3DY4 Software Defined Radio Project Report **Date: 04/05/24**

Group 03
Johnathan Gershkovich - 400408809
Kyler Witvoet - 400393313
Joshua Umansky - 400234265

Introduction

To consolidate knowledge of previous courses, this project assigned the task of creating a Software Defined Radio (SDR). With the aim of converting/decoding raw I/Q data into Audio Data and Radio Data System (RDS) signals in real time. We were required to navigate a complex technical specification for an SDR and to implement the specified SDR capable of decoding Mono Audio, Stereo Audio, and RDS data in real-time in a constrained environment (Raspberry Pi 4).

Project Overview

Utilizing the front-end radio frequency hardware given for the project, the Realtek RTL2832U chipset, and a Raspberry Pi 4, we implemented an SDR. The goal of the SDR was to receive frequency-modulated (FM) data in the form of I/Q samples, demodulate this data, and output the mono and stereo audio data contained in real-time, as well as the digital data contained in the signal transmitted using RDS protocol.

FM is a method of radio broadcasting utilizing frequency variation of the carrier wave according to the amplitude of the input signal. Software defined radios (SDRs) represent a significant advancement in radio technology, shifting many functions traditionally performed by hardware into software. This transformation allows for a flexible radio system capable of implementing different communication standards and modulation techniques simply by changing software parameters.

The basic building blocks of an FM receiver in an SDR context include finite impulse response (FIR) filters, FM demodulators, phase-locked loops (PLLs), and re-samplers. FM demodulators take the filtered signal and convert the frequency variations back into the original audio or data signals. FIR filters are utilized to isolate specific frequencies or bands of interest from the broad spectrum of incoming radio frequencies, effectively segregating different channels or sub-channels within a given broadcast. Re-samplers adjust the sample rate of the digital signal to match the processing needs of the system or the requirements of subsequent stages, such as digital decoding or audio playback. Phase-locked loops (PLLs) are used to maintain the receiver's tuning locked onto the frequency of the transmitter, using the signal from the pilot tone as a guide, ensuring stable and consistent reception.

Another critical component in FM broadcasting and reception is the Radio Data System (RDS), which allows for the transmission of additional data alongside the main program signal. RDS can convey various types of information, such as station identification, song metadata, traffic updates, and emergency alerts. In an SDR-based FM receiver, decoding the RDS signal involves extracting and demodulating the specific sub-channel dedicated to digital data, a task accomplished through precise filtering and digital signal processing.

By chaining these components together in a signal-flow graph, an FM receiver can be implemented in software, providing a flexible and powerful platform for radio communication. In the context of this project, there are three sub-channels within a given FM channel (200KHz band symmetric around the center frequency) that are of interest. The mono sub-channel (0 to 15KHz), the stereo sub-channel (23 to 53KHz), and the RDS sub-channel (54-60KHz). In addition to these sub-channels, two pilot tones (19KHz for stereo, and 114KHz for RDS) are present and must be extracted and remixed into the relevant signal.

<u>Implementation</u>

Labs:

Throughout the entire course, we were given tasks in the form of labs to introduce key concepts that would be needed to be used in the project going forward. Beginning with Lab 1, the concept of filters and software filter design was introduced with the task of implementing a low-pass filter in Python. We completed this using previously defined (Sci.Py) libraries, which we would then expand upon in a later lab to convert this implementation into C++ and without library support. Additionally, we were introduced to a band pass filter, extremely similar to the filter we would later implement in the stereo channel extraction portion of the project.

We began implementing our Python models quickly into C++, starting with Lab 2. In this lab, we took our working models from lab one, and implemented these functions in C++, specifically looking at creating the low pass filter coefficients function, as well as creating a convolution function. Our implementation of this filter function (impulseResponseLPF) and convolution would be carried forward through the course and into our project. While we followed the theoretical basis for creating our convolution function, we incorrectly implemented the state-saving aspect, which we would later correct before the project implementation.

During the final lab, Lab 3; which served as the base code for the project, we explored the power spectral density (PSD) of the FM channel, and visually plotted the frequencies that we would explore throughout the project. We would also create our FM demodulation function, which combined and demodulated the I and Q samples provided by the RF hardware. These I and Q samples had previously been passed through through a low-pass filter, and downsampler in order to have an intermediate frequency of 240 KSamples/sec. This entire process was summarized in the RFFrontEnd function, which would be expanded upon later in the project to implement upsampling and change the intermediate frequencies provided to the audio processing blocks of the SDR.

RF Front End:

At the start of the project, our approach was to use as many already-made C++ functions as possible from the labs that would be relevant to the project. We progressed through and implemented our low pass filter function, as well as an unoptimized version of block convolution with downsampling. While we would further improve the convolution function to implement resampling, this convolution function drastically reduced the computational complexity of convolution, as only calculations that would ultimately be present in the output would be computed. This is accomplished by changing the indexing to $x[n * Ds - k]$, where x is the input function and h is the filter). Our implementation of the RF front-end block in our SDR system extracts the FM channel for a block of I/Q data provided by the RF hardware. The I and Q components are interwoven together, and can be easily separated by extracting all elements with even indexes as the I data, and vice versa for Q samples, ultimately producing both an I and Q block. Through the use of fmDemodFormula, these I and Q samples can be used to calculate the FM demodulated data, at an intermediate sampling rate. This data is then fed into the Mono, Stereo, and RDS path. To accomplish demodulation, an approximation in the form of a formula is used, instead of the traditional arctan function, which is computationally expensive. The output of this function (fmDemodFormula) is the output of the entire RfFrontEnd function, which is the data that is then passed forward into the audio processing threads. As we moved further into the project and began implementing modes that required a different intermediate frequency, a new convolution function was written in order to implement resampling in an efficient manner. When we developed this new convolution function, we wanted to avoid doing calculations that would not be impactful to the output. In this process, as we are following the methodology of zero padding for resampling, a large portion of the calculations will result in zero, resulting in no effect on the output. Thus a method similar to the fast downsampler was used. This function iterates through the filter by U, starting at a phase of $D * n \% U$ (where D is the downsampling factor and U is the upsampling factor). And iterates through the input using a similar approach. Although this fast resampler function was created to handle upsampling, it is used throughout the project whenever convolution is required. When convolution is required but not up or downsampling, a value of 1 is used in the function call to replace the respective factor.

Mono:

The implementation of Mono audio mode 0 was implemented within the first week of the project, making use of the slower Conv_Ds function originally, and the low-pass filter. This implementation was quick as the Mono mode 0 was already written in Python during the third lab. We could troubleshoot the small amount of errors during this process by ensuring the size of all our vectors was correct, and we remained inside the defined range of our indexes. For this mode, we could test and ensure our code's accuracy by testing with the iq_samples.raw files provided during Lab 3. After ensuring this worked correctly, we transitioned to testing in real-time on the Raspberry Pi. When we were tasked with the remaining modes of Mono (1,2,3) these required upsampling at various portions of the code. This prompted us to write the fast resampler function, which was then implemented into both the RfFrontEnd, and the remainder of the Mono modes to meet the constraints given in the documentation.

For the implementation of Mono modes 2 and 3, an upsampling and downsampling value needed to be used to obtain the required output sampling rate of 44.1 KSamples/sec.

Mode 2	Mode 3

RF-to-IF: D = 10
IF-to-Audio - Requires Resampling
GCD (240k, 44.1k) = 300
U = 44100/300 = 147
D = 240000/300 = 800
	RF-to-IF: D = 5
	IF-to-Audio - Resampling Needed
	GCD (288k, 44.1k) = 300
	U = 44100 / 300 = 147
	D = 288000 / 300 = 960

Implementation of the remaining mono modes resulted in some challenges, the primary challenge being segmentation faults. These segmentation faults were difficult to debug given that almost no information is given with the error. To troubleshoot these, we used print statements in various points of the code, to figure out what was causing the segmentation fault.

Stereo:

To begin the Stereo path implementation, we had to implement a band-pass filter (BPF) and a phase locked loop (PLL). These functions were described to us in the form of pseudocode or Python and were required to be rewritten in C++, with some additional considerations such as state saving. The PLL also incorporated the numerically controlled oscillator (NCO) in the same function. With these functions implemented, we were able to implement the Stereo Carrier Recovery/Extraction. The pilot tone of the stereo channel, which is located at 19kHz, is extracted with the use of our band pass filter (impulseResponseBPF), and then passed into our PLL. When coding our PLL, we implemented state saving by holding the final values of; integrator. phaseEst, feedbackI, feedbackQ, ncoOut, and the trigOffset to operate on successive blocks. To validate our PLL, we graphed the output in MATLAB to verify graphically that the PLL was locking properly and our data at the output would be usable. As shown in the graph to the right, the phase of the PLL locks to the input of the wave. To perform the other portion before stereo data processing, the stereo carrier extraction is completed by using a band-pass filter set between 22KHz, and 54KHz, to extract the entire stereo channel. The delay introduced by the band-pass filters in both paths matches each other, to ensure that when mixing the paths in the stereo processing block the data correlates to each other. The final portion of stereo, Stereo Processing, involved mixing (pointwise multiplication) of the recovered carrier, with the stereo channels, which then would pass through a low-pass filter to perform sample rate conversion. At this time, the stereo channel (which represents the difference between left and right channels) is combined with a delayed processed mono audio block (delayed by the function delayBlock, which acts as a shift register) to then be outputted to aplay. In this path, the stereo data is added to the mono data to produce the left channel, and subtracted from the mono data to produce the right channel.

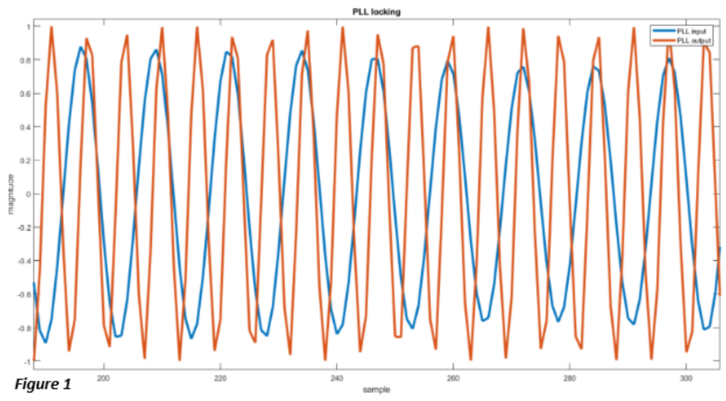


Figure 1

RDS:

The RDS path deals with extracting the RDS channel (54KHz to 60KHz), then downconverting and demodulating it to extract the digital bitstream. To implement this we used a similar approach to our stereo implementation, a band-pass filter is used to extract the entire RDS channel, which is then passed into the carrier recovery portion of the path, as well as passed into the 'delayBlock' function to match the delay of the tone extraction. For the pilot tone extraction, the data is squared nonlinearity (achieved by pointwise multiplication of each element with itself) to take only the positive portion of the data. This also provides a better resolution of the pilot tone inside of the channel. The pilot tone is isolated with a band-pass filter (113.5KHz to 114.5KHz), and passed through the PLL. Finally, in keeping with the stereo model, the pilot tone and the channel are recombined through pointwise multiplication.

The mixed data is then passed through a low-pass filter (Fc = 3KHz), this convolution also performs any resampling that is required. At this point, we implemented the root-raised cosine filter (RRC) that was initially provided in Python and converted to C++. The final step of the RDS demodulation is the clock and data recovery (CDR), which requires the development of a custom algorithm to implement. When observing the input data stream, the first portion of the initial block (block 0) results in unusable data. Therefore we chose to begin measuring from 8*SPS (Symbols per second constraint), at that point, we found the absolute maximum value within a span of one SPS and took that value as our offset (clock_offset). We would then sample the input data at this offset value, and iterate by SPS (Figure 3). As shown, this resulted in a good output of bit value (Figure 2).

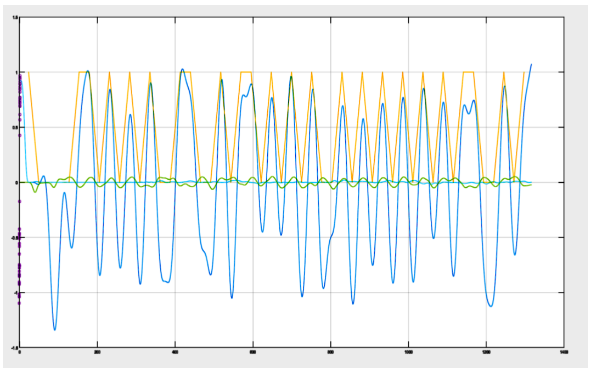
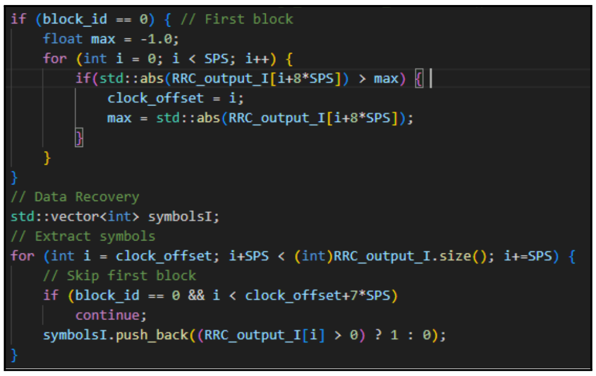


Figure 2 (Yellow output, Blue input)



```
if (block_id == 0) { // First block
    float max = -1.0;
    for (int i = 0; i < SPS; i++) {
        if(std::abs(RRC_output_I[i+8*SPS]) > max) {
            clock_offset = i;
            max = std::abs(RRC_output_I[i+8*SPS]);
        }
    }
}
// Data Recovery
std::vector<int> symbolsI;
// Extract symbols
for (int i = clock_offset; i+SPS < (int)RRC_output_I.size(); i+=SPS) {
    // Skip first block
    if (block_id == 0 && i < clock_offset+7*SPS)
        continue;
    symbolsI.push_back((RRC_output_I[i] > 0) ? 1 : 0);
}
```

Figure 3

The final portion of RDS is data processing, which begins with Manchester decoding. We implemented Manchester decoding by pairing consecutive bits together, and then deciding if the bits transitioned between a high-to-low (resultant 1), or a low-to-high (resultant 0) value respectively. Errors can occur if we have two bits of the same value paired together, this is a result of our data being out of phase with the real data stream. To correct this, we employed a strategy of shifting the index to bypass one of the paired bits, resulting in us being back in phase with the data stream, and repairing the bits going forward. To prevent the inversion of waves used for mixing from affecting the binary data, differential coding was applied at the transmitter, resulting in differential decoding being done at the receiver. To accomplish this, consecutive bits of the Manchester decoding output are XORed together to produce the RDS bitstream, which operates at 1187.5 bits/sec.

The final component of the RDS system is Frame Synchronization, which ensures that the received data frames are properly aligned with the transmitter's frame structure. Frame synchronization is accomplished by multiplying a sliding window of the most recent 26 bits received from the Manchester decoder with the parity matrix specified in the appendix of the project document [1]. After multiplication with the parity check matrix, which is a 26 x 10 matrix, the resulting matrix, which is 1 x 10, is compared to the syndrome words, also specified in the appendix of the project document. If there is a match then it can be determined that our system has detected that letter.

Our current implementation of RDS does not correctly identify letters, we are only able to identify the letter "A". We believe that there is likely an error in our CDR or Frame Synchronization components but didn't have enough time to fully debug.

Analysis and Measurements

To evaluate the timings of the different processing paths, and understand how alternating modes and filter quality affect the speed of each path, an analysis of the multiplications/accumulations and non-linear operations per sample or bit was completed. Multiple equations were determined for the different paths to represent the speed of a path based on the number of operations required for an output, such as a sample or a bit. These are the determined equations:

$$r_N = \frac{\frac{U_r}{D_r} \cdot b_d(5)}{\frac{b_d U_r}{2 \cdot D_r \cdot S_{PS}}}$$

$$s_N = \frac{b_d(4)}{b_d \cdot \frac{U}{D}}$$

$$m = \frac{\left(\left(b_d \cdot \frac{U}{D} \cdot \left(3 + \frac{n_t \cdot U}{U}(2)\right)\right) + b_d \cdot \frac{U}{D}\right)}{b_d \cdot \frac{U}{D}}$$

$$r_M = \frac{\left(2\left(b_d\left(3 + n_t(2)\right)\right) + 14 b_d\right) + \left(2 b_d \cdot \frac{U_r}{D_r}\left(3 + n_t(2)\right) + 2 b_d \cdot \frac{U_r}{D_r S_{PS}}\right)}{b_d \cdot \frac{U_r}{2 D_r S_{PS}}}$$

$$s_M = \frac{2\left(b_d\left(3 + n_t(2)\right)\right) + 10 b_d + \left(b_d \cdot \frac{U}{D}\left(3 + n_t(2)\right)\right) + b_d \cdot \frac{U}{D} + b_d + \left(\left(b_d \cdot \frac{U}{D} \cdot \left(3 + \frac{n_t \cdot U}{U}(2)\right)\right) + b_d \cdot \frac{U}{D}\right) + 2\left(b_d \cdot \frac{U}{D}\right)}{b_d \cdot \frac{U}{D}}$$

Simplifying the above equations yields the following table (MA represents multiplications/accumulations & NL represents non-linear operations):

	Multiplications/Accumulations	Non Linear
Mono	$2(numtaps + 2)$ MA/sample	
Stereo	$(\frac{D}{U}(17 + 4 \cdot numtaps) + 4 \cdot numtaps + 10)$ MA/sample	$(4\frac{D}{U})$ NL/sample
RDS	$(8SPS\frac{D_{RDS}}{U_{RDS}}(5 + numtaps) + 4SPS(3 + 2 \cdot numtaps) + 4)$ MA/bit	$(10 \cdot SPS)$ NL/bit

Where D is the downsample ratio between the intermediate frequency (IF_Fs) and the GCD of the IF_Fs and the output frequency (Audio_Fs), U is the upsample ratio between the GCD of IF_Fs & Audio_Fs and Audio_Fs. D_r is the downsample ratio between IF_Fs and the GCD of IF_Fs & 2375*SPS and U_r is the upsample ratio between the GCD of IF_FS & 2375*SPS and 2375*SPS.

	Mode 0	Mode 1	Mode 2	Mode 3
Mono	206 MA/sample	206 MA/sample	206 MA/sample	206 MA/sample
Stereo	2,315 MA/sample 20 NL/sample	1,893 MA/sample 16 NL/sample	2,501 MA/sample 22 NL/sample	2,958 MA/sample 26 NL/sample
RDS	107,017 MA/bit 260 NL/bit		122,597 MA/bit 450 NL/bit	

By timing the individual processing paths in the C++ code using std::chrono, the total time for the mono, stereo and RDS paths to process one block was determined. These timings were then normalized by dividing by the output block size in order to obtain the time to process one sample for mono and stereo or one bit for RDS. The table below shows the normalized time values.

	Mode 0	Mode 1	Mode 2	Mode 3
Mono	762.9883 ns/sample	763.0859 ns/sample	949.1156 ns/sample	984.76 ns/sample
Stereo	9.7272 us/sample	8.2186 us/sample	11.0971 us/sample	12.6811 us/sample
RDS	0.5997 ms/bit		1.1829 ms/bit	

The timings generally follow the expected performance of the individual paths based on the determined operations per sample/bit, although some discrepancies are noticeable. First of all, the mono path should generally only correlate with the number of taps, disregarding smaller detailed operations. In the timings above modes 0 and 1 have a time per sample that is more than 10% smaller than that of modes 2 and 3. One possible explanation for this is that one of the multiplication operations in the inner loop in convolution depends on the upsample factor. When this factor is equal to one as it is in modes 0 and 1 the multiplication operation would likely not be computed for nearly the same amount of time as a value greater than 1, the compiler likely fully cuts out this multiplication as no computation is required, thus modes 0 and 1 could take less time due to this.

All of the above timings were done using 101 taps. Based on the equations derived above, it can be observed that all of the multiplication/accumulation counts correlate with the number of taps, increasing or decreasing the number of taps used should linearly increase or decrease the performance time. Below are the timings using 11 taps.

	Mode 0	Mode 1	Mode 2	Mode 3
Mono	118.6523 ns/sample	92.6758 ns/sample	143.2653 ns/sample	156.9728 ns/sample
Stereo	2.3519 us/sample	1.7771 us/sample	2.5169 us/sample	2.9020 us/sample
RDS	48.8250us/bit		89.3749 us/bit	

Not all of the values seem to scale linearly as we would have expected closer to a 10x decrease in time. This is likely due to the small tap size, as the constant time values of operations not in loops (which were not included in the calculations) are more visible in the total when there are less loop iterations. A lower tap count leads to less loops overall. Accordingly, we can easily see the linearity between the tap count and processing time when we scale up to 301 taps and run mono in mode 0. The timing with this number of taps is 2.1537 μs/sample, which is very close to three times as large as the speed at 101 taps, showing the linearity between the number of taps and processing speed.

<u>Proposal for Improvement</u>

To enhance the user experience of our SDR and boost the efficiency of future development, we propose two significant additions.

Firstly, to enhance our development and debugging process we propose the addition of an automated logging function. This function would simplify data visualization by automatically accumulating signal data of a specified variable during program execution. Removing the need to manually add logging code each time a signal needs to be visualized, this function would also allow for easy visualization of state transitions due to accumulating the data during the program's entire runtime. The collected data could then be easily analyzed using gnuplot or exported to MATLAB or Python for further analysis, significantly speeding up the debugging process.

Pseudocode for the logging function:

```
void logging(data, id):
        - Open .dat file named after the id
        - Append all elements in data variable to the .dat file
        - Close the file
```

To enhance the user experience of the SDR we suggest the implementation of a Python-based user interface (UI). Providing a more accessible and error-resistant environment for running the SDR's various modes, by removing the need of using the command line to run the program. This UI would be implemented in Python using the "Tkinter" library to build the UI and the "subprocess" library to execute the linux commands necessary to run the desired mode and path of the SDR.

Regarding the optimization of runtime performance for hardware with limited computing resources, we identified two technical improvements that would provide the greatest performance boost.

The first being replacing the convolution function with a fast fourier transform (FFT) approach.

Pseudocode for the FFT convolve function:

```
void FFTconvolve(input , filter, state):
    - X = FFT(input, state)
    - H = FFT(filter)
    - Y = X * H
    - output = IFFT(Y)
    - return output
```

This replacement would dramatically improve efficiency by reducing the runtime of the current fastResampling function from O(n^2) time complexity to O(n * log n). Additionally, there is already a basis for this function provided in the fourier.cpp file in the “FFT_optimized” function, meaning that only an implementation of an IFFT function is required. To incorporate resampling the input would be zero-stuffed before being passed into the FFT, and the output would be downsampled after being passed through the IFFT. There is likely a way to improve this further, similar to how the current convolution function incorporates the upsampling and downsampling factors to make resampling fast.

The second method for improving the runtime complexity would be to replace the use of “std::atan2” in the RDS I and Q separation with a method similar to formula approximation of the “fmDemodFormula” function which is used in the RF front end to combine I and Q samples.

Project Activity

Week	Johnny	Josh	Kyler
Feb 15 - 17	Review Project Documentation	Review Project Documentation	Review Project Documentation
Feb 18 - 24	Mono M0, debugging no audio feedback	Fast downsampling + convolution, debugging segmentation faults	Mono M0, Fast downsampling & convolution, debugging
Mar 17 - 23	Mono M1,M2,M3	Debugged segmentation faults & state saving for Mono M2	Mono M1,M2,M3, Fast resampling & convolution, debugging
Mar 24 - 28	Wrote and debugged stereo (audio not splitting, PLL problems), implemented RDS extraction, recovery demodulation, as well as CDR. Debugged fmDemod	Wrote and debugged stereo (seg. fault, audio bleed), designed CDR algorithm with Johnny, implemented CDR/Manchester/Differential decoding	Stereo M0, M1, M2, M3 RDS M0, M2. Implemented multi-threading for all paths, implemented Manchester/Differential Decoding and Frame Sync.

Mar 29 - Apr 5	Project Documentation	Project Documentation	Project Documentation

Individual Challenges

Johnnathan:

The start of the project went smoothly implementing the mono path for the first mode using the down sample based convolution, although we did not achieve audio output for the first day as there was a casting error. I found this casting error by printing blocks of the mono output data and noticing that the data seemed like overflow data from the float type being used before casting. Switching the casting type to static_cast fixed this and produced audio output instead of simply noise. Moving on from this I implemented the PLL and band pass filters which had been given in python, naturally, a straightforward process. I later implemented these two functions with the front end process created by the other two group members and coded the mixing with the delay and interweaving of the channels. The stereo path was riddled with problems from the beginning of testing, we would not even get a mono audio out originally, after debugging and finding misused variables the audio was audible only as a mono audio, no splitting was occurring between the channels. In case I had somehow mis implemented the given bandpass filter and PLL functions I debugged both. By outputting the frequency domain of the bandpass input and output I confirmed by graphing that it successfully stopped frequencies outside of the band. Matlab was preferred over gnuplot to plot graphs as it allowed easy inspection of the data in the graph as well as concatenation of other data, like multiple blocks together, making my workflow much faster. After plotting the PLL/NCO output with the input signal in the time domain, the PLL appeared to lock on, although when I plotted a block that was not the first, the start of the block seemed to be out of sync. After confirming the code of the PLL was correct I noticed the state variables were not properly passed as pointers, meaning the PLL was not saving state between blocks, a common issue I ran into when coding the stereo and RDS paths. Coding slower, more organized and naming variables correctly would have saved a lot of time debugging. After finding that other states were not correctly passed as pointers the stereo audio was still playing back as if it were mono. After a full day of debugging Josh noticed we had not even used our delayed data for the mixer, rather the original data.

Implementing RDS extraction, recovery, mixing up to the CDR was fairly straightforward as it required usage of functions i had previously coded like the bandpass and PLL. I also implemented the RRC filter, originally implementing it incorrectly, but after re-coding it a couple of times I confirmed it was correct. Despite confirmation that the functions being used previously worked for the stereo path, there seemed to be some issues when using them in the RDS path. The primary issue was that at the start of each block, a short period of samples would be very large compared to the rest of the data, which would then settle down. Although the data could possibly still be normally recovered in the CDR, it seemed to indicate one of the functions was not implemented correctly, likely misusing a state. I outputted data at each point in the RDS pre CDR, in order to graph the frequency domain data before and after each filter, to ensure the correct frequencies were passing through, as well as the time domain data before and after the pll, before and after mixing and before and after the RRC filter. There seemed to be a very noticeable spike in most of the data, including the delayed extracted channel data, the filter data and the outputs. I assumed that the problem likely lay with the shared function which led to all of them, fmDemod. I inserted print statements at the beginning and end of fmDemod to print out the state as well as the output data. There was a noticeably large value in the last value of the output. After inspecting the code I found there was an indexing error, where one of the arrays used to calculate the output was indexed one past its last index for the last value of the output. This had gone unnoticed as one off sample is not audible when listening to the audio, and the compiler did not throw an error as C++ indexes by taking a memory offset from an initial position. If there is memory in the stack to read at that offset then that memory is used, even if the index is beyond the last index of an array. This meant some other variable initialized in the stack was being used to calculate the last value in the fmDemod output, creating a very large spike in one sample. After correcting this, I implemented CDR, a fairly straightforward process, using a simple algorithm to find the highest absolute point in one symbol range and assuming that as the symbol offset from the start of the block. It was important to skip the first 7 symbols as the PLL had not locked for those symbols and they would be bad for CDR. The data appeared to be in the correct format, although after plotting multiple blocks, it was evident that the amplitude fell to a very low value when a new block started, sometimes causing data to be unreadable, after inspecting all of the states and whether they were used properly, as well as ensuring no memory leaks were present with valgrind, or indexing errors, a solution was not found.

Josh:

We began this project at a very good pace, starting with the Mono path, mode 0 during our scheduled lab time we worked together as a group to complete a not fully optimized version. This original version was able to read data from a .raw file, and output a .bin file which could be played for audio. We were able to implement the fast downsampling convolution to this mode, and had this working before leaving the lab that first week. The final two weeks of the project saw us as a group be extremely productive, I myself worked to solve the issue of segmentation faults occurring when working with Mono mode 2. These faults were specifically occurring after our convolution (fastresampler) as the vectors were being sized incorrectly at various parts of the function. To locate this bug, we used print statements to pinpoint exactly where the fault was happening as the error message was very unclear. Then we would print the size of all relevant vectors at the point directly before any function, in this case fastresampler, and work out by hand the expected values after the function. Typically this would result in a noticeable difference with one of the vectors, and correcting the resizing would eliminate this segmentation fault. Secondly, when debugging mono we were consistently having issues where the audio would cut out for a small portion of time, which we determined was a fault of the state saving being incorrect. When I went through the fast resampler, we were not saving enough of the previous data, as the sizing was incorrect and in reality we were saving only a fraction of what was needed to complete the next computation. This was changed to save exactly the size of the filter used in the convolution, and the state was passed by reference in the function call, instead of being passed directly.

During the final week of the project, I implemented stereo alongside Johnnathan, directly into C++, and attempted to have this code operate directly on the Raspberry Pi. While the code ran on the Raspberry Pi, we noticed that the audio data was not split into a left and right channel, and the same audio was played in both ears. We went through and looked at every variable, and noticed that the delayedMono audio was not being processed by our code at all, and instead we were using data that was out of phase with our stereo process. This was because of improperly named variables, leading us to be confused on which data was used. After fixing this, the audio did split properly between ears, and the remainder of the stereo modes could be implemented rather quickly by changing constants (by the constraints given) in the function call. The small amount of bleed remaining was solved later by Kyler, as we had an incorrect operation sign involved in the mixing of mono and stereo data.

My final task was to implement the final portion of RDS, alongside Johnnathan we came up with the CDR algorithm described in the Implementation section together, and I wrote the algorithm into our project, which we then tested together using MATLAB plots. After this was complete, I implemented the Manchester decoding, and differential decoding, although in hindsight (with the help of Dr. Nicolici during the lab interview) we improperly implemented Manchester decoding. We originally assumed that after CDR we assign binary values to the samples found through our CDR algorithm, but we should have held this value as a pure number until Manchester decoding. This would have allowed us to interpret the data better, as a sense of a low to high (or high to low) transition could be found even between two positive (or negative) numbers that had a large difference.

Kyler:

In the first weeks of the project I worked on implementing the Mono path with Josh and Johnny, we mainly only worked during our designated lab times on Tuesdays, and we finished the slow implementation of Mono mode 0 together. Towards the end of the project we began to pick up our development pace, after getting Mono mode 0 and 1 working in real-time together we decided to begin splitting up the work. I focused on wrapping up the rest of the modes for the Mono path and implemented upsampling, followed by a fast resampler. One of the major issues I was running into when implementing the fast resampler was segmentation faults due to indexing issues, to debug these segfaults I would place simple print statements throughout the code, which made it easier to figure out what was causing the issue. For example, printing the line number after each major operation in the code, or the index inside of a loop to see where the code breaks.

While Johnny and Josh worked on implementing the Stereo path, I began optimizing and modularizing functions in Mono to improve the efficiency for later modes and paths. To do this I ensured that all function calls that passed large vectors utilized pass-by-reference, I moved all one-time operations such as filter creation outside of the main loops, and reduced the number of multiplications and divisions as much as possible. I then began to implement multithreading, by developing a thread-safe queue class and separating the RF Front End and Audio Processing into individual threads that can run in parallel. Even though the Mono path worked in real-time, I could tell there was a bug because some of the modes sounded a bit muffled. To solve this I tracked/followed each signal through the program and ensured that all of the signals were flowing properly and the calculations were correct according to the project specification and lecture slides. I eventually found that I wasn't calculating the IF sampling frequency correctly, and was instead passing the RF frequency, which was messing up some of our filters as it resulted in a lower cutoff frequency, which removed some of the high-frequency components in the audio data. Fixing this made the audio sound clear.

After finishing the Mono path I converted Stereo to multithreading and to run in real time. After completing this there was still some audible bleed between the left and right audio channels. To solve this I followed a similar procedure to how I debugged the muffled audio in Mono. In the end, I found this bug was located in the final combination of the mono data and stereo data, where the stereo data was divided by 2 instead of multiplied by 2.

Once I finished with Stereo I began working on RDS, at this point, Johnny, Josh, and I were working cohesively and were figuring out the conceptual side of RDS together. While they were working on implementing the early stages of RDS like the Channel Extraction and Carrier Recovery, I worked on implementing the later stage of Data Processing, starting from Manchester Decoding. To do this I began researching Manchester Decoding, Differential Decoding, and Frame Synchronization to confirm my understanding. I implemented these functions first in a separate C++ file and tested them on recorded data from the CDR. However, we were never able to confirm that our output from the CDR was correct and since I was using this recorded data I was never able to confirm if my implementation of the decoding and frame sync was correct.

Regardless of the output not being correct, our RDS system ran without any errors and I began to implement multithreading for it while Johnny and Josh continued debugging. Integrating multithreading for RDS was more complicated than for Stereo or Mono because the RDS thread had to run in parallel with the Audio Processing thread, and both needed to use the same block of FM demodulated data produced from the RF front-end thread. To solve this I considered multiple options such as modifying the queue class to handle multiple consumers, but decided to simply make a second queue for RDS as this was the simplest option and required the least amount of debugging. Using a second queue also allowed me to make the SDR RDS path more efficient by rebalancing the computations required by each thread so that the RF Front End function also handled some of the RDS processing. This final ‘frontEnd_RDS’ function would first demodulate the I/Q data, pushing this block to the Audio queue, then run the RDS Carrier Recovery processes and push this data to the RDS queue. Then the Audio and RDS threads could continue processing their respective blocks.

Conclusion

As a group, we were able to correctly implement both the mono and stereo pathways in their entirety, and made good progress towards RDS being functional. Overall we were able to apply previous course knowledge, and new concepts along the process of design and development of a software defined radio. Through the project, we gained a better understanding of how to adhere and implement industry standards into a system we were building from scratch, as well as built the foundational knowledge on how to optimize digital systems to meet hardware restraints. We will be using all of these skills next year during co-op and in our future careers as engineers.

References

[1] N. Nicolici and K. Cheshmi, “COE3DY4 Project Real-time SDR for mono/stereo FM and RDS”, COMPENG 3DY4, Dept. of Elect. and Comp. Eng., McMaster University, Hamilton, ON, Canada, 2024. [Online]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/554053/viewContent/4569193/View

[2] N. Nicolici and K. Cheshmi, “FM Modulation and Receiver,” COMPENG 3DY4, Dept. of Elect. and Comp. Eng., McMaster University, Hamilton, ON, Canada, 2024. [Online].

[3] N. Nicolici and K. Cheshmi, “(...)Sampling,” COMPENG 3DY4, Dept. of Elect. and Comp. Eng., McMaster University, Hamilton, ON, Canada, 2024. [Online].

[4] N. Nicolici and K. Cheshmi, “Project Overview,” COMPENG 3DY4, Dept. of Elect. and Comp. Eng., McMaster University, Hamilton, ON, Canada, 2024. [Online].

[5] N. Nicolici and K. Cheshmi, “Stereo,” COMPENG 3DY4, Dept. of Elect. and Comp. Eng., McMaster University, Hamilton, ON, Canada, 2024. [Online].

[6] N. Nicolici and K. Cheshmi, “PLL,” COMPENG 3DY4, Dept. of Elect. and Comp. Eng., McMaster University, Hamilton, ON, Canada, 2024. [Online].

[7] N. Nicolici and K. Cheshmi, “Multi-Threading,” COMPENG 3DY4, Dept. of Elect. and Comp. Eng., McMaster University, Hamilton, ON, Canada, 2024. [Online].

[8] N. Nicolici and K. Cheshmi, “RDS,” COMPENG 3DY4, Dept. of Elect. and Comp. Eng., McMaster University, Hamilton, ON, Canada, 2024. [Online].

[9] N. Nicolici and K. Cheshmi, “Project Report and Presentation Guideline,” COMPENG 3DY4, Dept. of Elect. and Comp. Eng., McMaster University, Hamilton, ON, Canada, 2024. [Online].