COE3DY4 - Real-time SDR for mono/stereo FM and RDS

Group 0
Amr Elhossan, Omar Abdulrahman, Hydar Zartash, Anthony Acosta
elhossaa@mcmaster.ca, abdulrao@mcmaster.ca,
zartashh@mcmaster.ca, acosta3@mcmaster.ca

April 5, 2024

1 Introduction

The 3DY4 project culminates three years of academic exploration into the practical creation of a Software Defined Radio. We aim to engineer a design that conforms to a complex, industry-standardized medium, and overcome the challenges of developing a solution while meeting form-factor and real-time constraints.

The scope of this project involves building a software-defined radio, using provided front-end RF hardware based on the Realtek RTL2832U chipset to capture FM radio data, and processing the signal using C++ running on a Raspberry PI 4 to extract mono/stereo audio and Radio Data System (RDS) information in real-time.

2 Project Overview

In this project, the provided radio frequency hardware is used to acquire an FM signal formatted as interleaved I/Q samples. Through Unix pipes, these samples are delivered to our software-defined radio, responsible for demodulating the data, and extracting the contained mono/stereo audio and the digital Radio Data System (RDS) data.

There are key building blocks that enable SDRs. The first is the FIR filter composed of a series of taps that when convolved with an input signal produces a signal with frequency components only in a certain range, the filter's passband. Techniques can also be used to change the sampling frequency at the output of a filter. The second is block convolution, as for a continuous real-time system, data must be processed in blocks, but processes like convolution must maintain smooth transitions. The third is the PLL, which is required to generate clock signals for downconverting DSB-SC modulated signals for extraction at the baseband. Another key is multithreading, as the signal flow from the RF front-end to the audio and RDS threads must be parallelized to attain real-time performance.

Before any data can be extracted, the 8-bit I/Q data must be demodulated. An RF front-end producer thread filters and decimates the incoming signal to isolate the usable data then demodulates it to produce usable information for the other consumer threads.

The audio thread can produce either mono or stereo depending on the mode of operation. In mono, to extract the audio, a low pass filter and resampler are used, as the signal is already at the baseband. Stereo extraction requires DSB-SC demodulation, which utilizes a PLL and a pilot to generate a carrier necessary to perform downconversion. In the stereo mode, mono is also extracted and both these signals are then combined to produce L and R channel audio.

RDS is a separate consumer of the RF thread, requiring a more complex process to extract data. The actual digital data is encoded using BPSK, so the signal must also be shaped to extract the information. CDR is performed to extract binary data from the waveform, and Manchester and differential decoding is performed to obtain a meaningful bitstream. This bitstream is synchronized to frames following the standard, and the contained data can be extracted from these frames and parsed.

3 Implementation Details

3.1 Lab Concepts

In the labs preceding the project, we built up concepts that would be crucial in realizing our SDR. In Lab 1, we developed an intuition for the frequency domain and how to interpret the behavior of signals and filters. We were introduced to DSP primitives in python from the *SciPy* library, and then we built our own models to achieve the same desired behaviour. Lab 2 required porting these building blocks to C++, the language we would be using for

the project because of its faster running times as a low-level compiled language. We were also introduced to tools we could use to analyze our source code, such as *GNUplot* for graphing and the *Chrono* library for timing analysis. Lab 3 required consolidating knowledge from the previous two labs to build the Python model for mono mode 0.

3.2 Filter Generation

The first key building block was to develop functions to generate finite impulse response (FIR) filters starting in Lab 1. The pseudocode, provided in the lab document, assigned tap coefficients along the sinc function to approximate a perfect rect in the frequency domain. In the modelling stage, we were able to inspect our LPF's performance by using the *SciPy signal* library's *freqz* function and plotting the attenuation across a range of frequencies. Creating a BPF was effectively the same, however, a final step involved upconverting by the center frequency. Moving from the model to the implementation was straightforward, as given we knew the model worked, we only needed to compare the generated filter for given sets of inputs and check to see consistency.

3.3 FM Demodulation

In Lab 3 we also modeled an efficient solution to demodulate the incoming RF signal. Traditionally, given I/Q components, an arctan function is used to extract the phase information and generate a demodulated signal. Our task involved using a description of the algorithm to implement it in code. It was sufficient to test our implementation by replacing the arctan version of the function and comparing the PSDs. We observed if the behaviour was similar and then translated it to C++.

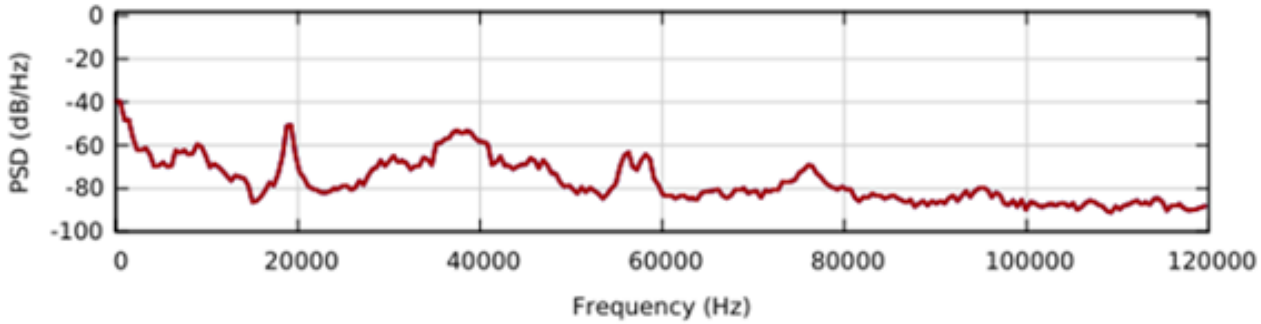


Figure 1: Output of FM demodulator

3.4 Block Processing

A cornerstone of a Real-time application is block-based signal processing, a concept first introduced in Lab 1. The incoming data is live streamed and arrives at a high sample rate. Attempting to hold the entire data stream would overwhelm storage while also defeating the purpose of a real-time system. To allow for block-based processing, any function that relies on previous blocks in the processing of the current block require a form of state-saving. In general, the process of "blockifying" any flow was to first develop the single pass model, and then attempt to match the result with the original model. We chose to verify rigorously in the model and trusted that the algorithms would hold constant in C++.

Converting our block-based convolution into C++ resulted in a subtle bug with state indexing. In Python, we made use of negative indexing to index the state array when $n - k < 0$. When translating to C++, we copied it without considering what negative indexing meant in C++. We thought that any Python quirks would be caught by the compiler. However, the faulty block-based convolution compiled with no errors. Consequently there was discontinuity between blocks and the sound quality reduced as a result.

Another interesting bug, a direct result of the compiler, was when we changed the iterators to unsigned data types. They were initially on signed int32 but we changed them because the compiler was warning us that comparisons with different signedness could result in undesirable behaviour. This broke the state transitions since those relied on $n - k < 0$. Due to the unsigned data types that difference is strictly positive. Fixing the bug was simple since we initially had the block convolution working but we changed it to appease the compiler.

```
1 void convolveFIR(std::vector<float> &y, const std::vector<float> &x, const
      std::vector<float> &h, std::vector<float> &initial_state, const int
      decimation_factor)
2 {
```

```
    y.clear(); y.resize(x.size()/decimation_factor);
    for (unsigned int n = 0; n < x.size(); n+=decimation_factor) {
        for (unsigned int k = 0; k < h.size(); k++){
            if (n-k < 0) {
                y[n/decimation_factor] +=
    h[k]*initial_state[n-k+initial_state.size()];
            }else{
                y[n/decimation_factor] += h[k]*x[n-k];
            }
        }
    }
    initial_state = std::vector<float>(x.end()-h.size()+1,x.end());
}
```

Listing 1: Faulty convolution algorithm

3.5 RF and Mono

Implementing the RF front end and mono audio processing involved piecing together FIR filters and decimations by the signal flow graphs provided. This was considered trivial enough to move directly to the C++. Especially since we created a working model in Lab 3.

An important observation, necessary to get mono working in real-time, was that it is unnecessary to compute the convolution for each input sample if the output is going to be decimated. Our validation efforts here were simple, as in addition to listening to the produced audio, we were able to inspect the frequency plots of both the demodulated data and the mono audio to see if the frequency data matched the shape expected based on the spec.

To validate our LPF was isolating the mono channel, we plotted the PSD after filtering. We checked if the frequencies above the mono channel cutoff appeared attenuated. Decimation was simple enough to not require rigorous validation. We confirmed decimation was correct by listening to the audio from aplay.

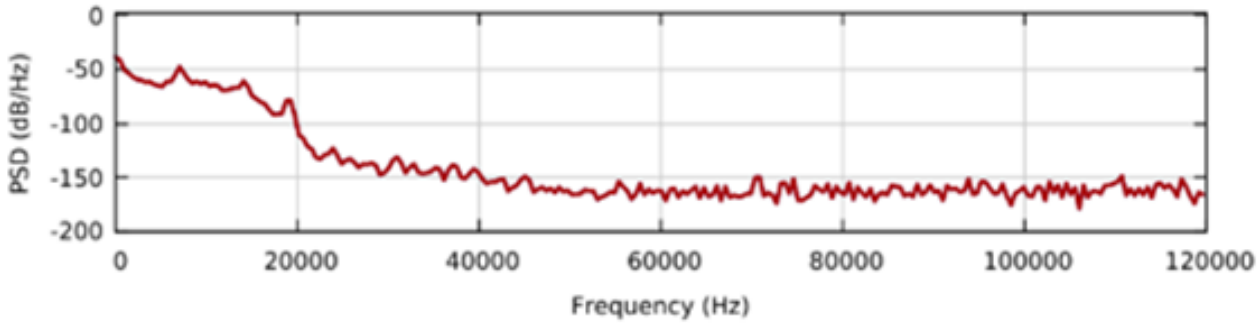


Figure 2: Output of LPF in mono path

3.6 Resampling

For modes 2 and 3, a fractional ratio between the intermediate frequency and output audio frequency makes the approach above insufficient. To apply the rational resampling by upsampling by an integer factor and then down-sampling would be far too computationally expensive and would not meet timing requirements. To circumvent this, we implemented polyphase filter-bank resampling, which uses phases of our filter taps to compute the upsampled data without requiring unnecessary computations.

To do so, we increase the number of taps in our filter by a factor of the upsample rate, so that when we select a number of phases equal to the upsample rate the number of partial products remain the same. We also apply a gain to compensate for the division of power across the images. Decimation still works as described before, as we only select, and therefore compute, the values that will end up in the output signal. The validation flow for the resampler was the same as the rest of mono, discussed previously.

3.7 Stereo

The novelty in the stereo path comes from the DSB-SC modulated stereo signal centred at 39 kHz. Unlike mono, processing stereo first requires the entire stereo band to be downconverted to 0 Hz. Additionally, to retrieve L/R audio data, the signal must be combined with the mono audio. The overall signal flow involves band pass filtering the 19 kHz pilot tone and the stereo signal, producing a phase locked carrier at twice the frequency of the pilot tone,

mixing the filtered stereo and carrier, and low pass filtering to remove the image produced when the stereo band is shifted down and up through mixing. The mono data is also delayed to match the delay these filters introduce, which is accomplished using an all pass filter with a phase delay.

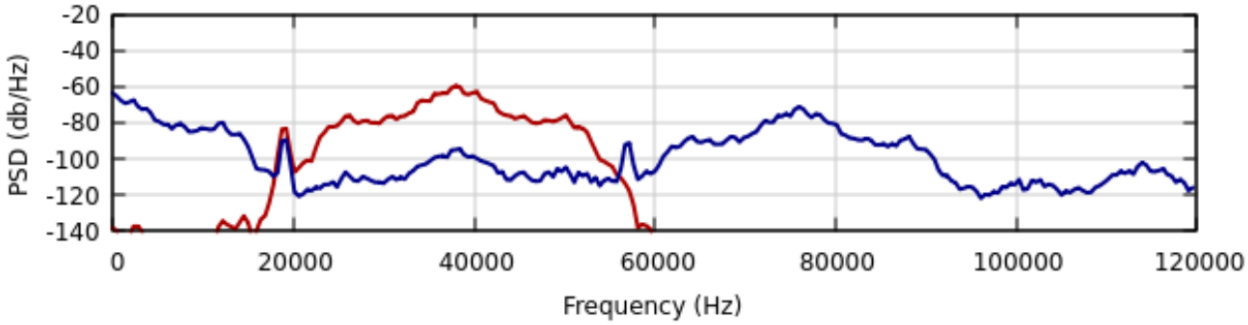


Figure 3: Extracted and post mixing stereo band

3.8 RDS

The RDS algorithm is more complex than the mono/stereo paths. Our flow for this section of the project relied on modeling the entire path in Python as a single pass algorithm first, then implementing block processing and state saving at every stage, before moving to the implementation in C++. The Python model was cross-referenced with the end-to-end RDS example in the pySDR spec, and we were able to benchmark our algorithms against the ideal result this way.

Due to project timeline constraints, some our algorithms (mainly CDR) were only implemented in a preliminary form, and were not as rigorous as needed to deal with a weaker signal or lower SPS.

3.8.1 Channel Extraction and Pulse Shaping

Like stereo, the RDS band is DSB-SC modulated. Unlike stereo, we do not use the pilot tone to downconvert the signal. Instead use a trick called squaring nonlinearity to generate our own pilot tone at the second harmonic of the RDS center frequency, which in implementation is an element-wise squaring of the original signal. We can then lock our PLL to this tone with an NCO scale factor of 0.5 and then mix the extracted RDS channel to downconvert the signal similar to the signal flow in stereo.

3.8.2 CDR

Our CDR algorithm implementation was simple and is a major area for improvement. For any block, we find the peak sample value by scanning within the range of our SPS value and taking the sample offset with the largest sum of magnitudes. We then use the sample offset that yields the highest average magnitude, and determine the symbol based on the sample value at the offset.

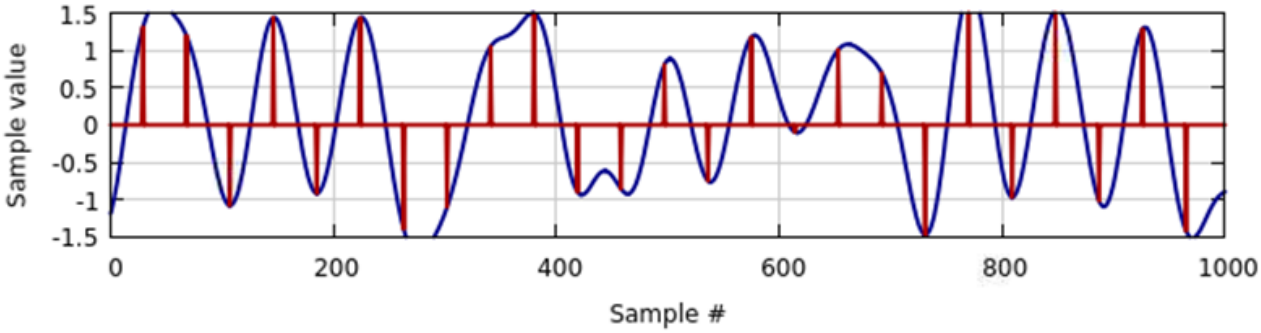
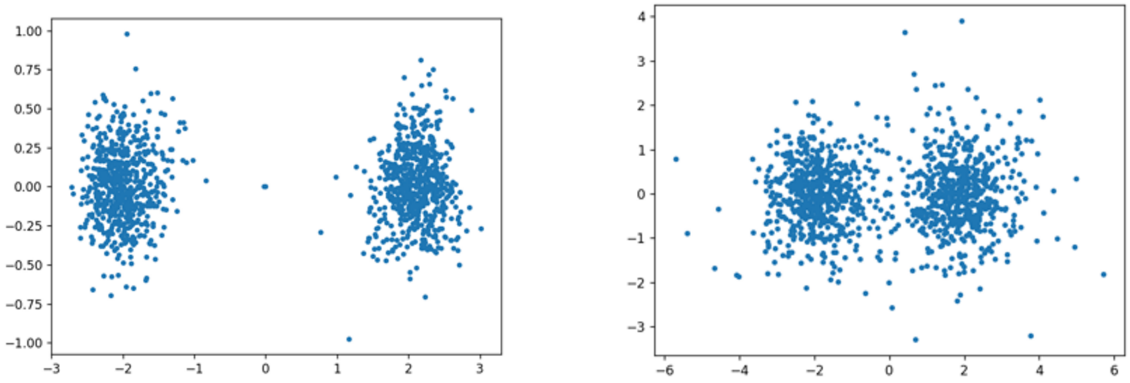


Figure 4: RDS symbol sampling

This approach is not ideal, as it is prone to error when the waveform is close to 0 at the sampled point. However, in development we found that for a strong signal with a high SPS, we were able to correctly extract the symbols

with a high enough accuracy to achieve frame synchronization and parse the data. We can increase our confidence in this assertion by viewing the constellation plot of our post CDR data.



(a) Strong RDS signal

(b) Weak RDS signal

Figure 5: Constellation plots of a strong and weak RDS signal

3.8.3 Manchester and Differential Decoding

Given the bitstream derived from the waveform at the output of CDR, the intended symbols must be determined based on the Manchester encoding. In Manchester encoding, a hi-lo represents a 1, while a lo-hi represents a 0. To pair our symbols correctly, we must determine whether to pair our symbols from the start of our block or if the first symbol should be skipped.

To determine the offset, we take our first valid block (i.e. the first block we use, once the PLL settles) and use a brute force approach of attempting to assemble bits from symbol pairs with both possible offsets, and "scoring" the number of violations (HH or LL pairs). We then select the offset yielding a better score.

State saving for the decoding happens conditionally, as we only need to use the last symbol from a block if the number of symbols in the block (after the offset at the start) is odd. We use a boolean flag representing this condition, passed between blocks to determine if the decoded block should start by including the symbol from the previous block, or if it should just start at symbol 0.

Differential decoding is simple, as each value is computed as the XOR of the previous value with the current. For state-saving, only the last symbol needs to be saved.

3.8.4 Frame Synchronization

The next step is to synchronize the incoming bitstream to extract meaningful information. We shift in the bitstream of the output from the differential decoder and scan the 26 LSBs until we match a block. Matching a block is done by calculating a syndrome resulting from matrix multiplication with a predefined 26x10 parity matrix. Once a block is found, the bits can be shifted in 26 at a time, and the 16 data bits can be saved, as discussed in the parsing section. To track valid groups, we parse only when a sequence of ABCD is received.

3.8.5 Data Parsing

Parsing the data involves interpreting sequences of bits with the provided specifications. For a 104-bit group, 40 bits were used for the checkwords and 64 represent the 4 data blocks. Since we don't need the checkwords at this point, we can contain all the relevant data blocks in a 64-bit register and mask out bits to extract the data we need. To extract the PI data, we need to take the 16-bit PI code from block A, which is the 16 MSBs of our register. PTY is similar, the five bits from block B (bits [42:38] of our register) can be used to index a list of PTY codes. To assemble the program service, we need to collect the block D data across 4 groups and assemble the 8-character program service. To accomplish this, we create a 64 bit register representing the 8 character string, and as groups are processed, we replace the appropriate 16 bits in the register (determined by the two LSB of block B) with the 16 bits from block D of the current group. Every 4 groups, we can check if the string is different from the last and then output to the terminal.

To verify this approach, we used a bitstream generated by our best model (SPS at 39, single pass) and fed it into our parser 64 bits at a time. We then checked to see if we could reliably get the PI, PTY and PS on the terminal.

4 Analysis and Measurements

- N: The number of filter coefficients.
- r: The frontend decimation coefficient.
- d: The audio backend decimation coefficient.
- u: The audio backend upsample coefficient.
- d_r: The RDS backend decimation coefficient.
- u_r: The RDS backend upsample coefficient.

	Mode Parameters					
	N	r	d	u	d_r	u_r
Mode 0	101	10	5	1	640	247
Mode 1	101	4	9	1	∘	∘
Mode 2	101	10	800	147	96	19
Mode 3	101	3	1280	147	∘	∘

Table 1: Mode parameter list

4.1 Analysis

4.1.1 Multiply and Accumulations

The mono path is composed of three filters. Two filters extract the I/Q 100 KHz FM band in the frontend while the third filter extracts the 15 KHz mono band located at the base of the FM channel. For all four modes of operation, the block size is $(1470 * r * d)/u$.

In the frontend filter since decimation occurs alongside the convolution $(1470 * d)/u * N$ multiplications are performed. Consequently, in the backend mono sub channel extraction, $1470 * N$ multiplications are performed. Performing the calculations for $N = 101$:

$$pp/output_{mode0} = (2 * (1470 * 5)/1 * 101 + 1470 * 101)/1470 = 1111$$
$$pp/output_{mode1} = (2 * (1470 * 9)/1 * 101 + 1470 * 101)/1470 = 1919$$
$$pp/output_{mode2} = (2 * (1470 * 800)/147 * 101 + 1470 * 101)/1470 = 1200.32$$
$$pp/output_{mode3} = (2 * (1470 * 1280)/147 * 101 + 1470 * 101)/1470 = 1859.91$$

Table 2 summarizes the number of necessary partial products required to compute one output sample in the mono path.

Mono			
	N		
	13	101	301
Mode 0	143	1111	3311
Mode 1	∘	1919	∘
Mode 2	∘	1200.32	∘
Mode 3	∘	1859.91	∘

Table 2: Mono partial products per output sample analysis

The stereo path is composed of seven filters. The first two frontend filters share the same usage as the mono path. Two band-pass filters extract the 19 kHz pilot and the stereo subchannel centred at 38 kHz. The fifth filter is a delay block, used to phase align the mono audio with the stereo, implemented as an all-pass filter with a shifted impulse. Finally, two filters are used to extract both the mono and downconverted stereo bands. Like mono, all four modes have a block size of $(1470 * r * d)/u$.

For the same reason as mono, $(1470*d)/u*N$ multiplications are performed in the frontend filters. Decimation of the sample rate to the output audio rate does not occur until the last two filters. As a result, the band-pass filters and the delay block all perform $(1470*d)/u*N$ multiplications. The mono and stereo baseband extraction require $1470 * N$ multiplications. Perform the calculations for $N = 101$:

$$pp/output_{mode0} = (5 * (1470 * 5)/1 * 101 + 2 * (1470) * 101)/1470 = 2727$$
$$pp/output_{mode1} = (5 * (1470 * 9)/1 * 101 + 2 * (1470) * 101)/1470 = 4747$$
$$pp/output_{mode2} = (5 * (1470 * 800)/147 * 101 + 2 * (1470) * 101)/1470 = 2950.3$$
$$pp/output_{mode3} = (5 * (1470 * 1280)/147 * 101 + 2 * (1470) * 101)/1470 = 4599.28$$

Table 3 summarizes the number of necessary partial products required to compute one output sample in the stereo path.

Stereo			
	N		
	13	101	301
Mode 0	351	2727	8127
Mode 1	∘	4747	∘
Mode 2	∘	2950.3	∘
Mode 3	∘	4599.28	∘

Table 3: Stereo partial products per output sample analysis

The RDS path contains seven filters. Again like stereo and mono, the first two frontend filters serve the same purpose. The next filter extracts the RDS band centred at 57 kHz. After squaring the RDS signal, a generated pilot tone is extracted using a narrow band-pass filter centred at 114 kHz. The RDS signal is passed through an all-pass filter to phase align it with the generated carrier. Then two more filters are used the first one extracts the 3 kHz baseband RDS signal and then this filtered signal is passed through a root-raised cosine filter to reduce inter-symbol interference.

$(1470*d)/u*N$ multiplications are performed in each filter until the baseband RDS extraction. $(1470*d)/u*u_r/d_r*N$ multiplications are performed for the RRC filter and the baseband extraction filter.

$$pp/output_{mode0} = (5 * (1470 * 5)/1 * 101 + 2 * (7350 * 247/640) * 101)/36 \approx 119017.3$$
$$pp/output_{mode2} = (5 * (1470 * 800)/147 * 101 + 2 * (8000 * 19/96) * 101)/39 \approx 111788.9$$

Table 4 summarizes the number of necessary partial products required to compute one output bit in the RDS path.

RDS			
	N		
	13	101	301
Mode 0	15319.1	119017.3	354695.1
Mode 2	∘	111788.9	∘

Table 4: RDS partial products per output bit analysis

4.1.2 Non-Linear Operations

The PLL is the only location in the source code where non-linear operations occur. In the main loop of the PLL four of these operations are performed per iteration. The number of iterations depend on the IF block size i.e,

$(1470 * d)/u$. When processing mono, the PLL is not used therefore the following analysis is for stereo and RDS alone.

The following calculations refer to the number of non-linear operations per output sample in stereo:

$$nlo/output_{mode0} = (4 * (1470 * 5)/1)/1470 = 20$$
$$nlo/output_{mode1} = (4 * (1470 * 9)/1)/1470 = 36$$
$$nlo/output_{mode2} = (4 * (1470 * 800)/147)/1470 = 21.8$$
$$nlo/output_{mode3} = (4 * (1470 * 1280)/147)/1470 = 34.8$$

Table 5 summarizes the number of non-linear operations required to compute one output sample in the stereo path.

Stereo			
	N		
	13	101	301
Mode 0	20	20	20
Mode 1	∘	36	∘
Mode 2	∘	21.8	∘
Mode 3	∘	34.8	∘

Table 5: Stereo non-linear operations per output sample analysis

The following calculations refer to the number of non-linear operations per output bit in RDS:

$$nlo/output_{mode0} = (4 * (1470 * 5)/1)/36 = 816.7$$
$$nlo/output_{mode2} = (4 * (1470 * 800)/147)/39 = 1356.9$$

Table 6 summarizes the number of non-linear operations required to compute one output bit in the RDS path.

RDS			
	N		
	13	101	301
Mode 0	816.7	816.7	816.7
Mode 2	∘	1356.9	∘

Table 6: RDS non-linear operations per output bit analysis

4.2 Hypothesis

- For the mono and stereo paths, since the number of partial products computed for the mono and stereo baseband extraction is equal across all modes of operations, then we should observe their timings to be relatively similar.
- Relating to the above hypothesis for all mono, stereo, and RDS: filters that contain the same amount of partial products should have similar execution times
- For the mono and stereo paths, since the mode 1 IF block size, $(1470 * d)/u$, is the largest among all modes, then we should observe mode 1 to have the longest execution time for IF filters.
- Relating to the above hypothesis for all mono, stereo, and RDS: we should observe the following ordering regarding run time for IF filters, mode 1, mode 3, mode 2, mode 0.
- For all paths, since increasing the number of filter taps increases the number of partials products per output sample per filter, then we should observe a strict increase in execution time when increasing the number of filter taps and vice-versa.

4.3 Measurements

Mono								
	Mode 0		**Mode 1**		**Mode 2**		**Mode 3**	
	Front end 100 KHz filter	Mono baseband filter	Front end 100 KHz filter	Mono baseband filter	Front end 100 KHz filter	Mono baseband filter	Front end 100 KHz filter	Mono baseband filter
Min	6.92945	1.06750	11.7995	1.07145	7.54584	5.69852	12.1444	5.90920
Avg	7.09212	1.10943	12.0769	1.15398	7.61059	6.33165	12.1856	6.59617
Max	8.11734	1.53154	12.5962	2.31424	8.10795	9.99406	12.3481	10.2913

Table 7: Mono filter measurements in milliseconds

Mono				
	13 taps		**301 taps**	
	Front end 100 KHz filter	Mono baseband filter	Front end 100 KHz filter	Mono baseband filter
Min	1.30606	0.153757	19.5835	3.16401
Avg	1.34927	0.162263	19.8947	3.30645
Max	1.53346	0.341032	23.8970	5.88472

Table 8: Mono filter tap comparison filter measurements in milliseconds

Stereo				
		Min	Avg	Max
Mode 0	Front end 100 kHz filter	6.93144	7.06359	7.34357
	Pilot extraction	3.2506	3.38285	5.74843
	PLL	2.02114	2.0543	2.19139
	Stereo subchannel extraction	3.25312	3.31734	4.38227
	Delay block	3.25077	3.33466	4.95971
	Mono baseband filter	1.06258	1.17806	1.68122
	Stereo baseband filter	1.06665	1.10052	1.3902
Mode 1	Front end 100 kHz filter	11.8057	12.0471	12.7348
	Pilot extraction	5.85401	5.89934	6.09128
	PLL	3.59558	3.65306	4.05181
	Stereo subchannel extraction	5.84677	5.92344	6.48879
	Delay block	5.84931	5.89488	6.13867
	Mono baseband filter	1.06395	1.08283	1.24148
	Stereo baseband filter	1.06147	1.0977	1.62611
Mode 2	Front end 100 kHz filter	7.54853	7.7397	8.06121
	Pilot extraction	3.53065	3.56261	3.84281
	PLL	2.20373	2.24048	2.38854
	Stereo subchannel extraction	3.53054	3.57185	3.68626
	Delay block	3.52606	3.62569	3.98607
	Mono baseband filter	5.64685	5.75887	6.1013
	Stereo baseband filter	5.62423	5.69432	6.06484
Mode 3	Front end 100 kHz filter	12.1276	12.2148	12.478
	Pilot extraction	5.66064	5.70737	5.8851
	PLL	3.51125	3.56831	3.93704
	Stereo subchannel extraction	5.65514	5.73152	6.11347

	Delay block	5.64647	5.67725	6.13184
	Mono baseband filter	5.88869	5.95768	6.28867
	Stereo baseband filter	5.84975	5.88647	6.0683

Table 9: Stereo measurements in milliseconds

Stereo						
	13 taps			**301 taps**		
	Min	Avg	Max	Min	Avg	Max
Front end 100 kHz filter	1.30835	1.38161	1.88864	19.2195	19.4633	23.0745
Pilot extraction	0.445291	0.483084	0.998412	9.40128	9.50421	11.77
PLL	2.02114	2.0543	2.19139	2.02554	2.06789	2.27854
Stereo subchannel extraction	0.445531	0.515892	1.07619	9.39934	9.48526	10.046
Delay block	0.44755	0.492488	1.33957	9.39789	9.45134	9.89646
Mono baseband filter	0.151905	0.161592	0.22983	3.14822	3.31326	5.68037
Stereo baseband filter	0.153313	0.199718	0.387828	3.14557	3.2052	3.72223

Table 10: Stereo filter tap comparison measurements in milliseconds

RDS				
		Min	Avg	Max
Mode 0	Front end 100 kHz filter	6.93607	7.20179	8.93147
	RDS subchannel extraction	3.23952	3.28947	3.81687
	PLL	2.0119	2.07149	2.30345
	Generated pilot extraction	3.24483	3.30722	4.33101
	Delay block	3.25598	3.30194	3.86799
	RDS baseband filter	11.2681	11.4911	12.8319
	RRC filter	1.25661	1.27138	1.49211
Mode 2	Front end 100 kHz filter	7.5568	7.86967	11.7052
	RDS subchannel extraction	3.527	3.60075	4.03777
	PLL	2.19755	2.25163	2.39012
	Generated pilot extraction	3.53016	3.59147	4.02471
	Delay block	3.53333	3.63859	4.07321
	RDS baseband filter	6.04977	6.11815	6.79689
	RRC filter	0.702695	0.729916	1.09708

Table 11: RDS measurements in milliseconds

RDS						
	13 taps			**301 taps**		
	Min	Avg	Max	Min	Avg	Max
Front end 100 kHz filter	1.31076	1.39226	1.77703	19.4617	19.9697	28.2702
RDS subchannel extraction	0.443327	0.464825	0.860952	9.38808	9.50325	10.0684
PLL	2.02114	2.0543	2.19139	2.02554	2.06789	2.27854
Generated pilot extraction	0.443495	0.476822	0.865656	9.39951	9.62712	10.83
Delay block	0.443179	0.472549	0.708824	9.39743	9.54253	10.5167
RDS baseband filter	1.58839	1.67833	2.17149	4.11421	4.16245	5.22667
RRC filter	0.173127	0.183908	0.359421	3.62556	3.71509	4.64929

Table 12: Stereo filter tap comparison measurements in milliseconds

4.4 Discussion

Our measurements relatively agree with our hypotheses. Mode 1 had on average the longest run time for the IF block size filters; mode 3 followed close behind then mode 2 and finally mode 0 had the shortest run time. An example of an IF block size filter is the front-end filter. Although the filter operates on samples at the RF sample rate, due to the decimation, the number of effective operations scales with the IF block size.

From the measurements, in mode 1 the front-end filter takes on average 12.0471 ms, mode 3 takes 12.2148 ms, mode 2 takes 7.7397 ms, and mode 0 takes 7.06359 ms. There appears to be a discrepancy between the hypothesized and measured result since mode 3 took on average 168 μs longer. However, the IF block size of mode 1 and mode 3 are similar only differing by 430 samples. As a result, this discrepancy can be due to the many causes of code execution timing variation, especially since in every other case of the IF block size filters mode 1 was on average strictly slower.

However, there did exist discrepancies that indicated more profound causes. By the hypotheses, the baseband extraction filters at the end of the mono and stereo paths for all modes should have the same execution time. Yet, as discovered in the measurements the baseband filters for mode 2 and mode 3 took nearly 6x the time to complete.

The initial reasoning for this behaviour was compiler optimization since in our resampler/filter in modes 0 and modes 1 the upsampling coefficient is one. This means that any modulo or division operation should be skipped by level 3 optimization. Yet, the upsampling coefficient is not determined on compile time because the coefficient is resolved when the mode of operation is selected which is a run-time decision. Meaning the compiler should not optimize the modulo and division operations out. The next theory was that the modulo and division operations had variable clock latency within the processor. Yet again, after testing the running times for the division and modulus operators on divisors ranging from 1-1000 it indicates that the operations were fixed latency. Our final theory was that perhaps the compiler is optimizing the code but in a different way. We thought the compiler might realize an upsampling coefficient of one is a possible input parameter. It could implement a way to skip the modulus/division operations depending on a comparison. After using the *-S* to analyze the assembly, we could not identify any part of the code that attempts to avoid the modulus and division operations.

Furthermore, one very unexpected behaviour was when we increased the number of filter taps from 101 to 301. Overall, for all the filters we saw a strict increase in execution time which was expected. Yet, when testing the fractional resampler under this increase in RDS mode 0 the execution time actually decreased contrary to our hypotheses. We believe this must be related to the previous discrepancy since it again involves the fractional resampler.

5 Proposal for Improvement

In regards to future improvement, allowing users to listen to AM radio would increase the user experience. The RTL-SDR driver is capable of sampling AM frequencies; therefore, there is no reason to assume implementing AM to be not feasible. Analog AM receivers recover message signals by filtering specific AM channels then performing envelope detection and DC shifting. To implement AM radio in a software environment, we must create functional code blocks analogous to their analog counterpart. We have already defined functions to perform frequency filtering, the next step would be to determine how to achieve envelope detection in a digital domain. Once the envelope has been recovered we will perform DC shifting on the envelope vector. DC shifting can be done by subtracting each element in the vector by 1, however, envelope detection would be a bit more involved.

Furthermore, setting up a GUI would not only further improve the user experience, but it would also help us in developing the project further. One issue we ran into was having to recall how to setup the CLI execution with each auxiliary programs parameters between testing sessions. Abstracting the CLI program execution process behind a GUI would reduce issues and would not require the end-user to know the parameters for the auxiliary programs. This would be implemented by using one of the many GUI libraries in Python such as Tkinter.

Lastly, some optimization of our source code might involve implementing the delay blocks in stereo and RDS as vector shifts. This slight change would reduce the execution time of the stereo and RDS paths greatly. Also, converting our convolution-based filters to FFT-based filters would further reduce the execution times of all paths. To ensure signal quality, we must perform proper windowing of the block spectrum.

6 Project Activity

Week	Breakdown
Feb 12	Entire group finished Lab3, conceptualized project and assigned roles. Amr, Omar – Implement RF-front end, Mono 0 & 1 in C++.
Feb 26	Amr, Anthony– Build infrastructure for project implementation in C++. Hydar, Anthony – Resampling concept, model
Mar 4	Anthony, Hydar – model, test and implement polyphase filter resampler
Mar 11	Hydar – Mono mode 2 & 3 Amr – Begin Multithreading, Stereo Anthony – Begin multithreading. Omar – Stereo Support
Mar 18	Hydar – RDS modelling Anthony – Multithreading optimization Amr – Stereo, PLL implementation, Multithreading Omar – Stereo Support
Mar 25	Hydar, Amr- Implement RDS in source code Anthony – Timing analysis Omar – Stereo Stress testing
Apr 1	All members worked on report

Table 13: Week breakdown

Member	Critical Challenge
Amr	PLL drifting problem, PLL would lose lock and drift in unanticipated ways, interfering with DSB-SC demodulation in Stereo and RDS.
Anthony	Managing race conditions between audio and RDS consumer threads, and eliminating deadlock to allow resource sharing.
Hydar	RDS channel extraction and application layer, as extracting BPSK data from weak signals was imperfect, and errors propagating through the digital decoding make frame synchronization complicated.
Omar	Implementing branchless code in RF front end to avoid performance pitfalls such as if-else statement branch prediction failure.

Table 14: Member critical challenges

7 Conclusion

Over the past three months, our team has undertaken a project that significantly broadened our technical and non-technical skill sets, closely mimicking the complexities encountered in real-world project implementations. This journey through developing a real-time SDR system has enhanced our teamwork capabilities and deepened our understanding of concepts surrounding digital filtering, signal processing, frequency modulation, coding, debugging, and optimization.

The practical application of theoretical knowledge in signal processing, FM technology, and digital communication, alongside gaining familiarity with the Software Development Life Cycle and source-code management, has improved our technical skills. Furthermore, the project has improved our collaboration, project management, and problem-solving skills, laying a solid foundation for a future in the computing industry. We are very grateful for the invaluable experiences and skills acquired, which will undoubtedly help our professional development.

8 Final Remarks

We have more to say but we have ran out of page real-estate. ☹

9 References

[1] K. Cheshmi and N. Niccolici, “COE3DY4 Project Real-time SDR for mono/stereo FM and RDS.” McMaster, Hamilton, Feb. 12, 2024