3DY4 Project Report

Submitted By:

Sohel Saini (sainis43)
Chris Jiang (jiangc46)
Imad Ali (alii35)
Moh Alkouni (alkounm)

Group Number:

Group 45

Due:

April 4, 2025

Introduction

The primary objective of this project was to decode and implement a complex, industry-level specification for a real-time software-defined radio (SDR) system while addressing the practical challenges of embedded system design in a constrained environment. The system was designed to support real-time reception and decoding of analog FM broadcasts in mono and stereo modes and digital information transmitted via the Radio Data System (RDS) protocol. Achieving this required low-level signal processing pipelines that were robust, efficient, and capable of operating on limited hardware resources. The solution was deployed on a Raspberry Pi using a USB RF dongle as the RF front-end, with careful attention to sample rate conversion, demodulation, resampling, and filtering in both time and frequency domains. This project emphasized signal integrity, synchronization, system modularity, and real-time performance.

Project Overview

This project aims to implement a software-defined radio (SDR) system that demodulates and captures real-time analog FM audio and digital RDS data. In Canada, FM radio channels occupy the frequency range from 88.1 MHz up to 107.9 MHz, each being a 200 kHz wide channel centered around a specific carrier frequency. Once the FM is demodulated, we will be left with three main sub-channels: the mono audio signal (0–15 kHz), the stereo difference signal (23–53 kHz), and the RDS digital data signal (approximately 57 kHz, typically from 54–60 kHz).

Frequency modulation is obtained by varying the carrier wave frequency proportionally with the baseband message signal. In the receiver, the original message is recovered from the received signal by observing the change of the received signal's frequency with time. Compared with the specialized analog circuitry of traditional radios, for filtering, mixing, and decoding, software is utilized by an SDR for these functions on a general-purpose, embedded, or other type of computing platform like a Raspberry Pi.

One of the most critical features of the SDR design is resampling, the conversion of the rate of the signals from the high-frequency RF input down to the IF, and then down further to the output rate of the audio. Resampling is performed with three processes: adding zeros (upsampling), filtering out the undesired aliases or the spectral images, and then downsampling to lower the rate.

Filtering is performed with the application of the finite impulse response (FIR) filters, which can be obtained by convolving with the input. The filter coefficients contain a sinc function windowed to limit its extent, and this results in a linear phase and predictable frequency response. The system uses a phase-locked loop (PLL) to demodulate the stereo sound, creating the 38 kHz carrier frequency from the 19 kHz pilot tone on the FM broadcast. The regenerated frequency is then used for translating the (L−R) stereo difference channel down to the baseband, allowing us to reconstruct the right and left channels of the sound when combined with the mono (L+R) channel.

RDS supports the utilization of FM radio broadcasts to carry low-rate digital data along with their sound. RDS data is carried on a subcarrier of 57 kHz, the third harmonic of the 19 kHz pilot tone. The subcarrier is isolated and scaled down to baseband, and the system filters and resamples the data down to the symbol rate demodulates, and samples the digital data being carried.

Implementation details

Lab

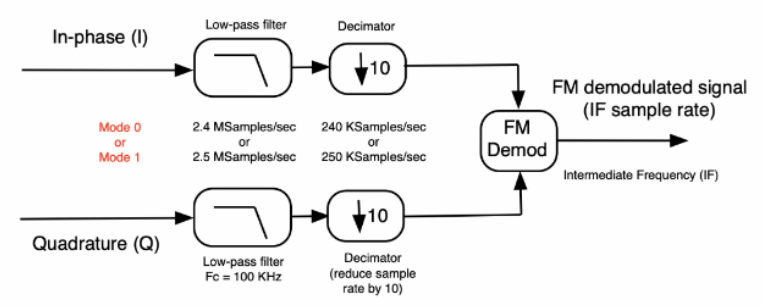
The labs laid the foundation for the software-defined radio (SDR) project by guiding us through the fundamental concepts of digital signal processing (DSP) and how they are implemented. In Lab 1, we used Python to explore Fourier transforms and digital filters. It was a great way to understand how convolution works and how impulse responses are built using sinc functions. Then in Lab 2, we re-implemented our Python logic in C++ using data types like float and double. This helped us see how signal processing can be done with more control and precision, especially when performance matters. Lab 3 was all about the FM signal processing pipeline. We worked with interleaved I/Q data sampled at 2.4 Msps, split it into separate I and Q channels, and filtered it to isolate the FM signal. From there, we used the arctangent method to demodulate the FM signal. After that, we filtered and downsampled it again to extract the mono audio. Finally, in Lab 4, we focused on benchmarking and testing different DFT and convolution algorithms. We made sure everything was accurate using test cases and even optimized the code by unrolling loops and precomputing index ranges to make things run faster.

RF Frontend

The RF frontend process starts by reading the block data, which are 8-bit unsigned samples that range from 0 to 255. The readBlockData() function reads the raw bytes from stdin and normalizes these values to -1 to 1, which we store into a variable called iq_data. The purpose of this was to convert the data to float for digital signal processing operations. The next thing that needed to be done was to split up the I and Q samples using a function we made called split_IQ().

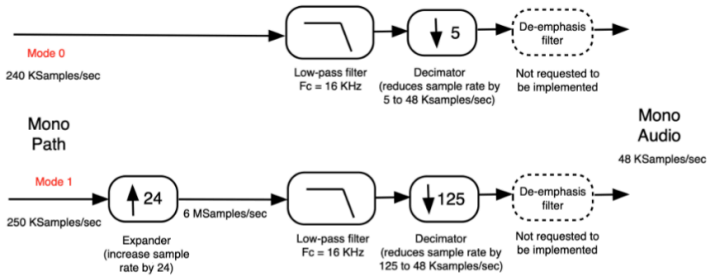
```
void split_IQ(const std::vector<real> &iq_data, std::vector<real> &i_data, std::vector<real> &q_data){
    for (size_t i = 0; i < iq_data.size(); i+=2) {
            i_data.push_back(iq_data[i]);
            q_data.push_back(iq_data[i+1]);
    }
}
```

We passed the iq_data to this function and pushed the I and Q data into their respective I and Q data vectors to perform operations on. Once the data has been converted and the IQ samples have been split, the next order of operations is to follow the diagram given to us in the project document.



From this diagram, the I and Q samples need a low-pass filter to be applied, with a decimator value to reduce the sample rate, and finally for the signal to be demodulated. In our implementation, the my_own_conv_state function applies a lowpass filter using the rf_coeff, and it combines the filtering with decimation/upsampling (depending on the mode). The rf_coeff contains the impulse response of our lowpass filter designed to isolate the FM channel, which had a cutoff frequency of 100 kHz. Our function handles decimation by only calculating the samples that would survive after decimation. This processes the samples efficiently as it only calculates what is needed. Finally, after the IQ samples have been filtered, we demodulate the signal using a function called my_own_fmDemodArcTan(). This function calculates the phase difference as well as handles the states. For example, the first samples in a block will use the previous I and Q states; the rest of the blocks follow the same pattern. This function produces an FM signal that will be further processed in mono, stereo, and RDS.

Mono Path



The Mono path consists of a low-pass filter to ensure we only keep the samples less than 16 kHz and then a decimator to lower the sample rate to the intended audio sample rate. In modes 2 and 3, the path also has an expander to increase the sample rate before the lowpass filter. The low-pass filter consists of 2 steps: first attaining the low-pass coefficients and then the convolution using the array of coefficients.

The *impulseResponseLPF_amp()* function generates a lowpass filter impulse response (filter coefficients) using a sinc function scaled to the desired cutoff frequency (16 kHz for mono). Since the mono path involves both upsampling (modes 2 and 3) and downsampling, the sampling frequency needs to be increased by the same factor the data is upsampled to ensure no aliasing in the outputted coefficients;

also, the number of taps (coefficients) needs to be increased. This is from Harris's rule of thumb; as the upsampling factor increases, the filter size must also. The *impulseResponseLPF_amp()* function also amplifies the coefficients by the upsampling factor since when upsampling the data, the energy is maintained in the frequency domain but distributed along a larger domain, causing each frequency spike to have less energy. This creates an issue when filtering since we discard a large portion of the domain and are left with a weaker signal. To mitigate this, we multiply each of the coefficients by the upsampling factor; therefore, when the signal is filtered, we are left with a much stronger output filtered signal.

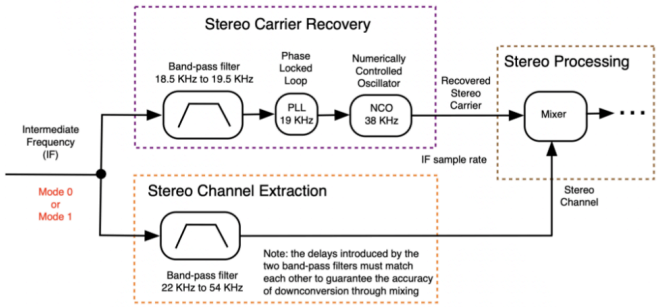
The convolution first involves upsampling the data; this is when we "pad" the i_data/q_data vector with 0's so it outputs a larger vector after convolution. In mode 2 the Intermediate Frequency (IF) is 240 kHz while the audio frequency is 44.1 kHz, since the audio frequency is not a factor of the IF, we would need to first upsample the data and then downsample. To find the factors, we first find the greatest common denominator (GCD) between the two (300) and then divide the audio frequency by the GCD to attain the upsampling factor; similarly, we divide the IF by the GCD to attain the downsampling factor. Then, after upsampling the data, we can use convolution to apply the lowpass filter coefficients to the data and attain the filtered data (F.1). Then, after convolution, we must downsample the data so it is now at the appropriate frequency for audio data.

$$y[n] = \sum_{k=0}^{k=Ntaps} h[k] * x[n-k]$$

F.1. The Convolution Function

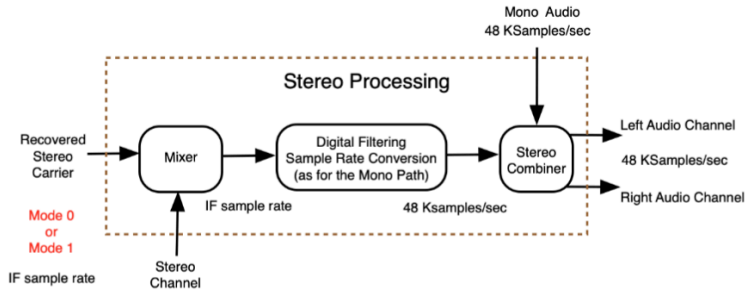
A resampler is implemented to do both the upsampling and downsampling within the convolution, thereby optimizing the mono path significantly. Upsampling has a time complexity of O(N*US) since it must iterate through the entire upsampled dataset, and downsampling has a time complexity of O(N*US/DS), and including convolution, the total time complexity is O(M(N+M)+N*US + N*US/DS)). By implementing the resampler, we significantly reduced the time complexity to O(N*US/DS).

Stereo



The stereo path begins with filtering and processing the demodulated FM signal to extract the stereo pilot. This is achieved by first using the *impulseResponseBPF(),* which outputs the coefficients for the BPF based on the low and high-frequency cutoffs. For stereo, since the DSB-SC signal requires the

pilot tone to accurately recover the stereo channels, a band-pass filter is applied to isolate the 19 kHz pilot tone from the rest of the signal, and the same is applied for the stereo channel at 22-54 kHz. The PLL generates a 38 kHz carrier by doubling the 19 kHz pilot frequency.



The recovered stereo carrier is combined with the stereo channel as cos(A)*cos(B)*2, since with trig identities this can be simplified to (cos(A+B) + cos(A-B)), where we can apply a low-pass filter to isolate for the lower frequencies (cos(A-B)). Finally, for the stereo combiner, we need the mono audio to combine with the stereo audio, as (L+R) represents the mono audio and (L-R) represents the stereo audio. By adding and subtracting these two, we attain the left and right channel audios.

To test the stereo path, we created test cases for the PLL since this is a key component of the stereo implementation. To ensure it was working correctly, we used a sinusoid at varying amplitudes, passed it through the PLL, and measured the frequencies at the output using a spectrum graph to verify that the output was correct. By changing the ncoscale we can increase or decrease the frequency of ncoOut, and by measuring it using a spectrum graph, we can watch for whether the inputted frequency has been increased or decreased as expected based on our input.

Multithreading

In this project, multithreading is used to enhance the efficiency of processing audio signals by running two separate threads concurrently: one for producing data and another for consuming it. The produce_RF thread is responsible for reading the incoming RF data, processing it through the RF front-end filtering, and then storing the processed data in a shared queue. Meanwhile, the consumer thread retrieves the processed data from the queue and performs further processing, such as mono or stereo audio handling. By running these tasks in parallel, the program can maximize its performance, allowing for continuous data processing without unnecessary delays.

The Queue class plays a critical role in facilitating communication between the producer and consumer threads. It uses a combination of a mutex and a condition variable to ensure thread-safe access to the queue. The mutex ensures that only one thread can access or modify the queue at any given time, preventing potential data races and ensuring the integrity of the data being processed. The condition variable is used for synchronization between the threads: when the queue is empty, the consumer thread waits for the producer to add data. Once data is available, the producer signals the consumer via notify_one(), allowing the consumer to proceed with processing.

To test our implementation, we used print statements in our Queue class to see whether the producer or consumer is the bottleneck to the project. We found the consumer to be slightly slower, which is to be expected as it has a large number of computations before it can output audio and receive new data.

Analysis and Measurements

Mode	Initial Block Size	IF-Block Size	Output Block Size	multiplications and accumulations per audio sample
0	192000	19200	3840	606
1	115200	28800	2880	1111
2	192000	19200	3528	~651
3	184320	15360	3528	~541

Mode	multiplications and accumulations per audio sample	Non-linear operations per audio sample
0	1784	20
1	3364	40
2	~1924	~22
3	~1580	~17

		Running Time (ms)			
Functions		Mode 0	Mode 1	Mode 2	Mode 3
IQ Samples		0.698	0.3185	0.602	16.38
Fm Demodulate		0.016	0.025	0.016	0.013
PLL		0.542	0.873	0.567	0.458
Convolution	Fm Demod	2.94	4.446	2.872	2.019

	Mono (16 kHz)	0.282	0.213	0.294	0.244
	Stereo (18.5 - 19.5 kHz)	1.461	1.073	1.667	2.945
	Stereo (22 - 54 kHz)	1.317	1.461	1.406	1.624
	Stereo (digital filtering)	0.126	0.180	0.216	0.190

Audio Path	13 Taps	101 Taps	301 Taps
Mono	0.0502ms	0.273 ms	1.018ms
Stereo	1.81ms	4.54 ms	13.56 ms

*All timing measurements represent runtime for one block

*All the values in the table are averaged for each block in an entire 5-second audio raw sample.

Analysis

For mono-processing, the theoretical values were strongly related to the IF block size since, for each block, the front-end would have to do a convolution, which has 101 taps, and therefore each data point is multiplied 101 times. While there were calculations in the mono path, those were done after the data had been downsampled again so it had a significantly lower block size and therefore fewer calculations. For mono audio, there were no non-linear operations. The stereo theoretical values follow the same pattern as mono (order of highest to lowest modes). The non-linear operations per audio sample are because of the PLL and are for the stereo carrier.

The taps investigation led to a clear increase in processing time as the taps increased; this is expected since as the number of taps increases the number of computations also increases. But increasing the number of taps isn't always a bad thing, since the quality of audio theoretically increases with the number of taps, and we saw this is true during our tests. At 13 taps, both stereo and mono were outputting highly muted audio with static still present, but at 101 taps, the audio was clear without any static. But at 301 taps, the audio was slow from the increase in computations it now has to do.

Proposal for Improvement

One useful and meaningful addition to our current FM demodulation system is the inclusion of a real-time audio spectrum display module within Python block processing. Currently, power spectral density (PSD) is graphed at the time of execution, providing a static picture of the entire signal. While

useful, this limits our ability to diagnose development issues, especially with real-time data. By using live PSD feedback during processing, we can see how the frequency content evolves over time, and it is much simpler to debug anomalies in real time. This would be especially useful in stereo production, where such things as the 19 kHz pilot tone, the 38 kHz stereo subcarrier, and the 22–54 kHz difference signal must be properly separated and reconstructed. A real-time spectrum display would immediately show whether these tones are bleeding into unwanted bands. For example, in early stereo debugging, a faulty PLL can fail to lock onto the pilot tone, leading to a weak or unsteady 38 kHz carrier; this would be visible in the PSD. Likewise, filter misconfigurations (e.g., incorrect cutoff ranges or insufficient tap counts) would manifest as missing frequency components or excessive spectral leakage. Implementation is possible and non-intrusive. Within the Python environment, this can be done by calling fmPlotPSD() or matplotlib.psd() on smaller chunks (e.g., every 3–5 blocks) and updating a live plot with plt.pause(). This avoids flooding the GUI but still gives real-time feedback. The performance impact is zero for offline processing, and the logic can be disabled through a flag when not needed.

This feature also offers the ground for upcoming real-time debugging within the C++ deployment phase. For instance, in the event that visual response is viewed as being valuable in the integration with audio drivers or hardware SDR input, this functionality can be introduced through GUI libraries including Qt or SDL with FFT libraries such as FFTW to visualize. This remains portable and offers useful long-term functionality.

In general, incorporating real-time spectrum observation adds transparency, accelerates debugging, strengthens knowledge of FM signal constituents, and provides a more interactive development process without necessarily having to fundamentally alter the current pipeline.

By adding a dedicated thread for live input of parameters such as the number of taps, block count, and processing mode, the program can be dynamically adjusted without needing to stop, rebuild, or restart. This allows developers to make real-time changes during testing, improving the speed and efficiency of debugging. Developers can quickly observe the effects of parameter adjustments and immediately identify issues, without the time-consuming overhead of halting and rebuilding the program. This dynamic approach not only accelerates the debugging process but also allows for continuous performance tuning and immediate feedback. It enables developers to isolate specific issues more easily and test different configurations in real-time.

Incorporating a graphical user interface (GUI) into the program would further improve the live debugging process by providing an intuitive and visual means for interacting with the system. A GUI can allow developers to easily modify parameters such as the number of taps, block count, and processing mode using sliders, input boxes, or dropdown menus, reducing the need for manual code changes or command-line inputs. This visual interface makes it easier to monitor the real-time performance of the system, visualize data flow, and receive immediate feedback on parameter changes, enhancing the overall user experience. With the ability to interact with the program in a more user-friendly way, developers can

test, adjust, and debug more efficiently, ultimately speeding up the development cycle and improving the quality of the system. The combination of a live input thread and a GUI provides a robust environment for continuous, interactive testing and optimization.

Project Activity

Week	Chris	Imad	Sohel	Moh
Feb 9-16	Reviewed Lab concepts and prepared for Cross examination	Reviewed Lab concepts and prepared for Cross examination	Reviewed Lab concepts and prepared for Cross examination	Reviewed Lab concepts and prepared for Cross examination
Feb 17-23 (reading week)	Met Group members and reviewed project document	Met Group members and reviewed project document	Met Group members and reviewed project document	Met Group members and reviewed project document
Feb 24 - Mar 1	Reviewed project documentation to implement previous functions and get started with initial phase of the project	Reviewed project documentation to implement previous functions and get started with initial phase of the project	Reviewed project documentation to implement previous functions and get started with initial phase of the project	Reviewed project documentation to implement previous functions and get started with initial phase of the project
Mar 2-8	Continued doing preliminary research and reading.	Reviewed Lab 3 to understand the model code.	Read over RF front end to understand how it worked and get an Idea of how to start.	Used the mono path from lab 3 to produce 48kHz mono audio. Verified correctness with PSD plots and .wav output.
Mar 9-15	Started working on the RF frontend by taking user inputs to be able to set the parameters depending on the mode.	Started mono and created the resampler using the lecture notes.	Implemented FM demodulation and helped on debugging RF front end and started to read theory on Mono and started implementation.	Implemented PLL to lock onto the 19 kHz pilot tone and used its NCO output to generate the 38 kHz stereo carrier needed for channel separation.
Mar 16-22	Finished the RF frontend and tested mono to see if it was being implemented correctly following the diagram given on the project document.	Debugged and tested mono on the Raspberry Pi, started debugging Stereo, and created test cases for the PLL, and the resampler	Implemented Mono up to the PCM conversion and started testing. Read on Stereo theory to help with initial implementation.	Applied a 22–54kHz bandpass filter, mixed it with the stereo carrier, and extracted the left and right channels. Applied a 16kHz final low-pass filter to clean the output.
Mar 23-27	Tried to debug stereo	Implemented	Ran tests on the	Fine-tuned and finalized

	and mono issues that we were facing (little bits of static). Collected live data and tested if our modes received audio other than static.	Multithreading including test print statements to ensure correct implementation and tested the program on the Raspberry Pi. Started looking into RDS based on the lecture notes.	Raspberry Pi to conduct live testing of Mono and Stereo modes.	stereo processing pipeline. Integrated mono and stereo code, verified output using .wav exports and PSD plots, and validated against project specs.
March 28-April 4	Final Cross-Examination and Worked on Final Report	Final Cross-Examination and Worked on Final Report	Final Cross-Examination and Worked on Final Report	Final Cross-Examination and Worked on Final Report

Implementing this project was not trivial due to the real-time nature, precision requirements of digital signal processing, and the requirement of modular block-oriented design. The most challenging part was realizing functionally and time-consistent stereo decoding, which required a deep insight into the FM stereo multiplex signal, phase-locked loop (PLL) dynamics, and precise filtering. As we were working with block-based processing, we had to perform state-saving carefully between all the pipeline stages to avoid discontinuities in the audio. Particularly tricky was getting the PLL to lock stably to the 19 kHz pilot tone, and instability in this stage manifested as distorted or noisy stereo output. Moreover, balancing computational efficiency and accuracy in resampling, filtering, and convolution added another layer of difficulty, especially in debugging on the Raspberry Pi, where system resources were constrained. Since a lot of the development took place closer to the deadline, we also had to debug along multiple signal paths in a rush, integrate between mono and stereo modes collaboratively, and preserve audio fidelity through PSD and .wav outputs. All of these had to be adapted to the strict specifications prescribed in the project documentation. Each of these challenges was surmounted by iterative debugging, theoretical comprehension, and diligent checking of outputs at every stage of the FM processing chain.

Conclusion

The entire project was an enriching experience in learning how to design and implement a genuine real-time FM receiver system. With the inclusion of both mono and stereo processing chains, we also had a good understanding of signal flow, frequency-domain filtering, and block-based processing. Despite some initial setbacks, we successfully modeled and tested the entire stereo signal path, demonstrating our ability to extract and provide high-fidelity audio from basic RF data. Overcoming the issues we faced with PLL synchronization, convolution accuracy, and stereo decoding was difficult, but it made us better as digital communications system learners and ready for the second half of the project. This work provided the foundation for further development to complete RDS functionality and simplified C++ integration for real-time applications.

References

[1] 3DY4 Project Document, Documentation, “COE3DY4 Project
Real-time FM Receiver: SDR for Mono/Stereo Audio and RDS” Bachelor of Engineering, McMaster University [Online through Avenue to Learn and Github]

[2] 3DY4 Project Document, Documentation, “3DY4 Project – Custom Settings for Group 45” Bachelor of Engineering, McMaster University [Online through Github]

[3] 3DY4 Lecture, Documentation, “COMP ENG 3DY4: Computer Systems Integration Project. Lecture Notes / Labs/Project” Bachelor of Engineering, McMaster University [Online through Avenue]