_samples():

The mono processing is built based on the Python code from Lab 3. The input undergoes mono channel extraction, filtering and downsampling to the desired 48 KSamples/sec audio sample rate. The mono path receives the demodulated data (*fm_demod*, *fm_demod_1*) as inputs from RF front-end path (***thread_demod()***). The cutoff frequency (Fc) in mono mode 0 is 16 KHz. For mode 1, the Fc can be chosen to but up to 24 KHz (maximum of $\{(\frac{U}{D}*\frac{IF}{2}), \frac{IF}{2}\}$). To reduce the number of variables and keep the two modes consistent, we chose to use the same Fc = 16 kHz as mode 0.

In mode 0, to reduce the number of computations, the 16 KHz LPF is combined with the decimator with decimation of 5. For the mode 1, the upsampling factor is chosen to be 24 because the greatest common divisor for 250K and 48K is 2, and U is chosen as the ratio of output sample rate (48KSamples/sec) and GCD value. The upsampler is combined with FM demod operation by calling ***demodArctan1()*** in ***demod()***. The upsampled output fm_demod_1 is resized as 24*fm_demod.size() with initialized value of 0.0. Each time a new fm_demod data with index k is processed, it is stored in fm_demod1 in the index of 24*k. To generate the output sample rate correctly, the downsampling factor is chosen to be 125. The act of downsampling is combined with the 16 KHz LPF.

When processing the block convolution ***effBlockConv()*** for mode 1 in C++, we found that it could not meet the expected speed of playing the audio through only jumping the indexes. To solve this, we found the indexes of the coefficients which would be used in the convolution computations as positions to jump to. The indexes used in the computations is a cycle of numbers. Thus, we stored the indexes and used an if condition to decide which indexes would be used in each cycle. After doing this, the mode 1 can be implemented efficiently to run in real-time. In summary, the main idea is to skip all data points which are zeros when convolving the data that is populated with many predictable zero points.

Stereo Path

The stereo path receives IF samples and passes them to two separate sub-blocks: stereo carrier recovery and stereo channel extraction, mixes them, filters and processes the final signals, and combines them with mono audio to produce the left and the right audio channels. In the ***fwrite()*** function, the order of the data needs to be left, right, left, right, etc. to output the data in stereo format. The influences of the stereo part to the output sounds are very subtle in most cases, and may not be noticed by directly observing the aplay output. To better understand the process and to ensure the result is correct, the Python model was constructed, and plots of the output waves were monitored at several stages in the process to ensure its correctness. The Python code was then refactored and optimized into C++ code.

thread_audio():

The ***thread_audio()*** waits until the notification from RF front-end, then initializes required variables and calls ***stereoExtrRec()*** to implement the whole stereo path.

In stereo carrier recovery, the stereo signal is locked to the second harmonic of 19 KHz pilot tone and is modulated onto a 38 KHz subcarrier. The first important part for stereo carrier recovery is to ensure the PLL (***fmPLL()***) locks the signal correctly. We plotted and compared the wave before PLL and the wave after PLL, it could be seen clearly that each input wave corresponds to two output waves, which means the 19 KHz input increases to 38 KHz and is successfully locked. The next important area of the code is making sure that between each block, continuity is maintained. We choose to save states (*integrator, phaseEst, triArg, feedbackI, feedbackQ, trigOffset, ncoOut_buf_I, ncoOut_buf_Q*) for current block in order to use them in the beginning of the next block.

After PLL, the signals are be mixed with the output of stereo channel extraction in ***mixer()***, where the IF samples are filtered by a bandpass filter will a passband from 22 KHz to 54 KHz (***stereoBPConv()***), then followed by a LPF which is included in the digital filtering sub-block.

audio_samples_stereo():

The process of digital filtering in stereo processing blocks is identical to the one in mono path. Thus, we directly call the same function ***blcokConv1()*** for mode 0 and ***effBlockConv()*** for mode 1.

combiner():

The combiner combines the stereo outputs after the digital filtering with the mono audio samples: left channel = (stereo + mono)/2, right channel = (stereo - mono)/2.

RDS Path

The RDS thread includes only mode 0, it is the most challenging and complicated one of the whole project. The RDS channel uses the same basic principle as for the stereo channel, with a center frequency at 57 kHz and a channel bandwidth iof 6 kHz, which can be extracted using a band-pass filter. The frequency of signals after squaring non-linearity are increased by a factor of 2, thus the input to the PLL is a signal with a 114 KHz tone (57Khz *2). In our design, we choose not to implement the all-pass filter, instead, we generate in-phase(I) and quadrature(Q) data concurrently by using cosine and sine function in the PLL and NCO (***rds_fmPLL()***).

In the rational resampler part of RDS demodulation, we design a upsampler with 19 decimation, and a downsampler with a factor of 80. This results in an output with a sample rate of 24 times of 2375 symbols/sec (24*2375 = 240000*19/80). Since we want to reduce the number of computations to meet the time limitation in real-time, the decimation for upsampler needs to be as small as possible, and a factor of 19 is the smallest possible amount that can meet the requirements. The maximum frequency for the LPF in the rational resampler is 19 kHz(maximum of $\{(\frac{U}{D}*\frac{IF}{2}), \frac{IF}{2}\}$). We chose to use 16 kHz in our design.

The most critical and essential part of the RDS path is the clock and data recovery block. The data recovered should be as accurate as possible to ensure the inputs for RDS data processing is correct. The PLL needs a certain amount of samples to lock onto the desired frequency, which causes several data at the beginning to be incorrect. To compensate for this, we drop the block 0 data to mitigate this noise. Also due to noisy inputs, we could not get ideal data to read in real-time, leading to some cases where samples follow the illegal pattern of high-high or low-low in the same bit.

Our design (line 386 to line 401 in rds.cpp) chooses the starting index which has the least cases of containing high-high and low-low cases as the “max_index” in block 1, and the following data would be sampled at every 24th sample following. The chosen symbols are used as the input to implement RDS data processing block (***manchester()***, ***frame_sync()***). In our design, the original block size from reading data through RF dongle is 1024*10*5 = 51200. After demodulation, the block size for IF samples is (51200/2)/10 = 2560, and following the rational resampler, the size of output after RRC filter should be 2560*19/80 = 608, which is not multiple of 24. To simplify the calculation, we choose to store 3 blocks RRC data, then send those 3 blocks data to implement clock and data recovery. This gives ush 608*3 samples to work with, which is multiple of 24, and data recovery will be much faster and straightforward to implement. Due to the time limitation for this semester, our project contains only until frame synchronization and error detection block.

manchester():

The function ***manchester()*** decides the value of bits, then performs XOR computations with every 2 adjacent symbols. Each time, the input size is 608*3/24 = 76, and the output size is 76/2 = 38. If there is any situation that high-high or low-low happens, the higher value would be seen as high, and the lower

value would be regarded as low. The last value of the XOR result of the current block would be saved as *buf_mach*, which is used as the input for the XOR with the first bit in the next block.

frame_sync():

The output bitstream with the 38 data for each 3 blocks from ***manchester()*** is used as the input of the frame synchronization and error detection block ***frame_sync()***. Because we drop the block 0 and send 3 blocks data at a time to the RDS data processing block, the first input bitstream should start at block 1 data and end up with block 3 data. To implement the real-time frame synchronization, the input and output size should be kept at a certain amount. It can be observed that as block_id = 3, the frame synchronization would only generate 12 times (from index 0 to index 11); as block_id >= 6, the frame synchronization would generate 38 times (from index 12 to index 49 for second generation). We choose to drop the data at index 0 to index 37 after the second generation has been done, move the data from index 38 to index 75 to index 0 to index 37, and store the next 38 new data into index 38 to 75.

In ***frame_sync()*** function, the framed 26 bits would be used for matrix multiplication with the H matrix, and the 10 bit result would be compared to syndrome to match the types. Since there are two types for C (C and C'), it needs to add a condition to ensure either C or C' could exist in one ABCD cycle.

4 Analysis and Measurements

RF Front-End

The input raw data first goes through the set of computations in the ***blockConv_I()*** function. The number of multiplications and accumulations per sample is based on the number of LPF coefficients used in the computations for each output sample, which is 101(same as the number of taps). Thus, for both mode 0 and mode 1, the number of multiplications per output sample of ***blockConv_I()*** function is 101. Since the runtime for mode 0 ***blockConv_I()*** is around 0.371056 ms, the runtime per sample output is 0.371056/2560=0.00014494 ms and the runtime per sample for mode 1 ***blockConv_I()*** is around 0.302349/2500=0.00012094 ms. The runtime for mode 0 and mode 1 ***blockConv_I()*** should be close, which is also supported by our output runtime measurement. Same principle applies to the ***blockConv_Q()*** function, and the runtimes for ***blockConv_I()*** and ***blockConv_Q()*** should be very similar.

The ***demodArctan()*** (mode 0) and ***demodArctan1()*** (mode 1) functions take the outputs from ***blockConv_I()*** and ***blockConv_Q()*** functions and use them for arctan computations. Each output in the fm_demod array implements three subtractors: Q[k]-Q[k-1], I[k]-I[k-1], I[k]*(Q[k]-Q[k-1]) - Q[k]*(I[k]-I[k-1]); four multipliers: I[k]*(Q[k]-Q[k-1]), Q[k]*(I[k]-I[k-1]), I[k]*I[k], Q[k]*Q[k]; one adder: I[k]*I[k]+ Q[k]*Q[k]; and one divider. The runtime of the ***demodArctan*** function for a single block is 0.006826ms, resulting in a per-sample runtime of around 0.006826/2560=0.0000026664 ms; and likewise the per-block and per-sample runtimes of ***demodArctan1()*** are 0.013483ms and 0.013483/2500= 0.0000053932 ms respectively. The runtimes for the ***demodArctan1()*** is higher than the runtimes for ***demodArctan()*** because there are two more operations which implement the upsampler.

The total runtime in RF Front-End for both modes of one block is 0.764904 ms, the total RF front-end runtime per sample is 0.764904 ms/2560 = 2.99e-4 ms (mode 0) or 0.764904 ms/2500 = 3.06e-4, and the times of multiplication and accumulation per output sample is 101.

RF Front-End Main Functions Runtimes:

Mode 0: ***blockConv_I()***: 0.371056 ms ***blockConv_Q()***: 0.363604 ms ***demodArtan()***: 0.006826 ms

Mode 1: ***blockConv_I()***: 0.302349 ms ***blockConv_Q()***: 0.294053 ms ***demodArtan1()***: 0.013483 ms

Mono

The fm_demod data used as the input to ***audio_samples()***, ***blockConv1()*** is used to generate the block convolution for mode 0, which uses 101 LPF coefficients in its convolution computations. For mode 1, ***effBlockConv()*** is used to speed up the run time due to the extra steps in the rational resampler. The number of coefficients used as the buf_wave data has not used up is 101, while after the buf_wave is used up, since some values of input x would be 0 from the rational resampler, jumping over values which are 0 results in the number of coefficients used as the buf_wave is not used up being 101*24. After the buf_wave is used up, it should be 101*24/24 = 101.

For mode 0, each output sample would have 101*2(blockConv_I() and blockConv_Q()) +1(demodArctan())+101(blockConv1()) = 304 multiplications or accumulations, and for mode 1, each output sample after the bud_wave has used up would be 101*2+1+101 (effBlockConv()) = 304 multiplications or accumulations.

The function ***blockConv1()*** has a decimation of 5, which means in theory, the runtime per block convolution should be ⅕ of the ***blockConv_I()*** in mode 0. From testing the runtimes for ***blockConv1(),*** the average runtime for it is about 0.0881751 ms, and the ratio of the runtime for ***blockConv1()*** and ***blockConv_I()*** is 0.371056/0.0881751= 4.21, which is close to the expected factor of 5.

The function ***effBlockConv()*** should has a theoretical runtime about 0.302349/5=0.0604698 ms (around ⅕ of the ***blockConv_I()***), while the testing runtime is around 0.161741 ms. The real runtime is much higher than expected due to some operations in the ***effBlockConv()*** which were added to choose the correct jumping indexes to run the block convolutions efficiently.

The total runtime in mono path for one block should be longer than the RF Front-End path, which includes some initialized variables and one 16k LPF combined with a decimator. The tested runtime is (1.40125 ms + 0.764904ms)/2560 = 8.47e-4 ms/sample(mode 0) or (1.60647 ms + 0.764904 ms)/2500 = 9.49e-4 ms/sample (mode 1).

RF Front-End Functions Runtimes:

Mode 0: ***blockConv1()***: 0.0881751 ms Mode 1: ***effBlockConv()***: 0.161741 ms

Stereo

Though the digital filtering part is similar to the mono path, the stereo path has more computations than mono path due to the stereo carrier recovery, stereo channel extraction, mixer, and stereo combiner blocks.

The ***stereoBPConv()*** is the most direct and simplest block convolution. It has no jumping indexes and does the convolution computations one by one. The number of multiplications and accumulations is based on the number of band-pass filter coefficients, which is 101. The block convolutions used in stereo path are identical as the mono, it uses ***blockConv1()*** for mode 0, and ***effBlockConv()*** for mode 1.

The ***mixer()*** function uses a multiplier to mix the extracted signals and NCO output, and the ***combiner()*** function includes two dividers, one adder, and one subtractor.

Thus , for mode 0, the number of multiplication or accumulation for each output sample would be the number of multiplication or accumulation in mono mode 0 (304) plus 101(stereoBPConv())*2+1 (fmPLL())+1(mixer())+101(blockConv1())+1(combiner()) = 610; for mode 1, the total number of each output sample should be the number in mono path plus 101*2+1+1+101 (effBlockConv())+1=610. The stereo part also includes one ***atan2()***, one ***cos()***, and one ***sin()*** in ***fmPLL()***, and one ***cos()*** and three ***sin()*** in the bandpass filter.

The runtimes for ***stereoBPConv()*** in mode 0 and mode 1 should be similar to ***blockConv_I()*** in mode 0 and mode 1. The total runtime in the stereo path for one block should be longer than the RF Front-End path since the stereo path includes an ***fmPLL()***, a ***stereoBPConv()***, a ***mixer()***, and a

combiner(), and the tested runtime is (2.15831 ms + 0.764904ms)/2560 = 1.142e-3 ms/sample, or (1.90321 ms + 0.764904 ms)/2500 = 1.067e-3 ms/samples.

Stereo Main Functions Runtimes:

Mode 0: ***stereoBPConv()***: 0.354032 ms
fmPLL(): 0.305373 ms

Mode 1: ***stereoBPConv()***: 0.300755 ms
fmPLL(): 0.2596 ms

RDS

The RDS path implements mode 0 only, and has the most computations among three paths. The blocks leading up to RDS demodulation are very similar to those in the stereo path, except different frequencies are used for the PLL, and the input signals are run through a squaring non-linearity block, which multiples the value by itself, in the RDS carrier recovery part. The mixer is followed by a 3 KHz LPF, which is combined with an upsampler with a factor of 19, and has 101 computations or accumulations in ***rdsLPFConv_3k()***. Due to the design of upsampling by 19 and downsampled by 80, the LPF coefficients used in the 16 kHz LPF after the buf_wave is used up is int(101*19/19) = 101. The RRC filter does the convolutions one by one, while it multiplies by 19 for each output result. Therefore, each output bit after the RRC filter would have RF front-end numbers (101*2) + 101*2 + 1(squaring) + 1(PLL) + 1(mixer) + 101(3k LPF) + 101(16k LPF) + 101(RRC) = 710 computations.

The last part which needs to do calculations for the output results is the frame synchronization. There is a matrix multiplication in ***frame_sync()***, which needs to multiply a 1*26 matrix to a 26*10 matrix. Thus, the total amount of computations for each output bit is 710+10=720.

The ***rdsLPFConv_3k()*** implementation is the same as the ***stereoBPConv()***, so the runtime should be very similar. The ***rdsRecBFPConv()*** function has one more operation than ***rdsLPFConv_3k()***, which multiplies the results by the factor of upsampler, 19, due to the separated signals energy by upsampling, so the runtime should also be closed to ***rdsLPFConv_3k()***. In theory, the runtime for the ***rds_16k_conv()*** should be ¼ of the runtime for ***rdsLPFConv_3k()***, however, the actual runtime is 0.183424 ms. This may be due to the additional operations required to choose the corresponding jumping indexes.

The RDS block also includes one ***atan2()***, one ***cos()***, and one ***sin()*** in ***rds_fmPLL()***, as well as one ***cos()***, and three ***sin()*** in the bandpass filter. The total runtime in the rds path for one block should be the longest path since the rds path includes the RDS data processing block, and the tested runtime is (3.11166 ms + 0.764904ms)/2560 = 1.514e-3 ms/bit, the total number of computations per output bit is 1451.

RDS Main Functions Runtimes:

Mode 0: ***rdsRecBPFConv()***: 0.331022 ms
rdsLPFConv_3k(): 0.345997 ms

rds_ fmPLL(): 0.313145 ms
rds_16k_conv(): 0.183424 ms

5 Proposal for Improvement

To benefit the user of our system, a user interface could be developed for users unfamiliar with the command line or linux systems. To achieve this goal, we could use python for tkinter GUI support or Java AWT to develop a user interface that could make calls to execute our code in an easy to use way. To benefit our own productivity when expanding our system, comments can be left on each function that has room to be optimized, so that we can easily find and modify them later to further improve performance.

To improve the runtime of our system, we could split the audio thread into separate threads for mono and stereo processing. Using two threads to implement each part respectively and later combine them together using combiner() function after each computation. Considering the diversity of hardware that users may use our system on if it were widely distributed, some hardware would have poorer

performance than the raspberry pi it has been benchmarked on, which can lead to a gap between each block processing. In this case, adding two threads is especially necessary. To achieve this goal, we can create a thread in our main function called ***mono_thread*** and change ***audio_thread*** to ***stereo_thread***. As both mono and stereo processing need demodulated data, then we will create a new queue ***combine_queue*** in the ***main()*** function for the new thread, and now we have two queues in which to store the demodulated data and outputs of our ***thread_mono*** and ***thread_stereo*** functions relatively, which will be used in ***combiner()***. The logic is the same as the threading implementation in the multi-threading part while the difference is each block of mono thread will wait for the stereo thread that has the same index as the mono queue to finish the implementation, or oppositely (stereo thread would wait for mono thread to be finished), then put the output to ***combine_queue***, which will be used in ***combiner()*** function to combine the mono audio and stereo audio samples and do the following implementations.

6 Project Activity

Week	Task	Contributions
1	Read the project document and refactorized the Python model for mono path to C++	All group members worked together to transfer Python code to C++
2	Optimized C++ mono 1 to implement real-time	All group members worked together to debug and tried several methods to speed up the C++ code
3	Finished mono path and worked on the stereo path	Yuxin and Hongwei built the Python model for stereo path, Declan and Haoran refactorized the Python code to C++. All group members worked together to optimize the C++ code
4	Finished the stereo path and started working on multi-threading	All group members worked together to learn how to implement multi-threading, and tried to find the best way to do it in this project
5	Finished multi-threading and began working on the RDS path	Declan and Haoran built the Python model for the RDS path up to the data recovery, Yuxin and Hongwei refactoriezed the Python code to C++. All group members worked together to debug and build the logic for the RDS data processing blocks.
6	Finished RDS path and started working on the report.	All group members contributed evenly to finish the final report.

7 Conclusion

All of us gained lots of experience in dealing with building Python models, refactoring Python code to C++,a deep understanding of optimizing by reducing the redundant computations to meet the requirements of implementing real-time as well as plenty of practical experience with signal processing concepts. We realized that Python is a powerful and useful tool in developing programs through building the pieces of models. It also helped us learn valuable skills in debugging code to find deep set errors. Finally, we successfully implemented the real-time implementation of a computer system together, and the tested results are agreed with the output types and aplay sound. We were very appreciative for the help from the professor and TAs.

8 References

[1] DSP Tricks: Frequency demodulation algorithms. https://www.embedded. com/dsp-tricks-frequency -demodulation-algorithms/. Accessed: 2021-04.

[2] pyFmRadio - Radio Broadcast Data System (RBDS) & Radio Data System (RDS). davidswiston. blogspot.com/2014/12/pyfmradio-radio-broadcast-data-system.html. Accessed: 2021-04.