COE3DY4 Project Report

Group 28
Mohamed Negm, Ahmed Allam, Hushaam Tariq, Syed Shakir
negmm@mcmaster.ca, allama2@mcmaster.ca, tariqh10@mcmaster.ca, shakis1@mcmaster.ca
April 8, 2022

Introduction

The objective of this project is to design a software-based radio that can capture mono audio, stereo audio and RDS data from a frequency modulated signal in real-time on a Raspberry PI equipped with a Realtek RTL2832U based RF dongle. The project scope focuses on consolidating and practically applying a variety of computer and electrical engineering concepts such as understanding signals in the frequency domain, signal modulation/demodulation, resampling, finite response filter design, phase-locked loops, optimizing software algorithms, and leveraging multi-threading to increase performance.

Project Overview

A software-defined radio processes radio data by defining components such as filters, demodulators, resamplers and mixers in software, as opposed to using the traditional hardware-equivalent components. In our specific project we are focusing on frequency modulated (FM) data. In FM, a message is carried by a sinusoidal carrier signal, where as the strength of a message signal grows the frequency of the carrier signal increases. When the strength of the message decreases the carrier's frequency decreases. The RF dongle serves as a starting point for our radio and provides input to our software as an I (in-phase component) and Q (quadrature component) sampled at the center frequency of the FM channel. To process this data, the first step is to low pass filter the RF data around the center frequency to isolate the FM data as the RF signal can be much larger than the FM bandwidth. The data can then be down sampled to the required rate, and finally the IQ data is can be input into an FM demodulator where the message signal is extracted by monitoring the rate of change of the phase modulated FM signal while using IQ samples to generate phase information ($\tan^{-1}(Q/I)$).

To generate the mono audio from this demodulated data, it is first low pass filtered to isolate the mono data range (filter to 16Khz) and then down sampled or resampled to the required output rate. To generate stereo audio, bandpass filters must be used to isolate the stereo pilot tone (18.5KHz to 19.5KHz cutoffs) and stereo audio data (22KHz to 54KHz cutoffs). Once the pilot tone is isolated, it is fed into a phase locked loop and a numerically controlled oscillator to generate a 38KHz sinusoidal signal. This is the stereo carrier, and it is then mixed with the extracted stereo audio data and amplified by a factor of two. Once this data has been mixed, it is then low pass filtered to 16KHz (the same bandwidth as the mono audio data), and down sampled or resampled to the required output rate. To get two audio streams the stereo data is added or subtracted from the mono data to get left and right audio respectively, as the mono data is a sum of the two sides. The mono data used for recombining must be fed through an all-pass filter to match the phase delays of the extra bandpass filters used in the stereo path. In the RDS path, the RDS data is extracted via a bandpass filter (54KHz to 60KHz). This data is squared, bandpass filtered from 113.5 to 114.5KHz, and fed into a PLL with NCO scaling of 0.5 to recover the RDS carrier. On a separate path the RDS data is all pass filtered to match phase delay and then mixed with the recovered RDS carrier. This data is low pass filtered to 3KHz and then resampled to the RDS symbol rate. The signal is then fed into a root raised cosine filter to cancel intersymbol interference, and the data is recovered by choosing a suitable sample and SPS rate. Finally, this signal is differentially decoded to get binary data. This data is frame synchronized by multiplying it with a parity array and checking the resulting syndrome to see if it matches an offset type. Once the frames are aligned and the beginning/ending of a block are recognized, data can be extracted from each block according to the groups in the RDS standard.

Implementation Details

The labs served as theoretical and experimental stepping stones for this SDR project. The first lab introduced the basics of digital signal processing (DSP) such as the Discrete Fourier Transform(DFT), Inverse DFT(IDFT), lowpass/bandpass filters, and block processing. Working in python, functions for impulse response coefficients, convolution, and low-pass filtering were created for processing the data in a single pass. Using lfilter, the single pass processing was turned into block processing. To reduce the

number of artifacts produced through discontinuities in block processing, a new convolution function was created to replace lfilter. A solid understanding of frequency spectrums and their corresponding signals in the time domain was also developed and was used to check the functionality of the filters. It was observed that the greater the number of taps, the better the filter worked but the lower the accuracy of the magnitude became.

The second lab focused on implementing DSP (Digital signal processing) primitives in C++. A majority of the lab consisted of refactoring the python code from lab 1 to C++ including impulse response, low-pass filtering, and block processing. The standard vector class was introduced as the primary means to store the audio data. GNU plot was also introduced here as a method to visualize time domain signals and magnitude spectrums for trouble shooting the software.

The third lab dove deeper into the core concepts of FM modulation where FM channel samples were broken down into their I (in-phase) and Q (Quadrature) components and processed to produce the intermediate frequency (IF) samples for mono audio. Using the lowpass filter function from lab 2, the channel for each component was filtered with a cutoff frequency of 100 KHz to extract the FM channel at the center frequency for RF sampling and then down sampled by 10. This reduced the input sampling rate of 2.4 Msamples/sec to 240 Ksamples/sec. At this point, the choice was made to do computation on all the samples rather than only the samples that would be kept after decimation as the computational cost of filtering did not greatly affect the quality of the output for this lab. Demodulation was done initially through the provided fmDemodArctan function but was replaced with a faster function where the derivatives are instead calculated as the differences between consecutive samples. To account for transitions between blocks, state saving was implemented by returning only the previously used I and Q values as that was all that would be needed in this fast demodulation method. The power spectrum density (PSD) of each processed block was used as a means of verification for the C++ code to ensure a comparable Demodulated FM plot to the python model was produced.

Settings for group 28	Mode 0	Mode 1	Mode 2	Mode 3
RF Fs (Ksamples/sec)	2400	1440	2400	960
IF Fs (Ksamples/sec)	240	288	240	320
Audio Fs (Ksamples/sec)	48	48	44.1	44.1

Figure 1: Assigned sampling rates for each mode

Continuing from lab 3 to the front-end development, the code from the lab was relatively similar to the code needed for the front-end but with the addition of three new modes. For modes 0 and 2, the input RF sample frequency had to be downsampled by a factor of 10, while for modes 1 and 3 it had to be downsampled by factors of 5 and 3 respectively. Float 32-bit representation for the I and Q data was chosen over Double 64-bit because the extra precision, while helpful, would not be greatly beneficial for our implementation. Also, keeping in mind the future need of high upsampling rates in the mono path for modes 2 and 3, the expense of doubling the memory requirements for the blocks was not worth the possible performance overhead.

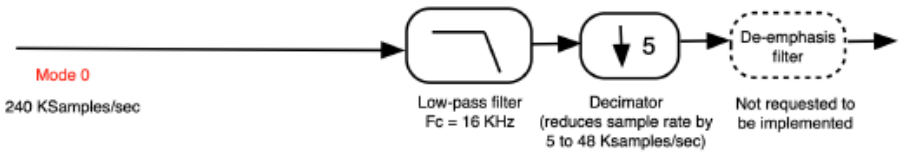


Figure 2: Signal flow graph for mono processing modes 0 and 1

Using the mono audio processing modeled in lab 3, the python code needed to be updated to include downsampling. Since it would be counterintuitive to compute data that is to be discarded, the fast convolution function *convolveBlockFastFIR* was created. This function uses this fact to reduce the number

of computations required by only computing every k-th value, where k is the decimation factor. To avoid aliasing due to the overlap of spectral copies of downsampled data, an anti-aliasing filter is needed prior to downsampling. Since the LPF required for mono processing is much tighter, with a cutoff frequency of 16KHz, the anti-aliasing filter can be ignored.

Moving on, the function was modeled and validated in python using both a control and a real data set. For simplicity, the logging done during validation only displayed the indexes of each of the y, h, and x arrays. Astonishingly, the indexes matched perfectly on the first design iteration, leaving no demand for debugging. As a final validation, the filter output from the fast convolution function was logged and compared with the output of the normal convolution and decimator pair. Finally, the audio output was verified in non-real-time using the audio samples provided. Additionally, the same tests were performed for mono mode 1 at a sampling rate of 1.44MHz. Since the provided test files were all sampled at a frequency of 2.4MHz, the *fmRateChange* python script was used to convert some raw files to the appropriate sampling rate. From here, the python code was refactored into C++ code and identical tests were performed on the refactored code except for the output data comparison. Instead of comparing the fast and slow convolutions, the output from the refactored fast convolution function was contrasted with that of the python model. Once again, hardly any problems were met and the program ran successfully in non-real-time and real-time.

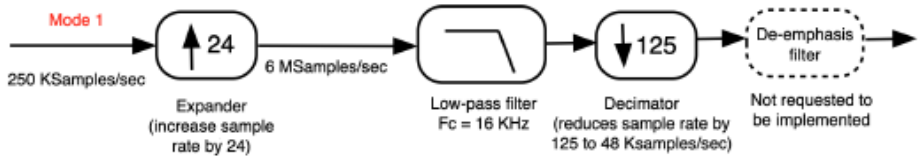
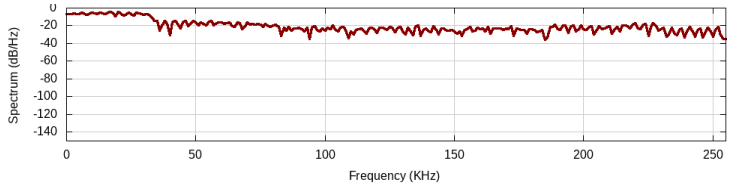


Figure 3: Signal flow graph for mono processing modes 2 and 3

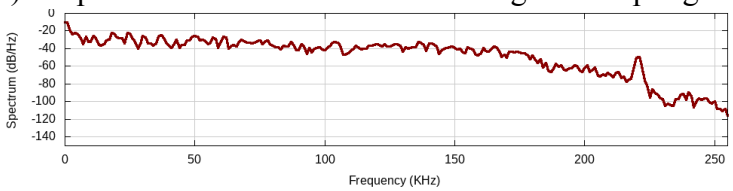
Upon closer inspection of the sampling rates in Figure 1, it can be observed that the scaling ratio between the RF, IF, and audio sampling rates are integer values for modes 0 and 1. However, this is not the case for modes 2 and 3, thus requiring a pre-filtering expander. To determine, the relative expansion and decimation ratios, the greatest common denominator of the IF and audio sampling rates was computed. Then, the expansion ratio U and decimation ratio D were calculated using the equations $U = \frac{Audio\ Fs}{gcd}$, $D = \frac{IF\ Fs}{gcd}$. To remove the extra images created by the expansion, an anti-imaging filter is required. But, once again, the cutoff frequency for the filter is less than that of the mono path filter. Thus all 3 filters were combined into a single LPF with adjusted indexing.

With all of this in mind, the first draft of the resampling function was modeled in python. Just as before, the function was tested with 2 data sets; a control and a real data set. This approach would prove to be successful in the long run as it exposed a major conceptual misunderstanding. The control data set consisted of an input array of size 72, a decimation factor of 3, and an upsampling factor of 4. While observing the resulting indexes, it was observed that the phase, the starting index of the impulse coefficients, was sequentially increasing. This observation was misinterpreted as a rule, thus causing confusion with respect to the real data indexing. As a result, a few steps backward were needed for further analysis and consolidation of resampling concepts. First, the resampling function was put aside and a function that only does upsampling was created. Then, using the same steps taken to implement downsampling for fast convolution, the new function was updated to accommodate downsampling. From here, both data sets were rerun and thoroughly analyzed. It was finally understood that the phase is in fact dependent on the decimation and upsampling ratios. Additionally, the phase can be calculated by subtracting the closest multiple of the upsampling rate from the relative y index. Moving on, the second major conceptual problem encountered was due to the impulse coefficient generation. As described in Figure 3, the filter input is the upsampled data at a frequency of $IF\ FS\ *\ U$. So, for the filter to work correctly, the filter input sampling rate must be scaled by the upsampling ratio. This step was forgotten,

resulting in a scaled-up cutoff frequency that simply allowed most frequencies to pass through. The resulting PSD figures can be observed in Figure 4. Finally, the python model worked reliably in non-real-time and was refactored into C++ code. No problems were met during refactoring and the program was run successfully in real and non-real-time.



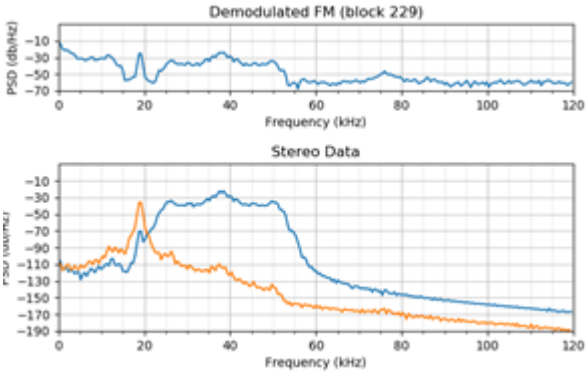
a) Output PSD for a filter without scaling the sampling rate



b) Output PSD for a filter with correctly scaled sampling rate

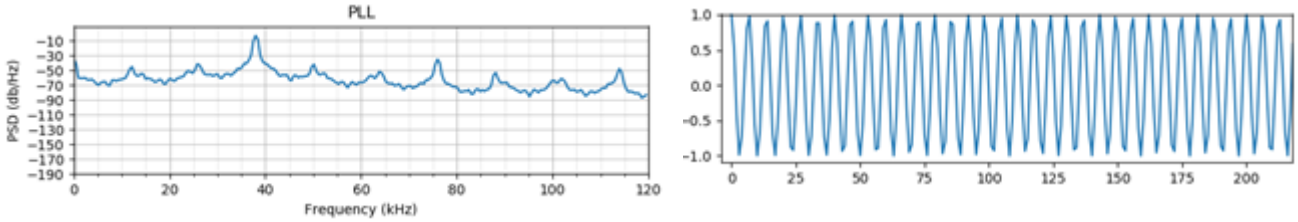
Figure 4: Filter output PSD graphs for a) a filter without scaling the sampling rate and b) a filter with a correctly scaled sampling rate

In order to do stereo audio processing, the stereo pilot tone and the left and right stereo data must first be isolated from the FM signal. Previously in labs, we designed a function to generate the impulse response of a low pass filter and then designed a convolution function to apply this low pass filter to a given signal. This was used to extract the mono audio data from the FM signal as it was all the data in the FM spectrum 15KHz and under. In this case, as the pilot tone and the stereo data exist in between two frequency cutoffs and so cannot be extracted using a low pass filter. Instead, a bandpass filter is required as it can isolate a signal in between two cutoff frequencies. A function called bandPass was created to generate the impulse response coefficients for a bandpass filter given a certain number of taps, sampling rate, a beginning frequency, and an ending frequency. In this function, the normalized pass band is explicitly defined between the ending and beginning frequencies, and a frequency shift is applied by the center frequency between the beginning and ending frequencies. Once the bandpass impulse coefficients are generated, they can be used with the previously created convolution functions used in the mono path. To extract the pilot tone a bandpass filter from 18.5KHz to 19.5KHz was used, and to extract the left and right stereo data a bandpass filter from 22KHz to 54KHz was used. To verify that data was being extracted properly, the PSD of the demodulated FM signal was plotted along with the stereo carrier and stereo data signals after filtering.

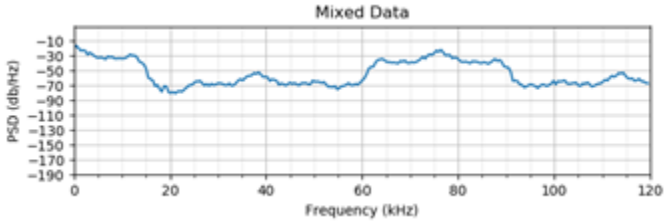


In the PSDs, we can clearly see the 19KHz pilot tone being extracted, along with the symmetrically shaped stereo audio data. The next step in the stereo processing path is to feed the carrier into a PLL. The required output of the PLL and the numerically controlled oscillator was a smooth 38KHz sinusoidal wave. To accomplish this the ncoScale was set to two (2x19KHz=38KHz) and state saving was implemented in the PLL. The values of the integrator, phaseEst, feedbackI, feedback and trigOffset are saved at the end of each PLL function call so they can be used in the PLL function in the next block of data. The last value of the PLL function output (ncoOut) is also saved so that it can serve as the first value of the PLL output in the next block. The functionality of the PLL was tested by plotting the output in the frequency and time

domains to ensure it was a 38KHz sinusoidal wave. The values of the output of PLL at the end of one block were also printed and compared to the outputs of the PLL at the beginning of the next block to ensure state saving was working correctly. The plots below show that PLL and NCO output was a 38KHz sinusoidal wave.



Once the stereo carrier is recovered it is mixed with the stereo channel data. This was accomplished by simply using a for loop to do element wise multiplication of the two vectors. A gain of two was applied to the result of this mixing to make up for the lost magnitude when multiplying the sinusoids. The results for this process were checked by plotting the PSD of the mixer output. The correct output would have the middle of the stereo channel data centered at 0 Hz. As we can see in the plot, the stereo data is centered at 0 Hz and goes up to 15 Hz (the same bandwidth as mono data) so the mixer is working as intended.



Now, the data can be low pass filtered and resampled with the exact same parameters as the mono audio. This stereo data is then ready to be recombined with the mono audio to generate the left and right audio channels. As the mono audio path is a combination of the left and right audio channels, the left channel can be found by adding stereo to mono and the right can be found by subtracting stereo from mono. The mono audio path used in stereo processing was all pass filtered before low pass filtering to match the phase delay created by the extra bandpass filters in the stereo path. The stereo path was tested using synthetic raw files that played different music in each channel to ensure that stereo separation was occurring correctly.

The one major bug encountered in the stereo path was caused by the failure to include an all-pass filter in the mono audio path. This bug was noticed when listening to the output audio, the two channels were not being separated completely. To locate the issue, first the entire stereo path was checked by plotting all the outputs of the bandpass filters, PLL and the mixer. It was clear that the correct data was being extracted, the PLL output was a 38KHz wave, and the mixer produced the required result. So, it was concluded that the error lied in the mono path when combining the data. The lack of the all-pass filter meant that the mono data was out of phase with the stereo data, so adding/subtracting stereo data from the mono data would not separate the channels correctly. After adding an all-pass filter to match the delay introduced by the bandpass filters, the output audio showed clear audible separation of the two channels. As the stereo path requires many convolutions (for front end, mono and stereo filters), it is very computationally heavy. To optimize the load when running live on the Raspberry PI, multithreading was used. The front end operates as a producer thread demodulating data for the mono/stereo processes to use as consumer threads.

Moving onto the most recent implementation of the signal flow graph which focuses on extracting the Radio Data System (RDS) path from the FM audio spectrum. Going through the high-level diagram of

this path, each block was thoroughly supported with reliable test cases and visual aid for verification purposes. The majority of the *RDS channel extraction* and the *RDS carrier recovery* sections required previously configured building blocks from the mono and stereo paths. The theoretical and practical understanding of filtering tools such as a low pass filter (from mono path), a band pass filter, and an all pass filter along with the PLL (from stereo path) have already been verified on the conceptual level with appropriate "output versus input" test cases and when tested for real-life performance with the raspberry pi. Hence, they can be leveraged without the necessity of any further assessment of the functionality of these blocks. With regards to the choice of the number of taps for the impulse response computation, it was crucial to start with a higher number such as 151 taps to ensure accuracy of the output at the cost of time. Once the results were approved, the final number of taps was settled down to 101 to lessen the time stress while remaining within the "safe" region. As an additional testing resource, a PSD plot shown in figure 5 was constructed displaying the extracted RDS channel and carrier with the rest of the spectrum attenuated to a magnitude as low as -150 dB/Hz. The graphing tool was further utilized to compare the signal before and after the Root-Raised Cosine Filtering phase. A visual matching of the resultant plot to figure 13 provided in the lab description suggested a successful design implementation up to this point.

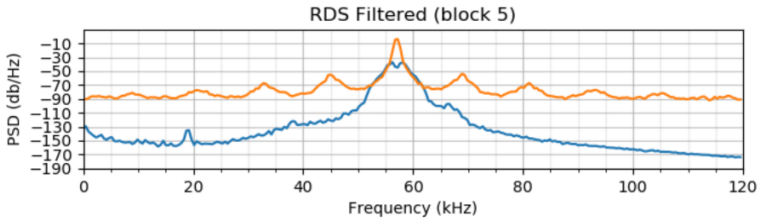


Figure 5 : RDS filtered channel (🟦) and RDS filtered carrier (🟧)

The values of the *phase_adjust* and the *bandwidth* of the PLL were decided based on multiple test runs of the IQ consolation graphs while keeping track of the closest to represent a "realistic signal" similar to that provided in figure 15 of the project description. With values of (⅜)π and 0.002 respectively, figure 6 shows the IQ samples of a randomly chosen block.

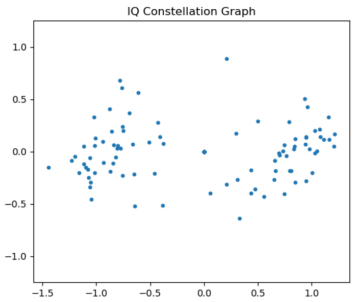


Figure 6: Constellation diagram for a real signal extracted from samples8.raw

To ensure clear separation of the two groups, "bad" sampling points were eliminated from the array during the Clock Data Recovery (CDR) and Manchester decoding phases.To do so, a few design decisions were taken to develop a more realistic constellation graph.

Initially, the case where either three consecutive highs (H's) or three consecutive lows (L's) was appropriately dealt with by flipping the sign of the third sampling point regardless of its initial sign or magnitude. Choosing which of the three samples to flip has its own advantages and disadvantages. For instance, the advantage of flipping the third sample handles the case where more than three consecutive samples of the same sign exist thereby needing to adjust the data only once. On the contrary, it runs the risk of ending up with an "undefined pair" (either HH or LL) for that particular section of the array.

More importantly, dealing with an odd number of samples within a block of data posed a couple of challenges. The procedure followed in this situation begins by pairing the last sampling point from the

current block with the first sampling point of the next block. This means, the new block of samples would technically start at the second sampling point. In the possible case where this pair ends up being "undefined", the decision of manchestering the pair to either binary 0 or 1 is decided by one of the sampling points forcing the other to change its sign.

Finally, within the CDR phase, an *irregular pair detection* algorithm was implemented which checks for HH or LL cases. Figure 7 displays a general breakdown of how the algorithm works. *Note: Another design decision was taken by choosing a limit of 0.3 as the maximum possible value of a sampling point for it to be considered a "bad" sampling point.*

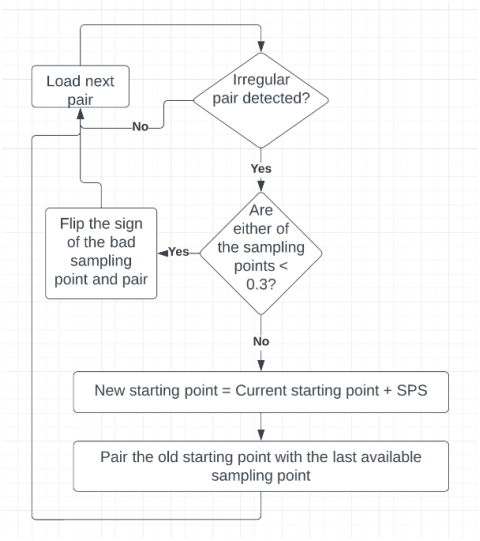


Figure 7: A flowchart of the *Irregular Pair Detection* algorithm

The last configured block for this project's RDS path is the frame synchronization block. After thoroughly designing a multiple variable input XOR gate, the Galois Field matrix multiplication method was first applied to an offset matrix and the parity matrix from the appendix for verification purposes. The results were compared and approved to match the expected syndrome corresponding to the chosen offset. Further testing was carried on by using the manchester decoded data as the input into the frame synchronization algorithm. A few blocks can be recognized, however, there is no clear pattern to the blocks outputted. As per the timing constraints outlined in the project specifications, the group was hindered from debugging the code to isolate the root cause of the problem. However, the group was left with two possible theories that would hint to such action, one being the design decisions taken in the CDR phase, and the other being a more upstream problem that was carried on up to this point. Unfortunately, due to the nature of the project deadlines, it was not possible to refactor the code into C++ as it was more important to continue the debugging process rather than refactoring a non-working code.

Analysis and Measurements

Table 1: Calculated Multiply and Accumulates(MACs) per audio sample for Mono and stereo paths at a fixed accuracy of 101 taps

Mode (Mono)	MAC/audio sample	Mode (Stereo)	MAC/audio sample
0	1111	0	2121
1	1313	1	2525
2	≈1200	2	≈2300
3	≈1567	3	≈3033

Using the measurement summarized in Table 1, there are 3 observations to be made. First, the stereo path has a significantly larger number of MACs. This is due to the fact that stereo processing has 2 bandpass

filters for channel and carrier recovery, as well as an FIR for digital filtering. Thus, the number of MACs per audio sample is much larger. Secondly, modes that have a smaller RF sampling frequency, that is modes 1 and 3, have a larger MAC/audio sample than their counterparts. This is because a decrease in the RF Fs will result in a lower front-end decimation factor, meaning that the arrays passed to the filters will generally be larger. Furthermore, an increase in the FIR input array size will lead to a larger number of MACs per block, thus increasing the overall MAC/audio sample result. Finally, the modes involving resampling have more MACs/audio samples than modes involving fast convolution. In resampling, the input array is upsampled by a factor that is smaller than the decimation factor. Theoretically, this upsampling ratio decreases the effect of the downsampling, thus more input data points will be used and the number of MACs/audio samples will increase.

Table 2: Calculated Multiply and Accumulates(MACs) per sample and per bit for RDS path at a fixed accuracy of 101 taps

Mode (RDS)	MAC/sample	MAC/bit
0	≃1772	≃3544
2	≃1389	≃2778

Starting from the front end, two low pass filters are utilized for the inphase and quadrature components. This is followed by two band pass filters for the channel and carrier recovery of the RDS path. All four filters perform a total number of multiplications equal to the *fm_demod* size * 101 taps (default number). Two more filters are used for the resampler and RRC blocks each performing *output_size* * 101 multiplications per output.

The initial block size used has no effect on the MACs. It is rather only dependent on upsampling (U) and downsampling (D) factors used in the resampler block. The two factors are computed based on the GCD of the input frequency (i.e 240 ksamples/sec) and the output symbol rate. It is observed that the symbol rate is directly proportional to the Samples Per Symbol (SPS). Thus, an increase in the SPS (as in the case of switching from mode 0 to mode 2), results in an increase of output symbol rate. It becomes evident that the ratio (U/D) decreases with an increase in the SPS rate and hence an overall less multiplications required

Table 3: Measured runtimes of mono path blocks for each mode in milliseconds

Mode	I Filter Runtime	Q Filter Runtime	FM Demodulation Runtime	Mono-Path Filter Runtime
0	128.2	126.73	11.56	21.36
1	201.73	203.56	21.07	35.02
2	108.67	106.11	11.41	141.56
3	368.07	367.21	35.31	429.32

Table 4: Measured runtimes of stereo path blocks for each mode

Block/Mode	0 (ms)	1 (ms)	2 (ms)	3 (ms)
I Filter Runtime	111.79	216.267	116.725	369.03
Q Filter Runtime	108.76	222.46	110.22	368.50
fmDemod Runtime	11.57	24.14	12.29	38.22
All Pass Filter Runtime	11.44	15.18	10.31	30.10
Channel Recovery Bandpass FIR Runtime	68.83	126.56	63.26	192.62
Carrier Recovery Bandpass FIR Runtime	65.15	126.04	64.39	194.76
PLL Runtime	307.2	530.28	276.8	845.67
Mixer Runtime	7.83	11.40	6.68	17.07
Mono FIR Runtime	21.52	35.23	122.93	338.27
Stereo FIR Runtime	21.95	35.5	121.42	327.29

As expected, the measured runtimes are directly correlated to the number of MACs/sample for each mode. This is because convolution carries the heaviest computational burden among all the operations required for audio processing. This relationship is clearly visible for both mono and stereo processing and can be observed by comparing Tables 1, 3, and 4.

Table 5: Measured runtimes for RDS path mode 0 at a default 101 taps

Block/Mode	0 (ms)
RDS channel bandpass filter	1105.04
RDS allpass filter	0.06
RDS carrier bandpass filter	1159.69
RDS PLL	25.58
RDS mixer	0.07
RDS resampler	248.68
RDC RRC filter	189.24
RDS CDR	0.25
RDS manchester decoding	0.03
RDS frame synchronization	11.24

Table 6: Measured runtimes for Mono mode 0 at a number of taps of 13 and 301

Number of Taps	MACs per Sample	I Filter Runtime	Q Filter Runtime	FM Demod Runtime	Mono-Path Filter Runtime
13	143	17.29	17.02	3.06	4.60
301	301	371.89	363.44	11.42	78.66

The number of taps in convolution is directly proportional to the number of MACs required per sample. So, increasing the number of taps will increase the number of MACs/sample, which translates to longer runtimes. This behavior is clearly observed in Table 6, where the measured runtimes for each block significantly decreased for Ntaps equal to 13, and the opposite for Ntaps equal to 301. However, this speed-up in computation for a lower number of taps is at the cost of audio quality. The difference in audio quality is severe between all 3 cases. For Ntaps equal to 13, the audio is low in volume with lots of static, whereas, the audio is clear with no static for Ntaps of 301. Also, the difference in audio quality between 301 and 101 taps is not as severe as the transition from 101 to 13. In fact, audio outputs generated by the program at 301 and 101 taps are nearly indistinguishable.

Proposal for Improvement

An improvement that could be made to better user experience is making it easier to switch radio stations. Currently, if the user wants to switch radio channels they would have to stop the code entirely, enter the frequency at which the new radio station is broadcasted at, and then start the SDR again. Instead, allowing for the changing of FM channels while the SDR is still running would make the SDR a more immersive experience. An improvement that could have been made during the development process would be generating a synthetic RDS file with known RDS data. This would have greatly improved the debugging process for the RDS path and would have made it easier to pinpoint where in the path things were going wrong.

A possible method for improving runtime performance of the SDR would be to remove the resize function within for loops with a static sized vector large enough to fit the maximum possible vector size while keeping track of the size of the updated vector through a variable. The time complexity for the resize function is linear [3], being equal to the number of elements inserted/erased, while the method proposed has a time complexity of 1 since only the size variable would have to be updated. Thus replacing resize in a for loop would have a small but noticeable difference on run time.

Project Activity

Week of	Progress And Contributions
Feb 14	-
Feb 21	All: Revisition of labs to prepare for project
Feb 28	All: Reviewing project specifications and project document
March 7	Ahmed: Mode 0 and 1 code refactoring in C++ Hushaam: Refactoring demodulation and convolution functions for RF path Syed: Debugging and testing demodulation and convolution functions for RF path
March 14	Ahmed, Mohamed: Resampler initial implementation and minor edits to top-level code Hushaam: Creating bandpass, allpass, PLL and NCO functions in Python for stereo processing and future RDS use Syed: Beginning refactoring of bandpass, allpass, PLL and NCO functions to C++
March 21	Mohamed, Ahmed: Debugged resampler and tested its functionality in real and non-real time Mohamed: Implemented and tested threading in non-real-time Hushaam, Syed: Testing python model for stereo carrier recovery, channel extraction and processing. Refactoring Python stereo model to C++. Testing synthetic and live outputs for stereo path.
March 28	Everyone: RDS modeling in Python Mohamed: RDS channel extraction and carrier recovery Ahmed: RDS Demodulation (rational resampler, RRC, CDR) Ahmed, Syed: RDS data processing (Manchester and diffr. Coding, Frame Sync and Error Detection) Hushaam: Checking PLL function for correct state saving, modeling frame sync function, debugging RDS upstream
April 4	Mohamed: Mono audio Implementation, Analysis and Measurement Ahmed: RDS Implementation, Analysis and Measurement Hushaam: Intro, Project Overview, Stereo Implementation, Improvements Syed: Labs and Front-End Implementation, Improvements, Conclusion

Conclusion

Through this project, skills to design, implement, and analyze complex systems for SDRs were developed by working from high-level modeling of basic primitives to prototypes in form factor-constrained environments using both python and C++. Knowledge from current and prior courses was also consolidated throughout the development of the project creating better understanding of their concepts through real-life application. Lastly, working on a larger scale engineering project in a team-based environment provided experience breaking down projects and working on them in parallel to others to progress more efficiently. Overall this project provided the skills and experiences necessary for handling future engineering endeavors.

References

[1] N. Nicolici - COMPENG 3DY4 Project Description 2022 (February 2022)
[2] N. Nicolici - COMPENG 3DY4 Lecture and Lab notes (February 2022)
[3] https://www.cplusplus.com/reference/