COE3DY4 Project Report

Group 24
Hassan Mahdi - mahdih2@mcmaster.ca
Parthav Patel - patep62@mcmaster.ca
Shaqif Ahmad - ahmes80@mcmaster.ca
Zain Talat - talatz@mcmaster.ca
April 5th, 2022

1 Introduction

The goal of this project is to create an implementation of a software-defined radio (SDR) system to receive frequency modulated (FM) mono and stereo audio in real-time. Along with this digital data through FM broadcast is received using the radio data system (RDS). The SDR is created using a virtual machine emulating a Raspberry Pi. The SDR is composed of C++ code with some python being used. The hardware used is an RF dongle on the Realtek RTL2832U to receive data and a Raspberry Pi 4 to hold the SDR.

2 Project overview

The beginning of this project starts with the data received by the radio-frequency (RF) hardware. The RF is received from the antenna and translated to the digital domain by creating an 8-bit sample for the in-phase (I) component and an 8-bit sample for the quadrature component. The I/Q data is received in an I/Q interleaved format where each 8-bit I sample is followed by an 8-bit Q sample.

Next, the RF front end block extracts the FM channel using a low-pass filter. This data is then decimated to the required intermediate frequency (IF) and then it is put through FM demodulation. There are 4 modes of operation with different sample rates with modes 1-3 being custom for every group. The data is then directed to 3 paths which are mono, stereo and radio data system (RDS)

The mono block extracts the mono audio which is in the 0 – 15 kHz section of the FM channel. This data is downsampled or resampled to 48 KSamples/sec or 44.1 KSamples/sec depending on the mode of operation. The output from this block is in 16-bit signed integer format that can be passed onto an audio player.The stereo block extracts the 19 kHz pilot tone with band-pass filtering so the downconversion of the stereo channel can occur. This data is then filtered and combined with the mono audio to create the left and right audio channels as the mono channel has the sum of the left and right channels while the stereo channel has the difference between the left and right channels. The input and output are in the same format as mono.

The RDS block recovers the subcarrier from the RDS channel. The data is downconverted and resampled, then clock and data recovery are performed. The bitstream is generated and the frame synchronization happens in the data link layer. Finally, the extracted bits are passed off to the application layer. The input is the same as the previous blocks, but the output is now bits, data frames and radio text.

3 Implementation details

3.1 Lab work + RF Front End

When implementing the RF Front End, building blocks from previous had to be developed. One key building block that was developed in labs was impulse response generation. The impulse

response generation was used multiple times in the RF front end as well as other threads. The initial implementation of the low pass filter was created in Python in the lab. This was done by creating our custom function which calculates the normalized frequency and then sums the results of the sinc function for every tap. The result for every index was then multiplied by $sin^2(\frac{i\pi}{\#ofTaps})$, the result of this function would generate the impulse response. Further in the labs, this Python function was refactored into C++, this refactored C++ low pass filter was crucial in our implementation of mono, stereo, RDS and RF front end. Another crucial building block learned from our labs was block processing. To process data in real-time the input data was divided into "blocks", the issue with this block method was that there would be disruption when traversing between blocks. In lab, we created a function known as *block_processing* which was tackled in Python. We added a state-saving mechanic which would allow us to save data from the previous block to the next block and perform convolution. The initial Python model required the state to be stored in an array using array slicing. When refactoring this function into C++ issues occurred as there was no known built-in vector slicing for C++, thus a separate function was created that would slice a vector manually to tackle this issue.

With the building blocks in place, implementation for the RF front end began in Python with *fmMonoBlock.py*. Initially, the coefficient for the RF front end is generated using the low pass filter (*impulseResponseLPF*) that was developed in the lab as a building block, these RF coefficients would be later used to extract the IQ data from the input data stream. This was similarly done in the Python model of RF front end as SciPy's *firwin* was used to extract coefficients. A function known as *readStdinBlockData* is used to read in values from a data stream file, the function would separate both I and Q into separate vectors to allow for further processing. To extract the I and Q data, the data has to go through a lowpass filter and then be decimated by a certain scale factor. Our implementation consisted of using the *block_processing0* function which consists of a convolution + state saving function from the previous lab with optimized downsampling (explained in 3.2 Mono Implementation). In order to receive the original signal the data must be demodulated, this is done through FM demodulation. Initially, the Python model used a function known as *fmDemodArctan* for demodulation, however during the lab a faster version was developed. This faster implementation was developed by using the equation $\frac{I\Delta Q - Q\Delta I}{I^2 + Q^2}$ instead of using arctan. To obtain ΔQ and ΔI, the current I and Q values were subtracted by the previous I and Q values. Once this function was validated in *fmMonoBlock.py,* it was refactored into C++ to allow demodulation.

3.2 Mono Implementation

When implementing the mono path, heavy consideration was taken from the modelling part of mode 0 which was done in Python. The implementation of mode 0 consisted of refactoring parts of fmMonoBlock.py to C++, this Python implementation allowed us to verify the correctness of the mono path as well as aid us in the development of the mono path in C++.

When initially implementing the mono path, the SDR checks which mode it is operating in, depending on the mode various variables will change such as the upscaling factor, downscaling factor, sampling rate, rf_FS and rf_decim. When initially developing mode 0, the function *block_processing0* was used for convolution and downsampling to produce the desired outcome. The initial process was to use the convolution + state saving function from previous labs and implement a separate downsampling function that would be called from the initial convolution function. However, while testing this out using the samples0.raw file on the virtual machine we received multiple "underrun" errors. Furthermore, the audio would often be choppy and cut off while running the test audio. After deliberation and discussions with TAs, we noticed that this error occurred due to the fact that our downsampling method was too slow in terms of runtime, thus causing underrun errors. This issue was solved by implementing downsampling into the convolution function, this was done by iterating by the value of the downsampling factor until the end of the yb.size(). Our initial thought process behind this was that we would only calculate every yb[n*decim] (multiples of decimation factor) value since the values of other yb would not matter when downsampling. Despite this fix, we received very poor

audio which did not resemble the sample audio, although the underrun issues were fixed. After further deliberation, we realized we were doing the calculations incorrectly and instead of going through every yb[n*decim] we should be going through every xb[n*decim] and to increment yb a separate counter was created. This optimized C++ implementation was written in the *block_processing0* function and was vigorously tested. After comparisons between Python sample audio, original sample audio and comparing output plots between Python and C++ we were able to verify that the implementation of mode 0 was correct.

Since mode 1 does not require upsampling, *block_processing0* was used to test this mode albeit the values of this mode were different compared to mode 0 so, different values of rf_FS and rf_decim were used. Testing was done by converting samples0.raw file to 1440 Ksamples/sec using the *fmRateChange.py* and comparing original output audio to the C++ output audio.

Implementing modes 2-3 came with some difficulty, since these modes required upsampling the original *block_processing0* function needed to be revamped to include upsampling before convolution and downsampling occurs. Initially, our thought process was to create another vector Xu which would hold the upsampled values, this vector would be fed into the convolution/downsampling function instead of the original vector Xb. This was done by initializing the vector Xu to all zeros while creating a for loop that would set the value of Xb[n] to every multiple of the upsampling factor Xu[n*u]. Multiple challenges occurred with this implementation, such as an aplay error. This error occurred due to the fact that many vectors were implemented without taking into account the upsampling factor. For example, the block_size did not take into account the upsampling factor and was not the adequate size thus causing an aplay error. The block_size vector was fixed to incorporate the upsampling factor, it was seen as *Block_Size = 1024 * rf_decim * mono_decim * 2 / mono_upscale.* Once this issue was resolved, the audio was able to play but resulted in choppy audio with underrun errors similar to the initial implementation of *block_processing0*. Since we already dealt with this issue before, we knew it was because the runtime of our function took too long. This was solved by creating a function *block_processing1* which incorporated upsampling into convolution without creating an extra vector Xu that would pad zeros. The implementation of *block_processing1* ignores any padding with zeros instead, working with the original data. The convolution only performed necessary calculations, because we know that we only need to retain output samples of upscaled vector yb at intervals of the downsampling factor and we know that any input of vector xb that isn't a multiple of the upsampling factor will result in 0. This allowed us to save a lot of runtime as we only performed necessary calculations and forgo any padding with zeros. However, after testing *block_processing1* it seemed as if the audio was slightly distorted and had some crackling, to us this meant that there was a small issue when doing the calculations for convolution. This issue was solved by adding a phase tracking to the impulse response as shown in the lectures. Once solving this issue, testing was done by comparing audio to original audio as well as plotting the output and we verified that all mono modes worked correctly.

3.3 Stereo Implementation

The most important building block in building such a system is deriving the impulse response coefficients of a filter based on the sampling frequency, the cutoff frequency and the chosen number of taps.

When choosing the number of taps for the FIR filters in our system, our decision-making depended on a few factors. Choosing a very large number of taps increases the number of multiplications and accumulations that need to be done during the convolution process. The audio will have an increased signal quality however it will severely diminish the performance of the system, especially when running it in real-time on a conservative computer such as the Raspberry Pi 4. Therefore we chose a number of taps equal to 151 for all of our low-pass filters as well as bandpass filters for phase delay reasons to be further discussed below.

During the initial phase of stereo implementation, we studied the bandpass filter algorithm outlined by the project description and implemented it straight into C++. It was straightforward so we decided to not model it in python to test its correctness. We first needed to extract the stereo signal that is modulated onto a 38 kHz subcarrier, which is locked to the second harmonic of the 19 kHz pilot tone. We extracted the pilot tone with fb = 18.5 Khz and fe = 19.5 KHz. We then extracted the stereo signal with fb = 23Khz and f3 = 53 Khz. We then needed to synchronize the carrier frequency with the subcarrier frequency used by the mixer to perform the DSBC modulation technique. We synchronize the 19KHz pilot tone using a phase-locked loop (PLL) scaled up 2 times using a numerically controlled oscillator. Now once we have a clean output signal that is filtered and amplified we pass it through the mixer. We perform downconversion by mixing the recovered carrier and the stereo channel; which is a low-demand task as we are doing pointwise element multiplication that is scaled up by 2. The output of the mixer will follow the same mono path mentioned above. The last thing we had to do was extract the Left and Right channels of the audio through a recombiner. To extract the Left channel we had to add the mono audio block and the stereo audio block pointwise element by element and divide it by 2 due to trigonometric identities which scale down our output. To extract the right channel we had to subtract the mono audio block and the stereo audio block pointwise and scale it down as mentioned above.

Implementing the stereo step by step was not too difficult initially. However, we ran into a problem when listening to the output using the special stereo.raw file which Dr. Nicola prepared for us to hear a unique sound in each ear to be able to verify our work. Both unique audios were being heard from both channels and the issue was that this problem could have been caused by the many different stereo implementations including the bandpass filter, the carrier recovery through the PLL and NCO, the mixer or the combiner. Moreover, it could have been an issue of sizes. However, a TA showed us that we needed to implement a phase delay using an all-pass filter for the mono path as a direct result of the delay introduced by the two bandpass filters that extract the 19 kHz pilot tone and the 23-52 kHz stereo channel.

As seen in figure 1.0 in the Stereo Appendix we carefully introduced an all-pass filter in the mono path to synchronize the mono and stereo audio blocks for recombining. The implementation of the all-pass filter was quite simple, we had to rearrange the input array and introduce the delay using state saving techniques. The output of that filter, called output_fm_demod was used by the mono path. And the original fm_demod was used by the stereo path. This way when the mono path and the stereo path will be synchronized when met at the recombiner. Furthermore, the number of taps for the all-pass filter was (N-1)/2 in order to match the two bandpass filters.

Even after introducing the phase delay, we were still getting mixed audio signals in both ears which prompted us to think that we need to shift our focus to inspect closely the vector sizes of the input arguments to filter PLL and other stereo processing functions. Before that, we modelled much of the stereo processing in python to make sure it had the correct implementation which it did. We made sure the stereo and mono had the same amount of taps and things that matter.

After quite a bit of debugging, we found the little bug. We introduced a very wrong sampling rate into the PLL which ended up messing up the synchronization process. The input sampling rate we inputted initially was mono_audio_fs which was 48Khz for mode 0. However, we quickly realized it should have been equal to the rf sampling rate divided by the rf decimator which is equal to the fm_demod sampling rate rf_Fs/rf_decim. After that, both audios played individually on separate channels.

3.4 RDS Implementation

Threading: A slight adjustment had to be made to the threading process with the addition of RDS which is a second consumer thread. Naturally, both consumers cannot be reading from the same queue since they require identical data sets. Therefore, a separate queue was added for the RDS thread

to read from. The RF or producer thread would then push the same FM demodulated data into two different queues at the same time. Upon testing, it was made clear that still, the consumer threads were receiving different blocks of data. What we realized was happening is that the producer thread had not yet been synchronized with the RDS thread. This issue was solved with the addition of a condition to the producer thread's mutex locking system. We had to make sure that new blocks of data would not be pushed into either queue until both queues were empty. We then were able to confirm that the RDS and mono/stereo threads were both receiving identical data sets.

Implementing the RDS up to the clock and data recovery portion was extremely straightforward. The necessary processing was made up of mostly the same filters which were used in the mono and stereo paths, with the one exception being the root-raised cosine filter. All other filters (including the PLL and resampler) were heavily tested and we were very confident in their functionality by this point. Since the pseudocode for the RRC filter was given, developing the equivalent C++ code was not challenging. However, since we chose not to model in Python prior, we ran a quick test to validate its functionality. We outputted a GNU plot of the pre-RRC data and compared it to a plot of the post-RRC data. The signal coming out of the RRC filter was much stronger with the peaks and troughs being very easily identifiable, as opposed to the signal prior to RRC filtering, which was extremely weak. Furthermore, it was clear that the location of these peaks and troughs aligned very nicely with our given SPS rate.

The real challenge began with the implementation of CDR. We would just like to preface this part by stating that due to time constraints, we were not able to fully complete and verify this section, along with frame synchronization. We will outline exactly where we felt problems could occur, as well as how we might go about resolving these issues if we were a little better at managing our time.

Upon inspection of the very first block of plotted pre-CDR data, the first problem presented itself very quickly. The incoming signal started out extremely weak. The first ~200 samples were not strong enough to identify any sort of sampling point. This issue was consistent with multiple different raw files. We were not quite sure if this was expected behaviour or if something had gone wrong with the RRC filtering, but after further testing, we went with the former. It was without doubt that the first block of data had to be analyzed separately to find the exact sampling point. In order to find the first sampling point, some arbitrary magnitude had to be selected as being significant enough to be considered a symbol. After taking a quick look at pre-CDR plots of various raw files, we chose 0.25. Through a for loop, the first value with a magnitude equal to or greater than 0.25 was found, and its index was saved as the starting point for another for loop. This second for loop would then analyze the points following the starting point to identify exactly where the direction of the signal changes (from increasing to decreasing or vice versa). Once this point is found, its index is saved as the first sampling point.

Before the sampling point is passed on, a modulo of the SPS is applied to align it correctly with the beginning of the next block. The CDR function is passed this sampling point, which it uses as the starting index in its for loop. The loop increments by a factor of the SPS rate, and relies on the fact that this interval provides an accurate location of the next symbol. Every iteration of this for loop assesses the current symbol against the next symbol and will output a 1 for a HL or a 0 for a LH. If the value at a sampling point is above zero, it is considered HI or else is considered LO. This is as far as our implementation goes for CDR, and there is one major problem that could arise as a result, invalid bit sequences. Through further inspection of plotted pre-CDR data, we realized that there are certain points in the signal at the sampling point that are extremely close to zero, and could be interpreted as a HI or LO. These special cases should have been assessed by looking at the previous sequence of symbols to figure out what the correct interpretation is. However, no check for this has been implemented, and the CDR function relies solely on the fact that the chosen sampling point will hold true throughout the entire signal. Furthermore, this issue cascades into causing state-saving issues as well. We assume that at the beginning of any block of data, the previous block was completed with a pair of symbols (either HL or LH). However, this is not always the case as a signal may end with just a HI, to which the next symbol must be paired from the beginning of the next block, and not start its

own pair. As a result, the last symbol of every block must be saved for the next block in case it is needed to pair. These two issues can cause very large sequences of invalid bitstreams.

4 Analysis and measurements

Assuming all filters run on N_taps = 101, the following mathematical analysis was done:

Number of MACs/Sample for each mode, for each path.

	Mode 0	Mode 1	Mode 2	Mode 3
Mono Path	1111	1111	1200	1420
Stereo Path	2242	2242	2422	2860
RDS Path	56214	N/A	59058	N/A

* See Appendix for written hand calculations

Runtime Measurements:

IF Time Measurements

	Mode 0 (ms)	Mode 1	Mode 2	Mode 3
In-Phase(I) LP Filter (Fc = 100 KHz)	2.42229	2.49195	2.84785	2.8762
Quadrature(Q) LP Filter (Fc = 100 KHz)	2.39216	2.51991	2.53254	2.86937
FM_demodulator (240 KSamples/sec)	0.073573	0.061277	0.072278	0.068574

The I/Q filters for modes 0 and 1 perform similarly in terms of speed with mode 1 appearing slightly slower. The block sizes for each mode are calculated based on the upsampling and downsampling factors in order to end up with as many whole numbers as possible after processing is complete. The input sampling rate is lower in mode 1, and as a result, a lower decimator is used for the filters. However, since the block size is changed accordingly, the number of output samples from the filters remains the same as in mode 0. Therefore, theoretically, we should be seeing very similar runtimes for modes 1 and 2. With modes 2 and 3, we have the introduction of an upsampling factor, in addition to a downsampling factor. These new factors are applied at filters in the later stages of the processing cycle, but still, cause changes in the block sizes. As a result, the number of output samples from the I/Q filters increases for mode 2 and even more for mode 3. Increases in runtimes for modes 2 and 3 are seen to reflect these conclusions.

Live Measurements Notes/Important:

The first two blocks in each mode for the RF-> fm_demod are significantly slower than the rest of the block sizes.

Mono Time Measurements

	Mode 0 ms	Mode 1 ms	Mode 2 ms	Mode 3 ms
LPF Fc =16Khz	2.4206	2.43712	2.45431	2.91316

When it comes to the final LPF for the mono audio, the number of output samples to be processed by the LPF will be the constant for all modes (1024 in this case). This is where the downsampler and resampler will ensure that during low-pass filtering, unnecessary data is not being processed. Therefore, we should ideally see very similar runtimes for all four modes.

Stereo Time Measurements

	Mode 0 ms	Mode 1 ms	Mode 2 ms	Mode 3 ms
Stereo Carrier Recovery Band Pass Filter 18.5 Khz to 19.5 kHz	2.40908	2.41133	2.40951	2.89062
Stereo Carrier Recovery PLL	1.17065	1.16138	1.17134	1.38965
Stereo Channel Extraction Band Pass Filter 22Khz to 54 kHz	2.4206	2.43712	2.45431	2.91316
Stereo Processing (Mixer+LPF+Combiner)	0.52555	0.530291	3.17369	3.26217
Total Time	6.52588	6.540127	9.20883	10.4556

Following the same thought process, we see an increased number of samples during intermediate filtering, which creates increased runtimes for modes 2 and 3. These modes require different block sizes due to resampling factors, which creates the larger sizes of intermediate data.

The impact of the number of filter taps on the runtime Mode 0:

	Number of taps = 13	Number of taps = 101	Number of taps = 301
Mono LPF Fc =16Khz	0.360348 ms	2.42321	4.86574
Stereo Carrier Recovery Band Pass Filter	0.360422	1.69057	5.18312

18.5 Khz to 19.5 Khz			
Stereo Carrier Recovery PLL	2.11657	1.18686	1.25256
Stereo Channel Extraction Band Pass Filter 22Khz to 54 Khz	0.360348	2.42321	4.86574
Stereo Processing (Mixer+LPF+Combiner)	0.160146	0.394236	1.05149

The number of filter taps dictates largely the time it will take for any single filter to run through all of its convolutions by determining the size of the impulse response. A larger impulse response will result in that many more MACs per output sample. As such, this is exactly what we see in the measurement results above.

5 Proposal for improvement

Debugging and validation is oftentimes the most difficult part of engineering a complex system. As we were developing this system step by step, we would utilize a set of runtime measurement commands whenever adding a new component (mono path, stereo path, RDS) to ensure that it would run at the required output sampling rate in real-time. However, having to consistently write these new commands every time we wanted to test became quite cumbersome over time. A function could be added that would essentially perform this task, and print to the command line whether or not the software would run in real-time. The function would be broken down into two sub-functions that would be called at the start and end of the target software. The function would take in the input and output sampling rates, compare them against the measured timing values, and output the results. These results could include exactly how much slower/faster the software is, which could help with debugging and most importantly, save time. In terms of improving the user experience, a complementary bash script can be written to execute the software. This script would provide much more selection to the user without having to directly edit the commands. The user could adjust the aplay, cat, and SDR parameters, as well as the mode of operation quickly and easily.

When it comes to optimization, there is room for improvement as well. Firstly, we can use the fast Fourier transform implementation instead of the discrete Fourier transform that is being used now. Additionally, depending on the number of processors available in the hardware, extra threading can be added to further partition the RDS demodulation. Lastly, certain parameters were arbitrarily selected. For example, the chosen block size or threading queue size can be tweaked slightly through careful consideration and testing.

6 Project activity

	Activity Detail
Week 1 Feb 14	**Hassan**: Received the RF Dongle and Hardware **Shaqif, Zain, Parthav:** N/A
Week 2 Feb 21	**Everyone:** N/A
Week 3 Feb 28	**Hassan**: Reviewed Lab and class material **Parthav**: Began to review lab and lecture material. **Zain**: Reviewed Lab and class material **Shaqif**: Reviewed Lab and class material
Week 4 Mar 7	**Hassan**: Reviewed Lab and class material, Setup Hardware and SDR tests **Parthav**: Worked on fully refactoring the lab 3 Python code into C++ (mode 0). **Zain**: Reviewed Lab and class material **Shaqif**: Reviewed Lab and class material
Week 5 Mar 14	**Hassan:** Fixing and refactoring the PSD function from lab 3 into C++ **Parthav**: I started with debugging and testing the refactored C++ code for mode 0 only. Upon completion, I began to work on the stereo implementation for mode 0. **Zain**: Assisted in developing Mode 0 and refactoring fmMonoBlock.py to C++. Tested Mode 0 using Gnuplot. **Shaqif**: Reviewed Lab and class material
Week 6 Mar 21	**Hassan**: Started implementing and debugging and refactoring some of the C++ code from lab 3 into the project. Including PSD functioning, however, the problem is DB seems lower than what it should be. Testing Mono Mode 0 real-time and live **Parthav**: This week was spent on implementing threading with the newly developed stereo implementation. I then spent the latter half of this week debugging and testing the stereo and mono paths with modes 1-3. **Zain**: Continued to work on Mono Mode 0-1 to optimize and debug, Started developing a slow version of resampling for modes 2-3 and continued to watch lectures to understand how to develop efficient upsampling. **Shaqif**: Testing and debugging mono mode 0 and started developing mode 1-3
Week 7 Mar 28	**Hassan**: Worked on stereo implementation and debugging. Modelled most stereo functions on python for debugging purposes. Attended TA, and Dr. Nicola meetings for help. After fixing the Stereo I started working on frame sync. Final Live testing of all modes and final modes debugging. **Parthav**: Worked through implementing the RDS path directly into C++. I spent most of this week on the CDR portion. **Zain**: Developed optimized version of resampling and tested, ensured that all mono modes worked efficiently, implemented mono modes into stereo and assisted in debugging stereo then worked on frame sync. **Shaqif**: Finished implementation of mode 1-3 working in real-time
Week 8 Apr 4	**Hassan**: Worked on project report, stereo implementation, and recorded the requested data into tables. **Parthav**: Worked on the RDS implementation, analysis for the measured data, and the proposal for improvement. **Zain**: Worked on Mono Implementation, Lab + RF implementation and Conclusion **Shaqif**: Worked on introduction, project overview and revised mono implementation

7 Conclusion

To conclude, over the past 3 months we were given multiple opportunities to expand and define our knowledge regarding multiple topics we have learned from the past 3 years. These opportunities were given through our labs as well as this project. In this project, we were able to enhance our knowledge of Fourier, digital filtering and digital signal processing and apply it to a real-world application. Throughout this learning experience, we have developed an even better understanding of how to optimize, debug and write efficient code to produce our desired outcome. All in all, this project allowed us to appreciate how much complexity goes into something that we consider "small" in our daily lives and will help prepare us for our future endeavours in the real world.

8 References

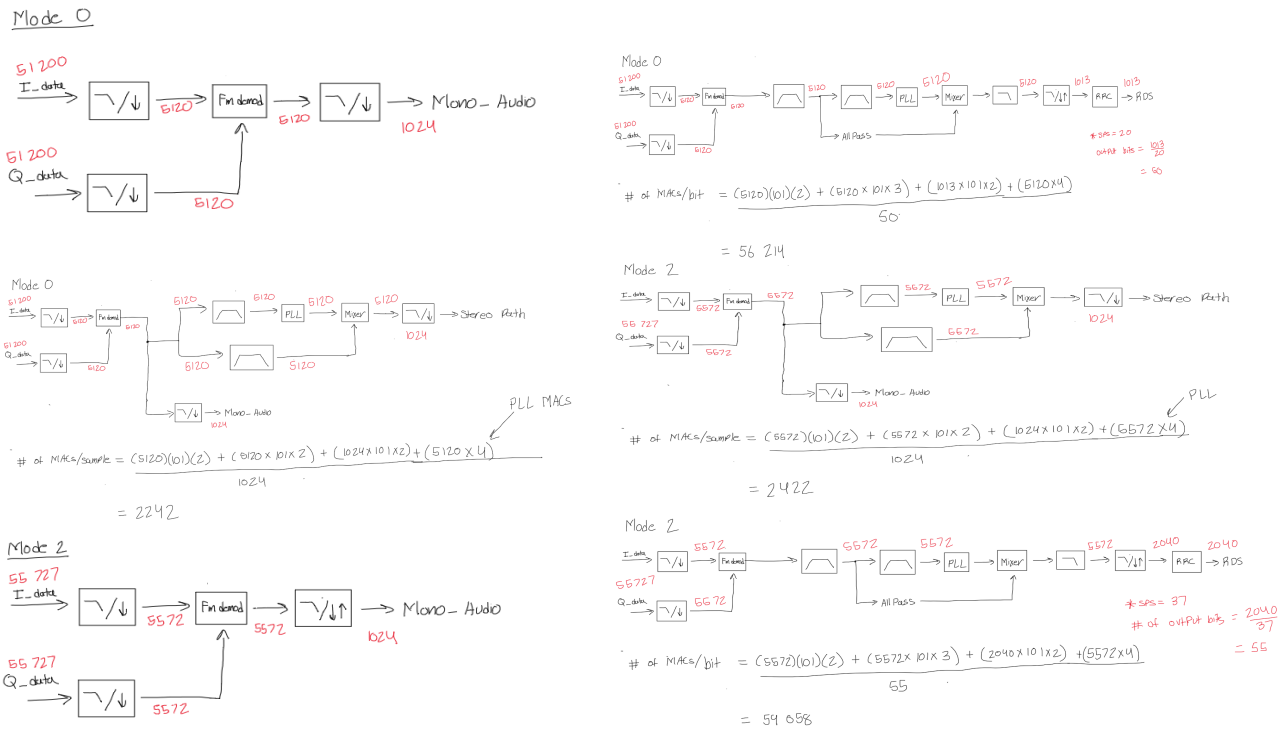
"The C++ Resources Network," cplusplus.com. [Online]. Available: https://www.cplusplus.com/. [Accessed: 08-Apr-2022].

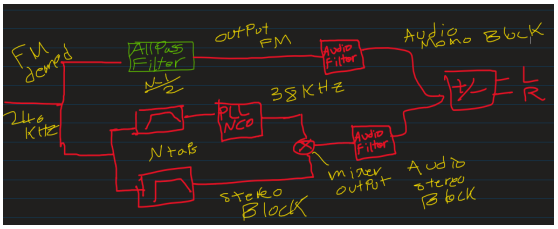
NumPy. [Online]. Available: https://numpy.org/. [Accessed: 08-Apr-2022].

Project Specs

Class Notes

9 Appendix





Mode 1

Block size = (1024)(6)(5)(2)

= 61 440

Intermediate sample numbers have not changed,

∴ still 1111 MACs/sample.

Mode 3

Block size = (1024)(4)(320)(2) / 441

= 53 498

# of MACs/sample = (6687)(61)(2) + (1024)(101) / 1024

= 1420