COE3DY4 – Project Report

Samuel Parent - 400336235
Ivan Lange - 400303704
Hamza Siddiqui - 400407170
Yash Bhatia - 400362372

Group 30
April 5, 2024

Introduction

This project concerns the development of an FM Software-Defined Radio to be used on a Raspberry Pi 4. The data would be provided by the RTL-SDR program as raw radio samples, which then must be decoded. This includes parsing the stereo audio and outputting it across two channels in real-time, and simultaneously parsing the Radio Data System (RDS) data and streaming it to the console. To make this work in real-time, multi-threading, block processing and various other optimizations are also used.

Project Overview

The big picture of the project is to use radio frequency (RF) signal processing to decode frequency modulated (FM) data. FM radio is a broadcasting standard where channels broadcast 200 kHz apart, with center frequencies from 88.1 MHz to 107.9 MHz. Pre-modulation, these broadcasts send data in frequency blocks across their ± 100 kHz range, known as subcarriers.

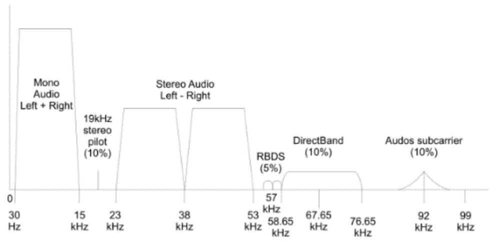


Figure 1: Frequency spectrum of an FM channel

For this project, we employ a Software-Defined Radio (SDR) receiver for demodulation and decoding of data, which can perform the same functions as analog hardware components in the discrete-time domain for reasonable sample (discretization) rates. To achieve this, processing tools are chained in a signal-flow graph. The most common of these tools is the finite impulse response (FIR) filter. An FIR filter attenuates desired frequency ranges and amplifies others. FIR filters sample the $sinc$ function and the number of samples, or taps, of the function determines the performance of the filter. The signal flow graph begins with 8-bit unsigned integers interleaved in-phase and quadrature data. The RF Front-End extracts the FM data through low-pass filtering, demodulates it, and decimates it to reduce the sample rate to an intermediate frequency (IF).

This data is sent to three different paths. The Mono path extracts the mono subcarrier, alters its sample rate to the audio output frequency, and sends it to the standard output. The Stereo path extracts the Stereo subcarrier and the pilot tone. As the Stereo data is transmitted using double-sideband suppressed carrier (DSB-SC), the carrier is extracted using the pilot tone, which is transmitted in-phase with the signal. A Phase-Locked Loop (PLL) is used to create a pure wave at 19 kHz in-phase with the pilot tone, and a Numerically Controlled Oscillator (NCO) converts this to the 38 kHz carrier. The carrier and subcarrier are mixed, and their sample rate is altered to match the Mono audio, which is delayed to match the stereo data. The difference between the Mono and Stereo data is sent as the left audio output, while the sum is sent as the right audio output.

The final part is the RDS data. RDS is a standard supporting the transmission of binary program metadata over FM radio. To extract this, the RDS subcarrier is extracted with a band-pass filter, and it is squared and altered to feed into a PLL, regenerating its carrier component at 57 kHz. This data is mixed and low pass filtered at 3 kHz, after which it is resampled to support the RDS symbol rate of 2,375 symbols/second. This data is fed into a root-raised cosine filter to minimize inter-symbol interference. This data is sampled and decoded using Manchester and Differential decoding, and 26-bit blocks are synchronized to expected frames described in the standard using their 10-bit trailing parities. The output at 90 bytes per second is then decoded using an application layer to display program information to the user.

The sampling rates and frequencies are not constant- there are four modes of operation from 0 to 3, and their corresponding sampling rates and frequencies must be used when resampling.

Implementation Details

Settings for group 30	Mode 0	Mode 1	Mode 2	Mode 3
RF Fs (KSamples/sec)	2400	2304	2400	2880
IF Fs (KSamples/sec)	240	384	240	320
Audio Fs (KSamples/sec)	48	32	44.1	44.1

Table 1: Group 30 Custom Constraints

		RF	Audio		RDS		
Mode	Block Size	Downsample	Upsample	Downsample	Upsample	Downsample	SPS
0	76800	10	1	5	247	1920	13
1	86400	6	1	12			
2	96000	10	147	800	19	64	30
3	115200	9	441	3200			

Table 2: Various Settings per Mode and Channel

Lab Work

In Lab 1 we created our own functions for generating LPFs and convolution in Python. These were based on the class notes. In Lab 2, the functions were ported to C++. This was challenging as we had to consider pre-sizing the vectors we used and manage their memory. We also began with block processing, so we added state saving to convolutions. This is needed to maintain continuity in the processing of successive blocks and was used throughout the project.
In Lab 3, we had our first chance to process FM data in Python and wrote our own FM demodulation function described below. We used pre-recorded IQ samples for testing and analysis and got a Mono output. This was the foundation for our project model and implementation.

Block Size

Block processing is necessary in the real-time implementation of the SDR since the data being fed into the software needs to be output as live audio. As a result, the size of data in each block plays a key role in ensuring that the real-time requirement is met and adequate data is available to perform the required processing per block. Therefore, a block size that equates to 22 ms - 44 ms worth of IQ samples is found to meet the criteria mentioned above. Using the custom group constraints provided to us, as seen in Table , the required resampling parameters were calculated, as seen in Table . These were determined by finding the greatest common divisor between the input and output sample rates.

The block size is calculated based on these resampling factors. For modes one and three, it is sufficient to ensure that the block size is a multiple of the audio and RF downsampling factors. However, since modes zero and two are required to run stereo and RDS, it is crucial that the block size provided post RF front end is a multiple of the downsampling rates for RDS and audio. This ensures that a proper integer number of samples is available and data between blocks is not lost. The block sizes presented in Table meet all of the requirements mentioned.

Filters (APF, BPF, LPF, RRC)

Filtering is relied heavily upon throughout the entire signal flow graph. Various finite impulse response (FIR) filters including all-pass, band-pass, low-pass, and root-raised cosine are used to process data in various parts of the signal flow.

These digital filters are implemented in two steps. First, the impulse response for each filter is generated at the beginning of the program. The low-pass and band-pass coefficients are generated by taking samples of the *sinc* function, since it corresponds to a rectangular brick-wall function in the frequency domain. To do this, a parameter is defined for the number of taps in the filter which equates to the number of samples of the sinc function captured. For the band-pass filter, this brick wall is shifted in the frequency domain by using modulation. Next, a convolution function is used each time data needs to be filtered in a block; this is done by convolving the existing impulse response with the data.

In *filter.cpp* and *filter.h*, various functions are implemented for impulse response generation and convolution. Many of these functions also perform resampling operations to accelerate the process in effort to meet the real-time constraint. The all-pass filter is implemented as a delay block (shift register object) that emulates the delay of the other filtering to synchronize data as it moves through carrier extraction to mixing. The root-raised cosine coefficients are implemented with a value of beta to pulse-shape the incoming RDS data.

RF Frontend

The RF front-end takes the IQ samples from the *rtl_sdr* program and deinterleaves them to get I and Q vectors. These are filtered to the baseband with a 100 kHz cutoff, and downsampled to a rate that fulfils the Nyquist criterion. These samples are demodulated as described below, and enqueued for other paths.

FM Demodulation

The I-Q samples in the RF front-end are the output of an analog mixer and LPF. The message is encoded as frequency changes, and therefore the message can be reconstructed by plotting the frequency of the wave at a given time instance- which is the derivative of the phase of the wave. For an efficient implementation, we replace the operations $p(t) = \arctan\left(\frac{Q}{I}\right) \Rightarrow m(t) = \frac{\mathrm{d}}{\mathrm{dt}}p(t)$ with a single operation $m(t) = \frac{I(t) * \frac{dQ}{dt} - Q(t) * \frac{dI}{dt}}{I^2(t) + Q^2(t)}$ which was derived in Lab 3.

PLL + NCO

A PLL or Phase Locked Loop is used to create a clean single frequency wave that has a constant phase difference with an input wave. The frequency of the output is a multiple of the input frequency. To implement our PLL we used a second-order control system with damping ratio of $\frac{1}{\sqrt{2}}$ and unity gain, giving $K_p = 2.666 * b_{Norm}, K_i = 3.555 * b_{norm}^2$[1]. The PLL uses a phase detector to check the error between the generated wave and input signal, and the loop filter uses integrator and proportional terms to

[1] b_{Norm} is the normalised bandwidth- higher bandwidth results in a faster lock, but a noisier output.

update the phase estimate. This is fed to the internal oscillator, which uses a counter called t_{offset} to generate the wave $w = \left(2\pi f t_{offset}\right) + \theta_{est}$ and update the feedback. Finally, the Numerically Controlled Oscillator multiplies this wave with the desired scale factor (only the real component is returned). To preserve the lock across blocks, we return the last NCO sample, as well as the loop filter output, θ_{est}, and t_{offset}.

Resampling

When having to account for downsampling by a non-integral amount for certain modes, a resampler was implemented which upsamples and downsamples by two factors that result in the desired ratio. To have this be fast, we integrated it with convolution such that the index we extract from the input can be calculated each time and fill the output vector accordingly. To take an input every decimation interval, we computed a phase based on the current index and used modulus to be a multiple of the upsampling factor. This was subtracted in our input index calculation to give an integer; if it was ever negative, we extracted values from state. This modification was verified by comparing it to a manual upsampler and downsampler combination, which yielded the same results but failed to meet timing constraints, while the final efficient resampler implementation met both criteria.

When resampling occurs, it is important that the proper impulse response generation function is used. Since resampling always implements a low-pass filter, the *impulseResponseLPF* function is overloaded with an upsampling parameter to scale the number of taps and amplitude of the response accordingly. Additionally, the sampling frequency is also upsampled accordingly, and the output block size is scaled by the upsampling and decimation factors' ratio.

Mono Path

After receiving the demodulated data, the mono path generates coefficients based on the mono cutoff frequency with an LPF. Within the loop, demodulated samples are dequeued and are convolved with the impulse response coefficients to obtain the float audio data. This data is then written to stdout after scaling to 16-bit signed ints.

Stereo Path

The stereo path uses the LPF to generate stereo LPF coefficients, then has two BPFs to generate the stereo BPF coefficients and the coefficients for the pilot. In the loop, the demodulated samples are dequeued, and from the float audio data in mono path, a delay matching APF is applied to demodulated samples. We then use resampler to generate the mono data in float format, then convolve for the two BPFs with stereo and pilot tone. Next, we apply the PLL and get the NCO out with our filtered pilot, apply a mixer with NCO, stereo BPF data, and the mixer gain of 2. Finally, we LPF the stereo data and split the left and right channels using the mono data and adding and subtracting the stereo LPF data for left and right respectively. Finally, we write an entire block of this data to stdout. Issues arose for this portion as push back was applied in the writing stage when we already resize the output vector, hence this was adjusted to be index-based and fill the vector as is. This also made the writing faster as we avoided the resizing overhead for each push back performed.

Threading & CPU Affinity

To obtain data fast enough and run audio and RDS data simultaneously, threading was implemented following the structure seen in the figure. As soon as the RF front-end produced demodulated samples, they were pushed into two queues: one shared between front-end and audio threads, and another shared between front-end and RDS threads.

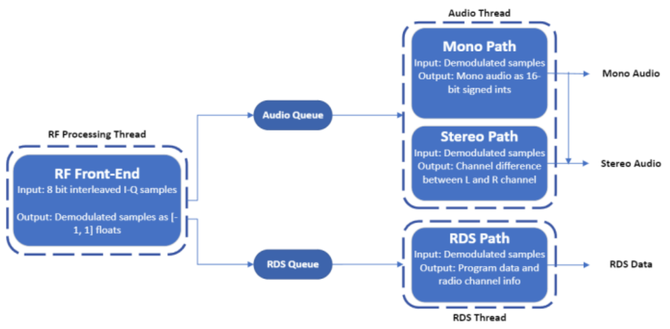

RF Processing Thread
RF Front-End
Input: 8 bit interleaved I-Q samples
Output: Demodulated samples as [-1, 1] floats
Audio Queue
RDS Queue
Audio Thread
Mono Path
Input: Demodulated samples
Output: Mono audio as 16-bit signed ints
Stereo Path
Input: Demodulated samples
Output: Channel difference between L and R channel
RDS Path
Input: Demodulated samples
Output: Program data and radio channel info
RDS Thread
Mono Audio
Stereo Audio
RDS Data

Figure 2: High-level diagram of threading

For a clean implementation of this to work effectively, all threading locks, mutex, etc. were abstracted within a separate queue template class. This class supported the relevant enqueue and dequeue operations. When accessing the queue shared between producer and consumer, mutex was used and was locked when accessing in either enqueueing and dequeuing, and condition variable waiting was used when queue is either full or empty as we cannot enqueue in a full queue nor dequeue an empty queue. Finally, the threads would notify each other when operations were complete and unlock mutex accordingly. This all allowed for well-defined behavior and resulted in a thread-safe queue. To ensure both consumer threads used the same vectors, their check sums were computed and printed to standard error, and both were seen to always be the same. As well, running the threads, as well as with RDS thread with audio playing in the background, were audibly verified to run as expected.

CPU affinity was used to eliminate thrashing and context switching caused by threads jumping between cores. This was done by assigning each thread a corresponding CPU core to run on during program execution. The results were verified via *htop* and improved the efficiency drastically.

RDS Channel Extraction

The RDS channel is extracted by using a BPF to get the entire channel. The channel is then squared to increase its frequency to twice the center frequency (57 kHz). This is fed to the PLL explained above with a scale factor of 0.5. Due to the low signal energy, we had many problems with carrier generation, but we reduced the b_{Norm} to 0.0005 to get a clean signal. This is mixed and filtered like the stereo path, and a rational resampler is used on the LPF to reduce the sampling rate to $sps * 2375$. Our Mode 0 factors for upsampling and downsampling were 247 and 1920 for 13 sps, and for Mode 2 were 19 and 64 for 30 sps.

The resulting wave is fed through a Root Raised Cosine (RRC) filter. Receiver RRC filters are used to minimize inter-symbol interference by amplifying the difference between symbols at given intervals (here at every 1/2375 seconds). We used the RRC response given in class for our implementation.

PLL Locking

The PLL is an integral portion of the signal flow graph in RDS. If the PLL has not locked properly, there is no chance of recovery in the remainder of our signal flow graph. To assure proper PLL locking, we give it ~20 blocks for locking where we skip the remainder to process the next block.

Block Aggregation

In order to simplify our implementation passed the point of the RRC filtered RDS waveform, we chose to do block aggregation. Due to our group's custom constraints, it was found to be impossible to have an integer number of symbols per block (spb) for mode 2.

With the block size we settled on for mode 2, 86400, we were left with 47.5 spb. By simply aggregating 2 blocks we are able to achieve an integer spb of 95. This greatly simplified our clock and data recovery (CDR) implementation.

This is not the end of our aggregation; in our mode 0 we are now left with 76[2] spb which differs in parity with our mode 2. To simplify our bitstream recovery, we wanted to have an even spb in both modes. Instead of aggregating 4 RRC filtered RDS waveform blocks, we decided to aggregate a further two blocks after the CDR as the number of samples saved at this point decreases by the samples per symbol.

The block aggregation makes use of modulo 2 counters where data is copied to an aggregation vector until counter overflow where the remainder of the signal flow graph is run.

As a bonus, this block aggregation guarantees that the number of bits per block surpasses 51[3], which is a requirement in our frame synchronization implementation.

CDR

Once the PLL has locked, we take approximately 10 blocks to adjust our sampling start. This is done by finding the value closest to 0 in the block and taking its modulo with the samples per symbol (sps). We take the average of this value over 10 blocks, and we do not recalculate this value unless we lose frame synchronization.

With the Sampling start adjustment, the remainder of CDR simply involves skipping sps samples with the correct adjustment with some state saving.

See the block aggregation section for more information on how we handle the problem of non-integer spb.

Bitstream Recovery

The bitstream recovery is done in 2 modes. The first mode is for initialization and will perform Manchester decoding twice: once with offset 0 and once with offset 1. We count the number of high-high (HH) and low-low (LL) symbol pairs we encounter, compare the result with each offset. If offset 1 were to have a lower count of HH and LL, we increment its score and decrement the score of offset 0. This is done till one of the scores hit a threshold where we set a bitstream select variable to winning offset. This scheme makes it such that if we achieve bitstream selection it is because one bitstream is heavily favored the other, removing random chance.

During Manchester decoding, we handle HL and LH in the typical manner. With a HH and LL, we do a supplementary check to search for weak HL's or weak LH's[4]. This is possible since the data is still in its analog form at this point in the signal flow graph. Doing this allows us to recover many of the points that would be near the vertical axis crossing in our constellation diagram.

---

[2] This number comes from 38 symbols per block for 2 aggregate blocks.

[3] This number comes from 2*26-1 where 26 is the number of bits per group.

[4] A weak High-Low in a High-High symbol pair is one where the first sample is larger than the second sample.

Bitwise vs Block wise Frame Synchronization

Like many of the functional blocks in the RDS path, the frame synchronization exists in the initial mode and a common case mode.

In the initial mode, we perform a bitwise search for any of the syndromes. Once a syndrome is found, we set the expected next syndrome, and we start 2 separate counters; one of which counts the number of bits since the last found syndrome, and another for the number of correct syndromes found. Not finding the correct syndrome 26 bits later will restart this search process. Once we reach a threshold, we switch to block wise frame synchronization. State saving is used to pass along the last 25 bits from one block to the next block.

In block-wise synchronization, we have information of the exact 26 bits where we expect a syndrome to exist. This value gets updated and passed along as a state across calls. As with bitwise frame synchronization, we pass the trailing bits from one block to the next through a state, however here it is not a constant value (previously 25), instead it is intuitively related to the remainder of the quotient of the total number of bits being operated on with 26.

Block-wise synchronization is where we extract information from the groups themselves. This part is quite straight forward and simply requires an understanding of the RDS standard and some state saving.

More interestingly, we perform score tracking during frame synchronization with what we call the rubbish streak, rubbish score, and rubbish threshold. The rules are as follows:

- The rubbish score is strictly greater than 0.
- The rubbish streak increments by 1 with a bad syndrome.
- The rubbish score increments by 10*(rubbish streak) with a bad syndrome.
- The rubbish score decrements by 1 with a good syndrome.
- The rubbish streak resets to 0 with a bad syndrome.

This rubbish tracking scheme is used to tell when we should restart the PLL locking, CDR offset setting, and initial frame synchronization. This happens when the rubbish score is greater than the rubbish threshold.

Testing

Throughout the project, we used the GoogleTest framework to make comparisons between our python model and our C++ source code implementation. This proved to be a rabbit hole as we struggled to get the data samples to match closely enough to pass our test cases.

The setup included a test fixture which allowed us to read data arrays from a .dat file. This was used to read in an array generated by our Python model which acted as the reference, and another array generated by the C++ source code which was the data to be tested. In theory this would allow us to quickly tell exactly where bugs were being introduced in our signal flow graph.

A large oversight was made during this development which was only found after this method was abandoned; the python model did not use an identical filtering algorithm as was used in the C++ implementation. This is on top of the fact that we were using double precision floating point values in the model while using single precision floating point values in the C++ implementation.

Analysis and Measurements

Analysis of Multiplications and Accumulations Performed

Using our block size per mode and decimation and upsampling factors, a sample calculation is shown below for computing the number of operations per audio sample for Mono Mode 0:

$$ops_{RF} = \frac{block\ size}{RF_{decim}} * N_{taps} = \frac{76\ 800}{10} * 101$$

$$ops_{Mono} = ops_{RF} + \frac{block\ size * N_{taps}}{IQ_{factor} * RF_{decim} * Audio_{decim}} = \mathbf{857\ 088}\ \boldsymbol{operations}$$

$$\frac{ops}{sample} = ops_{Mono} \div \frac{size_{intermediate} * Audio_{ups}}{Audio_{decim}} = \frac{857088}{768} = \mathbf{1111}\,\boldsymbol{\frac{ops}{sample}}$$

Path for Processing	Operations per audio sample
Mono Mode 0	1,111
Mono Mode 1	2,525
Mono Mode 2	1,200
Mono Mode 3	1,574
Stereo Mode 0	2,121
Stereo Mode 1	4,949
Stereo Mode 2	2,300
Stereo Mode 3	3,032

Table 3: Operations per audio sample

Path for Processing	Operations per bit
RDS Mode 0	88,202
RDS Mode 2	96,770

Table 4: Operations per bit for RDS

Notably, there are other operations performed on the block such as mixing of carrier and sub-carrier, squaring nonlinearity, and operations to derive the filter coefficients. However, these operations are (at worst) $O(intermediate\_block\_size)$, so their impact is negligible compared to the above which are $O(block\_size * N_{taps})$. Additionally, operations performed after RRC on RDS are on a very low bitrate, so they haven't been considered.

Analysis of Nonlinear Operations Performed

Within the PLL and NCO computations, nonlinear `atan2`, `cos` and `sin` are used while iterating over the PLL input based on the demodulated data sizes which vary per mode. With two cosine calls and one call for atan2 and sin per iteration, a sample calculation for Mode 0 can be performed as follows:

$$fmPLL\ input\ size = \frac{Block\ size}{IQ\ factor * RF\ Decimation} = \frac{76{,}800}{10*2} = 3{,}840$$

Hence, the atan2 and sin operations are performed 3840 times and cos is performed 3840*2=7680 times per block in mode 0. In the RDS path, the same PLL iteration occurs on the RDS carrier. The rest of the results per block can be seen in the table.

Function	Stereo Mode 0	Stereo Mode 1	Stereo Mode 2	Stereo Mode 3	RDS Mode 0	RDS Mode 2
atan2	3,840	7,200	4,800	6,400	3,840	4,800
cos	7,680	14,400	9,600	12,800	7,680	9,600
sin	3,840	7,200	4,800	6,400	3,840	4,800

Table 5: Operations for various functions in modes

Notably, the RDS processing takes 20 blocks to 'lock' the PLL. The above values are therefore the number of iterations for processing the average block, after the setup is completed.

Running Time Analysis

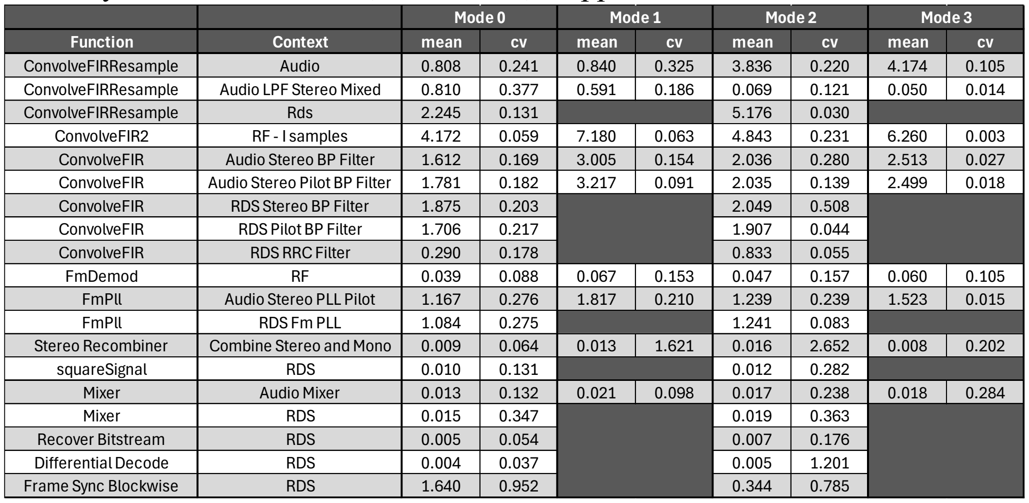
The mean runtimes in milliseconds for various functions (and where they are used) is seen in Table 4, along with their normalized standard deviations (displayed as *cv*) for each of the four modes. It is important to note that the block sizes for the modes are different, as seen in Table 3, as well as their RF, audio, and RDS decimations. The results in the table are therefore deceiving and must be analyzed further.

Taking the results of *convolveFIR2* for I samples in the RF front end as an example, normalizing the mean by the block size and the RF decimation relative to mode 0 gives the following relative run times. For *fmPLL*, the normalized runtime in the four modes is seen below.

Function	Mode 0	Mode 1	Mode 2	Mode 3
convolveFIR2	1.00	0.92	0.93	0.90
fmPLL	1.00	0.83	0.85	0.78

Table 6: Normalized Runtimes for Functions in different modes

In an ideal environment, the normalized runtime should be 1.00 for each of the modes, however, error in the measurements and internal optimizations being made affect this. ConvolveFIR2 shows that the runtimes are relatively the same. However, for fmPLL, it appears to be non-linear.



		Mode 0		Mode 1		Mode 2		Mode 3	
Function	**Context**	**mean**	**cv**	**mean**	**cv**	**mean**	**cv**	**mean**	**cv**
ConvolveFIRResample	Audio	0.808	0.241	0.840	0.325	3.836	0.220	4.174	0.105
ConvolveFIRResample	Audio LPF Stereo Mixed	0.810	0.377	0.591	0.186	0.069	0.121	0.050	0.014
ConvolveFIRResample	Rds	2.245	0.131			5.176	0.030		
ConvolveFIR2	RF - I samples	4.172	0.059	7.180	0.063	4.843	0.231	6.260	0.003
ConvolveFIR	Audio Stereo BP Filter	1.612	0.169	3.005	0.154	2.036	0.280	2.513	0.027
ConvolveFIR	Audio Stereo Pilot BP Filter	1.781	0.182	3.217	0.091	2.035	0.139	2.499	0.018
ConvolveFIR	RDS Stereo BP Filter	1.875	0.203			2.049	0.508		
ConvolveFIR	RDS Pilot BP Filter	1.706	0.217			1.907	0.044		
ConvolveFIR	RDS RRC Filter	0.290	0.178			0.833	0.055		
FmDemod	RF	0.039	0.088	0.067	0.153	0.047	0.157	0.060	0.105
FmPll	Audio Stereo PLL Pilot	1.167	0.276	1.817	0.210	1.239	0.239	1.523	0.015
FmPll	RDS Fm PLL	1.084	0.275			1.241	0.083		
Stereo Recombiner	Combine Stereo and Mono	0.009	0.064	0.013	1.621	0.016	2.652	0.008	0.202
squareSignal	RDS	0.010	0.131			0.012	0.282		
Mixer	Audio Mixer	0.013	0.132	0.021	0.098	0.017	0.238	0.018	0.284
Mixer	RDS	0.015	0.347			0.019	0.363		
Recover Bitstream	RDS	0.005	0.054			0.007	0.176		
Differential Decode	RDS	0.004	0.037			0.005	1.201		
Frame Sync Blockwise	RDS	1.640	0.952			0.344	0.785		

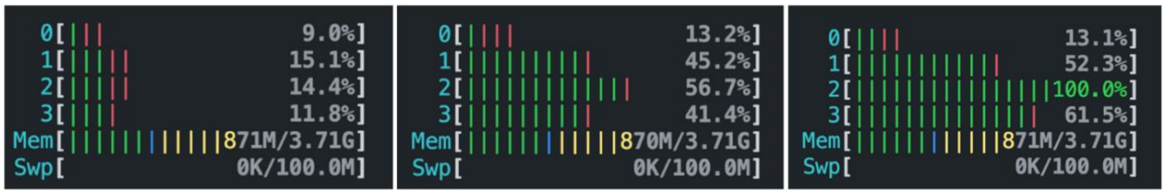
Table 7: Runtimes for building blocks of signal flow graph

As was spoken about in the implementation details, we assigned each of the main threads a specific core. This has allowed us to do additional analysis on the bottleneck of the signal flow graph as it relates to the number of taps. In **Error! Reference source not found.** shown below, we see that as the number of taps i ncreases to 301, core 2, which is associated with the RF front end thread, is at 100% CPU utilisation. This result makes sense as the RF Front end processes the largest number of samples through its low pass filter while all other threads see block sizes that are downsampled by the RF decimation factor.

It can also be seen that the number of taps plays a large role in the compute time of the program. As we decrease the number of taps from 101 to 13 we can see that the CPU utilization goes from 45-50% to 10-15% for all cores.

Core	Thread
0	All Other Processes
1	RDS
2	RF Front End
3	Audio

Table 8: CPU Assignment for Threads



```
  0[|||                 9.0%]
  1[|||||              15.1%]
  2[|||||              14.4%]
  3[||||               11.8%]
Mem[||||||||||||871M/3.71G]
Swp[                0K/100.0M]
```

```
  0[||||               13.2%]
  1[||||||||||         45.2%]
  2[||||||||||||||     56.7%]
  3[||||||||||         41.4%]
Mem[||||||||||||870M/3.71G]
Swp[                0K/100.0M]
```

```
  0[||||               13.1%]
  1[||||||||||||       52.3%]
  2[||||||||||||||||||100.0%]
  3[||||||||||||||||   61.5%]
Mem[||||||||||||871M/3.71G]
Swp[                0K/100.0M]
```

Figure 3: CPU Utilization during RDS Channel Execution for 13 taps (left), 101 taps (middle), 301 taps (right)

An experiment was carried out where we measure the runtime of the convolveFIR function for mode 0 in the RF front end for varying number of taps (while also varying the number of taps for all other filters). As can be seen in **Error! Reference source not found.**, the runtime increases linearly with the number of t aps.

As this experiment was carried out, we were able to observe its effects on the audio quality and the ability to obtain RDS data. Surprisingly, we were able to obtain valid RDS data for as low as 7 taps. The audio is also still audible at low numbers of taps, though it introduces a large amount of high frequency noise into the signal. This makes sense as the side lobes of our filters are rather large for low tap numbers. Naturally the audio quality improved significantly as the tap sizes increased. As the taps reached 201+ we started to miss our real time requirements[5].

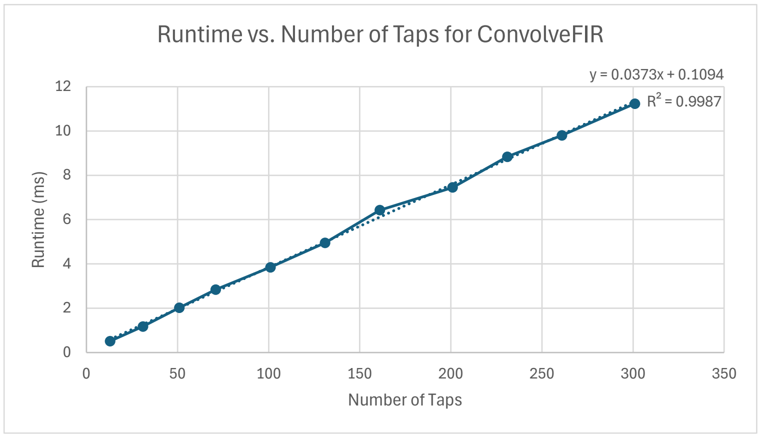


Figure 4: Runtime vs. Number of Taps for ConvolveFIR

[5] Aplay underruns

Proposal for improvement

Many features can be added to the software system to benefit users of the system as well as developer productivity. First, a user interface (UI) could be implemented to have graphical control of the control software. This would allow users to select their desired FM channel, stereo/mono listening, display of RDS data, and modes of operations all on a graphical user interface. This facilitates the operation of the SDR as general users are not familiar with the command-line interface that is required for operating the device. Additionally, when a GUI is introduced, a limit is put on user behavior to ensure that no error is introduced when calling the control software. Next, to improve the developer experience, pre-processor directives can be included that introduce a variety of debugging outputs and logging functionality. This way, custom debugging can be added for any code of interest and enabling them is as simple as changing macros in the code. This way, the real-time constraint is not affected by the debugging logic, and the developer can decide when to enable this functionality.

Many different areas of improvement became evident throughout the project, especially in terms of efficiency. For example, the resampler would compute the input index in the inner loop, resulting in many computations happening very often, despite it being possible to be done more efficiently. This can be achieved through computing the index in the outer loop in a different manner and then decrementing it for each iteration ran in the inner loop, resulting in fewer multiplications done in resampling.

Another example of efficiency improvement lies in the tail end of the RDS path. Many of the operations are done on bits which we keep in an array of boolean values. We could save on a large amount of memory accesses by instead packing bits into a uint64 and using bit masking to access any given bit. This also reduces the amount of memory required to run the program.

Conclusion

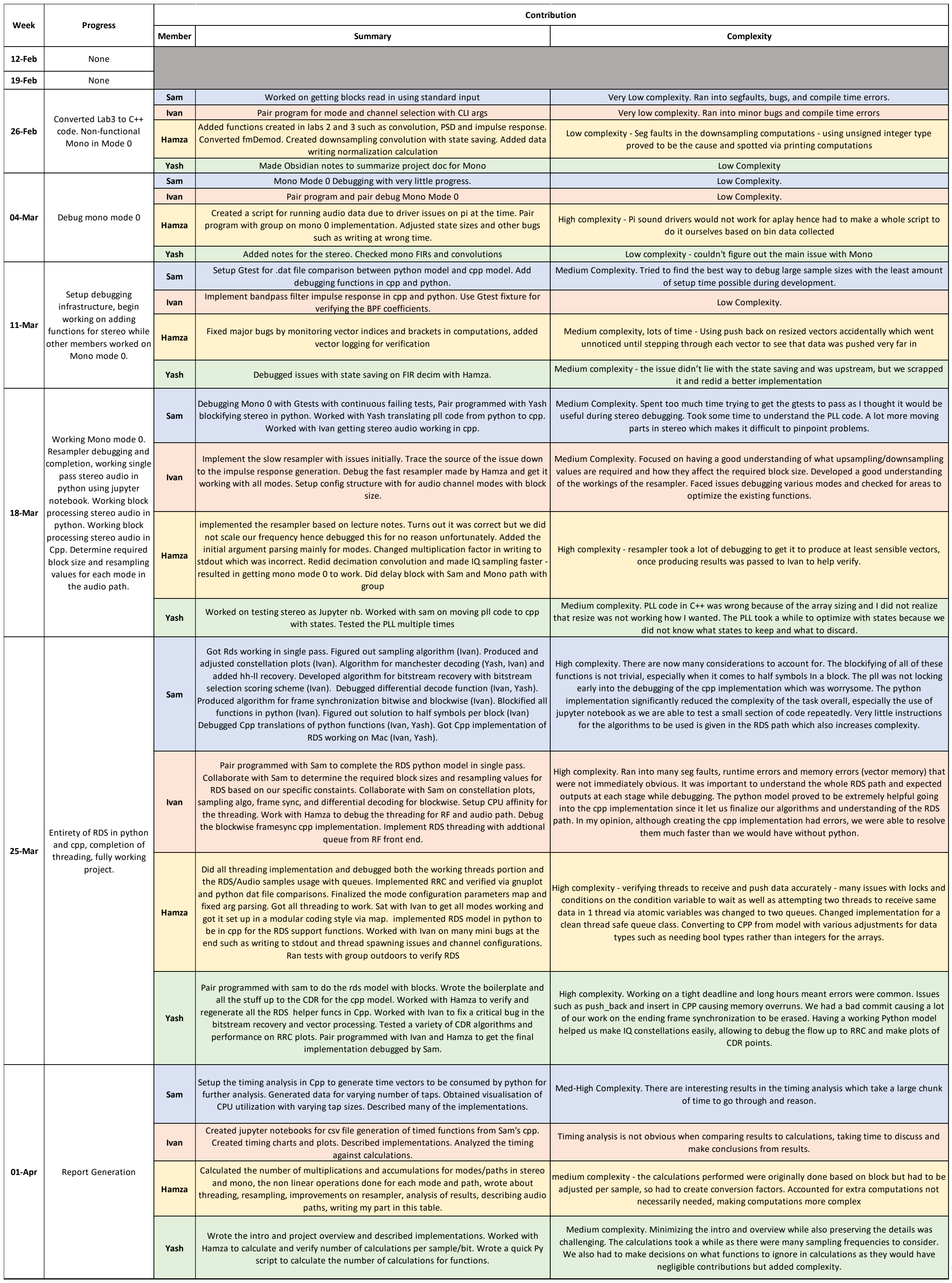
Having finished the project entirely, it is now possible to understand the complexity of a real-time system that interacts with actual data. The integration of various software methodologies was done successfully allowing for a thorough understanding of various concepts taught throughout our program so far. The theory covered was now applied in a practical setting and our learning was consolidated. Just as the famous “excavator digging out the ship”, we learned how to tackle a huge problem in the steps required, and unlike the famous image of the workers watching one person dig a hole, we learned to collaborate to get tasks done efficiently.

References

[1] Embedded Staff, "DSP Tricks: Frequency demodulation algorithms," 10 January 2011. [Online]. Available: https://www.embedded.com/dsp-tricks-frequency-demodulation-algorithms/. [Accessed 5 April 2024].

[2] K. C. Nicola Nicolici, "Project document," 12 February 2024. [Online]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/554053/viewContent/4569193/View. [Accessed 5 April 2024].

Project Activity



Week	Progress	Contribution		
		Member	Summary	Complexity
12-Feb	None			
19-Feb	None			
26-Feb	Converted Lab3 to C++ code. Non-functional Mono in Mode 0	Sam	Worked on getting blocks read in using standard input	Very Low complexity. Ran into segfaults, bugs, and compile time errors.
		Ivan	Pair program for mode and channel selection with CLI args	Very low complexity. Ran into minor bugs and compile time errors
		Hamza	Added functions created in labs 2 and 3 such as convolution, PSD and impulse response. Converted fmDemod. Created downsampling convolution with state saving. Added data writing normalization calculation	Low complexity - Seg faults in the downsampling computations - using unsigned integer type proved to be the cause and spotted via printing computations
		Yash	Made Obsidian notes to summarize project doc for Mono	Low Complexity
04-Mar	Debug mono mode 0	Sam	Mono Mode 0 Debugging with very little progress.	Low Complexity.
		Ivan	Pair program and pair debug Mono Mode 0	Low Complexity.
		Hamza	Created a script for running audio data due to driver issues on pi at the time. Pair program with group on mono 0 implementation. Adjusted state sizes and other bugs such as writing at wrong time.	High complexity - Pi sound drivers would not work for aplay hence had to make a whole script to do it ourselves based on bin data collected
		Yash	Added notes for the stereo. Checked mono FIRs and convolutions	Low complexity - couldn't figure out the main issue with Mono
11-Mar	Setup debugging infrastructure, begin working on adding functions for stereo while other members worked on Mono mode 0.	Sam	Setup Gtest for .dat file comparison between python model and cpp model. Add debugging functions in cpp and python.	Medium Complexity. Tried to find the best way to debug large sample sizes with the least amount of setup time possible during development.
		Ivan	Implement bandpass filter impulse response in cpp and python. Use Gtest fixture for verifying the BPF coefficients.	Low Complexity.
		Hamza	Fixed major bugs by monitoring vector indices and brackets in computations, added vector logging for verification	Medium complexity, lots of time - Using push back on resized vectors accidentally which went unnoticed until stepping through each vector to see that data was pushed very far in
		Yash	Debugged issues with state saving on FIR decim with Hamza.	Medium complexity - the issue didn't lie with the state saving and was upstream, but we scrapped it and redid a better implementation
18-Mar	Working Mono mode 0. Resampler debugging and completion, working single pass stereo audio in python using jupyter notebook. Working block processing stereo audio in python. Working block processing stereo audio in Cpp. Determine required block size and resampling values for each mode in the audio path.	Sam	Debugging Mono 0 with Gtests with continuous failing tests, Pair programmed with Yash blockifying stereo in python. Worked with Yash translating pll code from python to cpp. Worked with Ivan getting stereo audio working in cpp.	Medium Complexity. Spent too much time trying to get the gtests to pass as I thought it would be useful during stereo debugging. Took some time to understand the PLL code. A lot more moving parts in stereo which makes it difficult to pinpoint problems.
		Ivan	Implement the slow resampler with issues initially. Trace the source of the issue down to the impulse response generation. Debug the fast resampler made by Hamza and get it working with all modes. Setup config structure with for audio channel modes with block size.	Medium Complexity. Focused on having a good understanding of what upsampling/downsampling values are required and how they affect the required block size. Developed a good understanding of the workings of the resampler. Faced issues debugging various modes and checked for areas to optimize the existing functions.
		Hamza	implemented the resampler based on lecture notes. Turns out it was correct but we did not scale our frequency hence debugged this for no reason unfortunately. Added the initial argument parsing mainly for modes. Changed multiplication factor in writing to stdout which was incorrect. Redid decimation convolution and made IQ sampling faster - resulted in getting mono mode 0 to work. Did delay block with Sam and Mono path with group	High complexity - resampler took a lot of debugging to get it to produce at least sensible vectors, once producing results was passed to Ivan to help verify.
		Yash	Worked on testing stereo as Jupyter nb. Worked with sam on moving pll code to cpp with states. Tested the PLL multiple times	Medium complexity. PLL code in C++ was wrong because of the array sizing and I did not realize that resize was not working how I wanted. The PLL took a while to optimize with states because we did not know what states to keep and what to discard.
25-Mar	Entirety of RDS in python and cpp, completion of threading, fully working project.	Sam	Got Rds working in single pass. Figured out sampling algorithm (Ivan). Produced and adjusted constellation plots (Ivan). Algorithm for manchester decoding (Yash, Ivan) and added hh-ll recovery. Developed algorithm for bitstream recovery with bitstream selection scoring scheme (Ivan). Debugged differential decode function (Ivan, Yash). Produced algorithm for frame synchronization bitwise and blockwise (Ivan). Blockified all functions in python (Ivan). Figured out solution to half symbols per block (Ivan) Debugged Cpp translations of python functions (Ivan, Yash). Got Cpp implementation of RDS working on Mac (Ivan, Yash).	High complexity. There are now many considerations to account for. The blockifying of all of these functions is not trivial, especially when it comes to half symbols In a block. The pll was not locking early into the debugging of the cpp implementation which was worrysome. The python implementation significantly reduced the complexity of the task overall, especially the use of jupyter notebook as we are able to test a small section of code repeatedly. Very little instructions for the algorithms to be used is given in the RDS path which also increases complexity.
		Ivan	Pair programmed with Sam to complete the RDS python model in single pass. Collaborate with Sam to determine the required block sizes and resampling values for RDS based on our specific constaints. Collaborate with Sam on constellation plots, sampling algo, frame sync, and differential decoding for blockwise. Setup CPU affinity for the threading. Work with Hamza to debug the threading for RF and audio path. Debug the blockwise framesync cpp implementation. Implement RDS threading with addtional queue from RF front end.	High complexity. Ran into many seg faults, runtime errors and memory errors (vector memory) that were not immediately obvious. It was important to understand the whole RDS path and expected outputs at each stage while debugging. The python model proved to be extremely helpful going into the cpp implementation since it let us finalize our algorithms and understanding of the RDS path. In my opinion, although creating the cpp implementation had errors, we were able to resolve them much faster than we would have without python.
		Hamza	Did all threading implementation and debugged both the working threads portion and the RDS/Audio samples usage with queues. Implemented RRC and verified via gnuplot and python dat file comparisons. Finalized the mode configuration parameters map and fixed arg parsing. Got all threading to work. Sat with Ivan to get all modes working and got it set up in a modular coding style via map. implemented RDS model in python to be in cpp for the RDS support functions. Worked with Ivan on many mini bugs at the end such as writing to stdout and thread spawning issues and channel configurations. Ran tests with group outdoors to verify RDS	High complexity - verifying threads to receive and push data accurately - many issues with locks and conditions on the condition variable to wait as well as attempting two threads to receive same data in 1 thread via atomic variables was changed to two queues. Changed implementation for a clean thread safe queue class. Converting to CPP from model with various adjustments for data types such as needing bool types rather than integers for the arrays.
		Yash	Pair programmed with sam to do the rds model with blocks. Wrote the boilerplate and all the stuff up to the CDR for the cpp model. Worked with Hamza to verify and regenerate all the RDS helper funcs in Cpp. Worked with Ivan to fix a critical bug in the bitstream recovery and vector processing. Tested a variety of CDR algorithms and performance on RRC plots. Pair programmed with Ivan and Hamza to get the final implementation debugged by Sam.	High complexity. Working on a tight deadline and long hours meant errors were common. Issues such as push_back and insert in CPP causing memory overruns. We had a bad commit causing a lot of our work on the ending frame synchronization to be erased. Having a working Python model helped us make IQ constellations easily, allowing to debug the flow up to RRC and make plots of CDR points.
01-Apr	Report Generation	Sam	Setup the timing analysis in Cpp to generate time vectors to be consumed by python for further analysis. Generated data for varying number of taps. Obtained visualisation of CPU utilization with varying tap sizes. Described many of the implementations.	Med-High Complexity. There are interesting results in the timing analysis which take a large chunk of time to go through and reason.
		Ivan	Created jupyter notebooks for csv file generation of timed functions from Sam's cpp. Created timing charts and plots. Described implementations. Analyzed the timing against calculations.	Timing analysis is not obvious when comparing results to calculations, taking time to discuss and make conclusions from results.
		Hamza	Calculated the number of multiplications and accumulations for modes/paths in stereo and mono, the non linear operations done for each mode and path, wrote about threading, resampling, improvements on resampler, analysis of results, describing audio paths, writing my part in this table.	medium complexity - the calculations performed were originally done based on block but had to be adjusted per sample, so had to create conversion factors. Accounted for extra computations not necessarily needed, making computations more complex
		Yash	Wrote the intro and project overview and described implementations. Worked with Hamza to calculate and verify number of calculations per sample/bit. Wrote a quick Py script to calculate the number of calculations for functions.	Medium complexity. Minimizing the intro and overview while also preserving the details was challenging. The calculations took a while as there were many sampling frequencies to consider. We also had to make decisions on what functions to ignore in calculations as they would have negligible contributions but added complexity.

1

Table 9: CPU Assignment for Threads