COE3DY4 Project Report

Group 3
Jack Wawrychuk, Minhaj Shah, Sophie Ciardullo, Zachary Thorne
wawrychj@mcmaster.ca, shahm23@mcmaster.ca, ciards1@mcmaster.ca, thornez@mcmaster.ca
April 9, 2021

1. Introduction

The goal of this project is to explore the implementation of a software-defined radio (SDR) system for real-time reception of both frequency modulated (FM) mono/stereo audio, and digital data. The project requires the consolidation of a lot of foundational concepts including modulation and signal processing to understand and apply the processing of the FM broadcasted data. Rather than duplicating pre-existing initiatives, our goal was to explore front-end radio-frequency (RF) hardware and single-board computers to implement a real-time complex system.

2. Project Overview

Software-Defined Radio (SDR)

A software-defined radio (SDR) receives real-time FM mono/stereo audio as well as any digital data sent through FM broadcasting using the RDS protocol. The system as a whole consists of multiple fundamental components discussed below.

Frequency Modulation (FM) and the Mono, Stereo, and RDS Sub-Channels

Frequency modulation refers to the process of encoding information in a carrier wave to be received and demodulated for broadcasting. The Bandwidth of 1 FM channel is 100kHz. From 0 to 15kHz is the mono sub-channel, from 23 to 53kHz is the stereo sub-channel, and from 54 to 60kHz is the RDS sub-channel. It is also important to note the stereo pilot tone located at 19kHz, which is used to synchronize the stereo sub-channel to the second harmonic.

The three different FM sub-channel signals are processed differently; the mono audio output data is the sum of both the left and right audio channels. The stereo sub-channel contains the difference between the left and right audio channels so that when combined with the pilot tone and the mono audio data, the left and right channels can successfully be produced. RDS is digital data that needs to be decoded using digital signal processing techniques.

Finite-Impulse Response (FIR) Filtering and Demodulation

When FM audio data is received, it is first filtered using a finite-impulse response (FIR) filter. The filter coefficients, which we are responsible for determining in our custom filter implementation, are dependent on the number of FIR taps, the cutoff/bandpass frequencies, and the sample rate. To avoid latency in data acquisition and to satisfy reasonable memory requirements, the FM audio data is sectioned into blocks which are then filtered individually rather than the entire sequence being filtered in a single-pass.

After the FM signal is received, filtered, and downsampled, demodulation is performed on the remaining data points. Typically this is done using arctan, but for implementation we used

a fast version as described in the Embedded article which uses the derivatives of the I and Q data in order to get the demodulated result [1].

Phase-Locked Loops (PLLs)

A phase-locked loop (PLL) is a phase tracking device that essentially produces a clean output from a noisy input. A trade-off is that the faster a PLL can lock, its ability to lock to a weak input diminishes and more phase jitter will exist in the output. The PLL synchronizes the Stereo Carrier Recovery signal to the pilot tone to ensure proper phase-lock between the signals. The output from the PLL is multiplied by a scale factor using a numerically controlled oscillator.

Downsampling & Resampling

For each sub-channel, the signal is downsampled in order to achieve a desired frequency. However, the specific resampling factor differs depending on the signal processing mode. In Mode 0, the data is simply downscaled by a factor of 5 to obtain a final sampling rate of 48kSamples/sec. Differently, in Mode 1, the signal is first upsampled, then downsampled after being filtered to implement a fractional resampling. Due to the resampling, an anti-imaging filter needs to be used and Parseval's theorem needs to be applied to restore lost power.

3. Implementation Details

The process of implementing the FM radio on the Pi had four main stages: the initial labs, implementing mono, implementing stereo, and finally RDS.

3.1 The Initial Labs

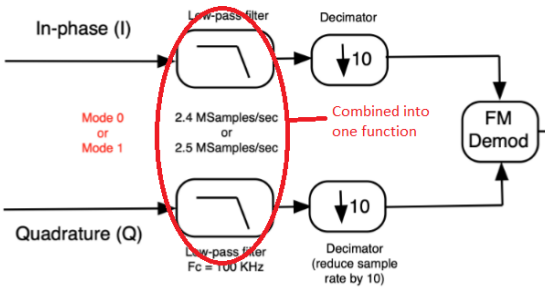
Our implementation of this project started with labs 1 and 2. The main purpose of these labs was to create the basic building blocks for the project. Lab 1 had three main tasks: create the impulse response function for a Low pass filter (LPF), a function to replace the built-in scipy filter function, and to become familiar with introductory block processing concepts. The main focus was to implement these functions in python so that we could have a model to refactor into C++. To implement the impulse response of the LPF in python, we simply followed pseudo code provided in the lab. To verify the correctness of our impulse response, we took the Fourier transform using scipy functions and plotted its magnitude to see if it resembled an LPF.

The basic filter function was easy to implement using a nested for-loop, since filtering is just convolution on the impulse response and the input function. The main challenge was understanding how block processing worked; once we acquired this understanding, we updated our filter function to return an array of size of the number of taps minus 1, which contained the last elements of the current block to be used in the next block.

In lab 2, we refactored our functions into C++. The main challenge in this was adjusting to using vectors, since type errors consistently arose unlike in the python implementation.

3.2 Implementing Mono

The process of implementing mono started with Lab 3. The code given for lab 3 provided a simple model for Mode 0 in python, including the RF



path where the I and Q samples were filtered, downsampled, and demodulated, as well as the basic path for Mono Mode 0. The figures show these two paths with the cutoff frequencies and decimation values. Initially, the scipy lfilter and firwin functions were used to implement the model without block processing. We found that using 151 filter taps was reasonably fast and yielded good results. We implemented block processing and chose a block size of 102400 because it was a large enough size to process a lot of data while maintaining reasonable runtime.

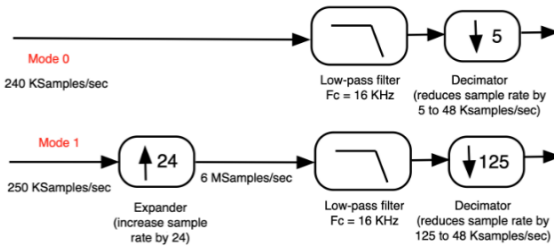
Next, we used the embedded article to implement a fast version of demodulation that utilized derivatives for the fast demodulation function [1]. We tested this in python before implementing it in C++ and noticed a substantial difference in the runtime.

The next step was to refactor each component into C++. We first refactored the mono path without block processing to ensure each component worked together correctly. Verification was done by comparing both the PSDs and quality of output to those of our python model. Then, we refactored the block processing model without optimization. We encountered one large error in this process; we accidentally resized the audio vector to the size of the I and Q vector and were initializing all of these vectors in the loop, which drastically increased our runtime.

Next, we worked on optimization. First, the downsampler was combined with the filtering such that we would only filter the samples that we would need post-downsampling to avoid spending time on discarded samples. We also created a function to take in both I and Q as an argument so their convolution and downsampling could be done simultaneously.

Next, we worked on reading in samples live. We read a block from standard in, cast each element as type char, normalized the values between -1 and +1, then cast it to type float. We chose to cast to float as it would be most space and time efficient, and normalized to make the values easier to work with. Once the values were read in, the data path was followed and then the data was cast to type short int and sent to aplay.

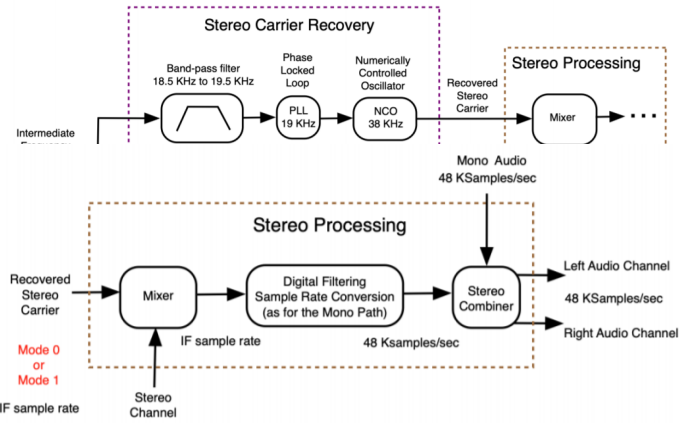
Next, we worked on Mono Mode 1. Because the signal is upsampled then downsampled as shown in the figure, we implemented a convolution function to combine the resampler with the filtering that contained the anti-aliasing and anti-imaging filter. Since our input is being upsampled, the number of taps in the LPF also needed to be upscaled. To do this, we created an algorithm to take each input of the array as though it had been upscaled, for example to consider the first index to be equivalent to the 24th element of the upscaled value. Then, we could multiply input values by their corresponding impulse response value. Once a non-zero value was found, we iterated through the impulse response by 24, multiplying it by the corresponding input value. We added this directly into our C++ live implementation since we were confident in our understanding of Mono.



3.3 Implementing Stereo

The implementation of Stereo required the implementation of new components including a Band Pass Filter (BPF), Phase Locked Loop (PLL) and the Numerically Controlled Oscillator (NCO). The implementation was broken down into two part; the Stereo Channel Extraction and

Stereo Carrier Recovery, and then the Stereo Processing. The figure shows the Stereo data path.



For the Python Implementation, we first implemented a non-block processing version. The BPFs were set up as in the previous labs and the PLL/NCO pseudo code for non-block processing was given in the project outline. After selecting a proper ncoScale factor of 2, we confirmed the functionality of the filters and PLL by examining plots from the generated data. For the bandpass filters, clear extractions of the desired frequencies were seen in the resulting plots. Next, mixing was added simply as pointwise multiplication followed by an LPF. Then, decimation was done and the final combiner either added or subtracted the stereo values from the mono values to obtain the left and right channels.

Next, the model for block processing was created which required state saving. For this, added to the list of parameters being passed to the PLL function as well as the internal operations of the function. For the stereo carrier recovery of each block, an additional array was passed as a parameter to the PLL function. This array contained the integrator, phase, I and Q feedback, trig offset, and the last element from the previous block. The initial values which were originally assigned inside the PLL function were instead assigned in the stereo block file for the first block.

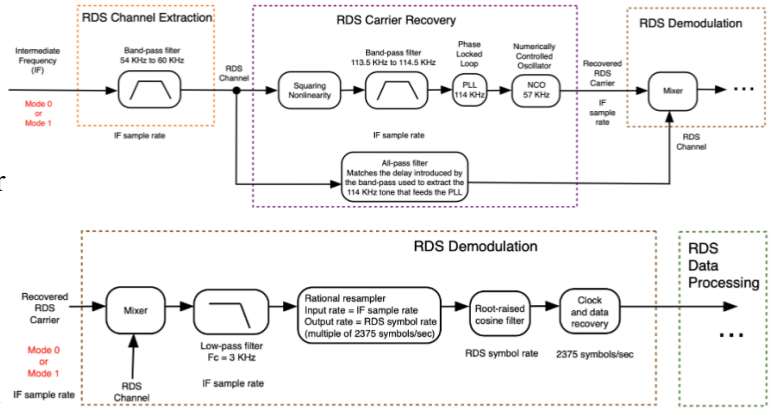
After implementing Mono, the process of implementing stereo in C++ was more efficient. Rather than implementing Stereo in basic and block form, we implemented it live right away. Once the BPFs, mixing, and additional functions were implemented, we decided to implement threading to address the choppy output audio.

We first implemented threading with a queue of vectors. We quickly realised that pushing vectors of that size onto the queue was eating up a lot of resources and decided to use pointers to make our threading faster. We created a chunk of static memory that would store the data, then we would pass the address of the first element in the static memory onto the queue which was of a void pointer type. The mutex was placed around this queue to ensure that the RF thread only pushed an address on the queue when it was not full, and the mutex ensured that the audio thread would pop nothing from the queue if it was empty. When doing this, we returned to our custom demodulation function to write the output directly to the static memory to reduce the runtime. In doing this, we updated the arguments of the function as well as how the output was indexed.

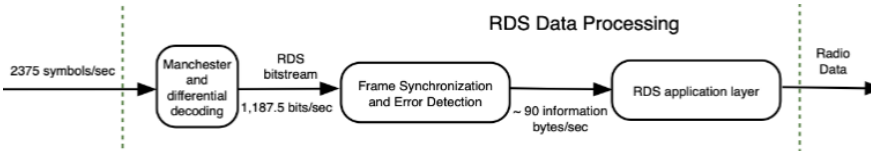
When implementing in C++, we noticed issues with the PLL; in the first revision of the refactored code, the resultant audio had audible discontinuities. The discontinuity was visible when the PSD was plotted for the transition between blocks from the output of the PLL. Upon analyzing the problem, a slight indexing error was noticed, specifically that the indexing was consistently off by one. This was fixed by adjusting the indexing of the returned values of the ncoOut array in the PLL function.

3.4 Implementing RDS

With guidance from the project document, we first implemented a model without block processing. Since the all-pass filter has no filtering delay, the RDS carrier is not in sync with the RDS channel data unless the appropriate phase adjustments are added to the PLL. To determine the proper adjustments, we first implemented RDS demodulation as outlined in the signal flow diagram. We mixed the data, passed it through the convolution function, then resampled to yield a multiple of 2375. We chose 24 samples per symbol, meaning the resampler needed to get us to a frequency of 57kHz. To do this, we upsampled by 19, passed it through an anti-imaging filter, then down-sampled by 80. We then passed this through an RRC filter to obtain data on which we could perform clock and data recovery.



For data processing, the symbols were converted to bits for differential decoding. Before this, a screening was completed to identify the starting point as either the zeroth or first symbol. To do this, we checked how many double highs and double lows occurred when starting at both element 0 and element 1; the element with the lower number would be the starting position. From here, the symbols were converted to bits in a for-loop. In each iteration, two elements were compared; if the first was greater than the second, it would be bit 1, otherwise bit 0.



For the differential decoding, each bit in the array was XOR'd. This data then entered the for-loop for frame synchronization where 26 bits from the vector were multiplied by the parity matrix, giving a result of one row with 10 elements. The elements were compared to the four potential syndromes; a print statement would indicate if one was detected. In the case that a syndrome appeared not 26 bits later, the could would mark this as a false positive.

Next, block processing of RDS was implemented. The block size was increased to 307200 because the previous block size was too small to obtain a desirable number of bits. Additional variables were introduced to save values for the next block. The first occurrence of block processing was during Manchester decoding; if the screening indicated the starting position to be 1, the last symbol was saved in the variable *lonely_bit* since the length was an even number. For every block other than block 0, this bit was compared to the first bit of the current

block, and the result was appended to the bit stream before differential decoding. The second occurrence was in frame synchronization since the while-loop would terminate when a block of 26 could no longer be taken. Thus, an array was made to take the last 27 used bits of the bitstream. Before the next block enters the loop, these 27 bits are appended to the front.

Since our C++ code reads in live data using block processing we decided to implement RDS directly into this by creating a separate thread dedicated to RDS. We implemented 2 queues to ensure that the RDS and audio threads could read the proper data. The RF thread pushed the same values onto both queues, but each had its own separate mutex and condition variables. This worked effectively, however other challenges arose as we needed to readjust the phase-adjust by an additional -pi/2 to ensure correctness in the C++ constellations.

Lastly, to improve efficiency, we created another thread dedicated to clock recovery and RDS data processing. We had the RDS queue which did all of the processes until the end of the RRC filter. The filter result was pushed onto another queue which would be popped in the frame queue for the rest of the processing. The runtime improved, but for further optimization we combined the PLL with the squaring nonlinearity and BPF. Ultimately, our implementation did not fully work as the audio exhibited audible discontinuities on the Raspberry Pi and the frames only synced up for at most four blocks.

4. Analysis and Measurements

For Mono Mode 0, the block size is 307200/20 = 15360 samples. The number of taps is 151 and the decimation amount is 5, resulting in 463872 multiplications.

For Mono Mode 1, the block size is again 15360. The number of taps is 3624, which is 151 upsampled by 24. The decimation amount is 125, resulting in 445318 multiplications. The runtime for mono mode 1 is ~8x longer than Mode 0, which can be attributed to the use of more complex math such as modulus division in the Mode 1 filtering.

For stereo, both carrier recovery and channel extraction used filters with block sizes of 15360 and 151, resulting in 2319360 multiplications for each. Due to the absence of decimation, this is ~5x more multiplications than Mono processing. The runtime for stereo channel extraction is ~5x larger than that of Mono processing, which is consistent with the increase in number of multiplications.

For Stereo Mode 0, the values shown in the table yield a total of 4673512 multiplications per block. It is ~10.5x the number for Mono Mode 0, while the runtime is ~13x higher. This slight discrepancy is likely due to increased overhead and more complex calculations.

For stereo mode 1, the values shown in the table yield a total of 4746114 multiplications per block which is similar to Mode 0. To obtain one audio sample, 5*151+1 multiplications and 5*1+3 accumulations need to be done as a result of the mixer, combiner, and demodulator.

Multiple filters were implemented to extract and process the RDS data with 11596800 multiplications per block. With a block size of 15360, there are 755 multiples per data point for processing. The frame synchronization thread entailed its own 20020 multiplications per block. For Stereo, the PLL loops 15360 blocks and the trig functions are called each time. For RDS, the

PLL is combined with the bandpass filter and squaring function so the trig functions are called 2319360 times.

Path	Function	Runtime/block(s)	Multiplies per block
RF Front end	Filtering + decimation for I&Q	9.294e-03	231936
	Demodulation	9.246e-05	15360
Mono (Mode 0)	Filtering + decimation combined	5.944e-04	463872
Mono (Mode 1)	Filtering + decimation combined	4.896e-03	445318
Stereo (mode 0)	Carrier recovery	2.975e-03	2319360
	PLL (similar for mode 0 and mode 1)	1.949e-03	15360
	Channel extraction	2.988 e-03	2319360
	Mixing	1.018e-05	15360
	LPF after mixing	4.10e-07	3072
	Combiner	3.51e-06	3072
Stereo (mode 1)	Mixing	1.124e-05	73728
	LPF after mixing	2.48e-06	15360
	Combiner	2.691e-06	2946
RDS Data processing thread	Data extraction	3.155e-03	2319360
	**PLL +squaring+BPF (PLLs contain sin, cos, atan functions)	1.170e-02	2319360
	LPF + mixer	4.168e-03	2319360
	Resampler + anti-imaging	5.886e-03	2319360
	RRC filter	7.72e-04	2319360
RDS frame-sync h thread	Various operations not limited to just multiplication, including “xor”, “and”	1.664e-04	20020

5. Proposal for Improvement

A feature that could improve the user’s experience is the addition of a de-emphasis filter at the end of the audio thread. This would improve the audio quality before it is played for the user because during transmission, pre-emphasis filters are used to attenuate certain frequencies which could be recovered using a de-emphasis filter. A second feature that could be added is an automatic phase adjuster. Since tuning the PLL can be quite time consuming, integrating an algorithm to look at the constellations and determine the phase adjustment would be beneficial.

To improve runtime efficiency, more threads could be added to complete multiple convolution calculations at one time for RDS, rather than multiple back-to-back as it exists now.

Another improvement could be to use a fast fourier transform, with the values multiplied by one another according to the time-and frequency-domain convolution/multiplication relationship.

6. Project Activity

Week	Progress	Contributors
February 22nd	Refactoring the basic version of the lab 3 code (no block processing)	Minhaj, Zach
March 1st	Mono Mode 0, unoptimized with block processing in C++	Sophie, Zach
March 8th	Optimizations and Implemented it in Live	Sophie, Zach
	Debugging Mono mode 0	Jack, Minhaj, Sophie, Zach
March 15th	Mono Mode 1, directly into the live implementation and debugging	Jack, Zach
	Stereo in python without block processing	Minhaj
	Stereo in Python with block processing/PLL state saving	Sophie, Minhaj
March 22nd	RDS python Model without block processing and with block processing	Jack, Zach
	Implementation of Stereo in C++	Minhaj
	Debugging Stereo in C++	Minhaj, Zach
	Implementation of pointer threading in C++	Minhaj, Zach
March 29th	Implementing and debugging RDS in C++ Live	Minhaj, Zach

7. Conclusions

This project provided all members of the group with significant theoretical knowledge about industry-level real time implementations of a computing system, which operates in a form factor-constrained environment. From this, members were able to conceptually build processes to achieve a variety of project specifications in a hands-on manner. All members of this group experienced an improvement in technical abilities, while also gaining significant experience in the Software Development Life Cycle (SDLC) throughout each phase of the project, and Source-Code Management Workflows. Ultimately, this project allowed all of the members to consolidate all of the signal processing knowledge learned through their degrees.

8. References

[1] E. Staff, “DSP tricks: Frequency demodulation algorithms,” 10-Jan-2011. [Online]. Available: https://www.embedded.com/dsp-tricks-frequency-demodulation-algorithms/. [Accessed: 09-Feb-2021].

[2] N. Nicolici, “COE3DY4 Project: Real-time SDR for mono/stereo FM and RDS,” 08-Mar-2021. [Accessed: 08-Mar-2021].