GROUP 04
RYAN MANCERA (mancerar@mcmaster.ca)
JAPMAN NAGRA (nagraj4@mcmaster.ca)
JEREMY CHERIAN (cherij2@mcmaster.ca)
SHAIYAAN ISHTIAQ (ishtis1@mcmaster.ca)
APRIL 5, 2024

Introduction

Over the given 3 months, our group has been tasked with creating a Software Defined Radio Receiver that can work in Mono, Stereo, and RDS while also being able to change into different 'modes', with each of these 4 modes having different sampling rates at different points in the receiver process. We have this SDR written in C++ on a portable but powerful Raspberry Pi 4, attached with a Realtek RTL2832U USB Dongle, a radio antenna, and headphones. Throughout this report, we will go into further detail on the steps of each of these processes, run times for sections of our code, and difficulties that we faced as a group and individually.

Project Overview

Our project first begins with receiving the 100kHz FM signal and going through front-end processing. This involves using FIR and LPF filters to get specific frequencies as well as downsampling to get an intermediate sampling rate which is specific to the mode we are in. This is done for both the I and Q samples coming in. We then combine(or demodulate) the I and Q samples by using a derivative demodulator function, which calculates the instantaneous change in phase of the RF signal.

```
current_phase[i] = (I[i] == 0.0 || Q[i] == 0.0) ? 0.0:(I[i] * deriv_Q - Q[i] * deriv_I) / denominator;
```

After the front-end processing, we go into the Mono, Stereo, or RDS path. Starting with the Mono path, we convolve an LPF filter with the incoming front-end data to receive the final audio information for Mono. While convolving we also have a resampler running so that we can change the intermediate frequency to meet the final sampling rate specification for each mode.

Stereo has a few more steps; It involves processing the noisy pilot tone(19kHz) using a BPF to get the pilot tone and running it through a Phase Lock Loop, which simulates a signal similar to the input and ensures that this simulated signal is at the correct phase. At the same time, we extract the bandpass (the two bumps around 38kHz) of the stereo frequency. We then mix the pilot and bandpass so that we can get the signal at baseband, resample it to meet our specification for the modes, and then run it through a stereo combiner to get the right and left stereo audio.

For RDS we have similar steps to Stereo, however, we aren't given a pilot tone so we need to create our own. That's why after doing the channel extraction at 57kHz, we have to square nonlinearly to create a duplicate of the RDS band at 114kHz, which acts as our pseudo-pilot tone. We then run a BPF around 114kHz to extract this pilot tone and run the output of the BPF through a PLL+NCO to get the correct phase and to downconvert the frequency to 57kHz; We then mix this pilot tone with our RDS band, the output of our initial channel extraction around 57kHz to center the RDS band around the Baseband, or 0kHz. We then extract the positive or right side of this RDS band, which is now centered at the baseband, with an LPF of 3kHz. This output is run through a rational resampler to ensure we have 2375 symbols/sec and an SPS value matching our specifications(Mode 0: 30 SPS, Mode 2: 43 SPS). We then have a Root Raised Cosine Filter to smoothen out the transitions between HIGHs and LOWs on the sinusoidal wave and reduce Intersymbol Interference (ISI), making it easier for the Clock and Data Recovery Block to detect the difference between possible bit values. Our Group partially completed the CDR process, with the detection of HIGHs and LOWs in an input signal being mostly correct. If we were

to continue we would have completed the CDR by pairing HI and LO values using Manchester decoding, and this paired output is put into Frame Synchronization, which checks the bitstream against the 5 syndromes to sync the bitstream to a frame. The last step is the application layer which outputs important information about the audio signal such as PI code, program type, etc.

Settings for group 4	Mode 0	Mode 1	Mode 2	Mode 3
RF Fs (KSamples/sec)	2400	960	2400	960
IF Fs (KSamples/sec)	240	320	240	120
Audio Fs (KSamples/sec)	48	40	44.1	44.1

Implementation Details

Lab 1 is a general introduction to Python giving us an idea of how to implement three things in Python fourier transform algorithms given pseudocode, design of digital filters using given libraries and our implementations, and finally processing audio signals in blocks, which focuses on working with data sets that can be broken down into blocks to manage resources. This allows us to understand the use of the discrete fourier transform and the inverse discrete fourier transform. It also allows us to use well-documented filter design libraries such as SciPy and NumPy to showcase how these libraries can simplify DSP tasks and also to compare our implementations of their functions. Functions implemented in the block processing section allow us to see two common practices in DSP, processing entire signals in one pass, which is done without state saving, and there is breaking the signal down into chunks to manage resource constraints. Lab 1 gives us a good introduction to block processing and filter design in Python which is easier to manipulate in comparison to C++.

Lab 2 takes the Python code from lab 1 and requires you to refactor this code into C++, but since there are no given libraries like NumPy or SciPy, you would refactor the functions that you developed yourself in Python into C++. Again there are discrete fourier transform and inverse discrete Fourier transform implementations, this time in C++. Python allows us to plot graphs using matplotlib, but in C++ GNUplot is used to visualize your filter design. The impulse response LPF developed in this lab is used again in the project. Finally, there is the block processing section which implements a single pass convolution function and there is the block processing portion of this file which allows us to break down large amounts of data by breaking it down into blocks and using state saving to process these blocks. These files provide a great backbone for the functions that are developed in the project because a lot of the functions from this lab are reused in the project. The switch from Python to C++ shows a shift to efficiency and real-time processing, which is one of the main real-world constraints given to us in this project. This lab allows students to validate DSP algorithms which can be carried over to the project.

Lab 3 switches back over to Python modeling for understanding the flow of processing signals to produce mono audio. There are 4 files in this lab fmSupportLib which are the support functions that are essential for signal processing. fmMonoBasic is for processing the audio signal with a single-pass convolution. fmMonoBlock focuses on block processing which again handles large data by breaking it down into blocks, allowing us to have state saving. Finally, fmMonoAnim visualizes the FM signal with animation. This lab takes us through the signal flow graph of generating our impulse response, doing the convolution, and then downsampling the output to get the sampling frequency that we want.

RF Front-End Processing

The processing done on the RF front is the most crucial step to get the project up and running. Samples that are read from the standard in contain the in-phase, the I samples, and the quadrature, the Q samples. It is a multistep process that involves reading the I and Q samples from the standard input, splitting the I and Q samples, and applying convolution on each I and Q, while simultaneously downsampling to isolate the FM signal and then they are passed into the demodulation function. In terms of actual implementation, we are provided with readStdinBlockData which reads raw data that is then converted into a vector of floating

point samples that have been normalized between -1.0 and 1.0. This is then passed into our function which splits this vector into I and Q vectors that are split based on whether the index is odd or even. After they are split both I and Q are convolved with an impulse response of 101 taps with a cutoff of 100 kHz and a decimating factor based on the constraints given. The general constraint for modes 0 and 2 are front-end decimating factors of 10. Given our constraints for modes 1 and 3, our front-end decimating factors are 3 and 8 respectively. The very first implementation of this convolution included convolution with the whole signal and then performing downsampling, but this implementation is still too slow to be run in real-time. The steps to develop the downsampler included the convolution, calculating the output vector size and then doing index manipulation to only convolve certain elements of the x vector with the impulse response. This downsampled vector is then passed into the demodulation process. This process iterates over the I and Q samples, to compute the demodulated signal, we are calculating the change in phase over the change in time. This is done by calculating the rate of change for both I and Q. Then cross multiplying the rate of change of Q with I and the rate of change of I with Q This is all divided by the magnitude of the current vector, which normalizes the value. These functions run the same way they did in lab 2 with a void function where the output is passed by reference into the function and it is updated from there. This demodulated data is the output and is what is going to be passed to all the different paths of this project including mono, stereo and rds. In the C++ version of this implementation, the vectors through each function call were printed and compared to the Python model, and when they close within an error of 0.001 then the next step could be implemented. Other debugging steps that could be done at this point would be plotting this in both C++ using GNUplot and comparing the plot with the Python model created in lab 3 using matplotlib. Another debugging way that could be utilized would be to perform the fast Fourier transform on the output signal for both Python and C++ to analyze the frequency components. Utilizing the Python model created in lab 3 allowed this implementation in C++ to be easier to understand and debug.

Mono

Implementing the mono path in C++ is very similar to the RF front end in terms of refactoring and in terms of the way we debugged the code. We took a step-by-step analysis. The very first implementation of the mono path included generating the impulse response for the mono cut-off of 16 kHz, then convolving the entire input signal with the impulse response coefficients, and then downsampling, like what was done above in the rf front end. The way this slow downsampler was debugged was by using the cat command to pipe into our program executable and then redirecting the writing to a file by redirecting the contents to a file, and then using the cat command in Linux and piping that into aplay. At first, the audio did not sound correct because the way we were writing the audio data was incorrect. We were making the final audio_data vector the size of the block when in reality it should be a much smaller size because of the downsampling that has to be done in mode 0. After the size was fixed accordingly the redirected contents played the correct audio. Then the faster downsampler was implemented, by resizing the y vector to be the size of the input vector divided by the downsampling factor, and then only adding the partial product to the y vector if the modulo of the current y element = 0. This would do the downsampling and convolution in one function. Again this function is not fast enough to run in real time because the number of multiplications done in the functions is the whole input signal with the impulse response, we only want the input signal at every downsampling factor to be multiplied with the impulse response. The fastest downsampler that was implemented resizes the output vector still, but when doing the multiplication for the partial products that need to be done, only every decim factor that we want is convolved with the impulse response, meaning the computational runtime would be much lower in comparison to the second implementation. The way the fast downsampler was debugged was by comparing the output vector with the contents of the output vector in the Python implementation. After successfully getting the correct audio, and being able to process data in real time for Mode 0, the project.cpp file was updated with the correct values for mode 1 and the downsampler function worked for mode 1 without any issues. The biggest challenge in mono was the development of the resampling function. There was no development of

a slow resampler, where there was an upsampler, then convolution, and then downsampling, as the lecture notes given to us were very knowledgeable and useful in the development of the resampling function. The gain created for the impulse response LPF is created to upscale the impulse response coefficients which has to be done when increasing the size of the impulse response coefficients, the gain has to be adjusted to decrease the distortion, which could happen from a poor signal-to-noise ratio. The conv_rs function included in our code performs convolution whilst combining this with resampling. In the conv_rs function the output vector is resized to be the input signal multiplied by the ratio between the upsampling and downsampling rates, changing the sampling rate of the output signal. Each value of the output vector sums up the partial products of the h vector starting at a phase which is calculated using the equation (n*ds)%us, where ds is the downsampling factor, us is the upsampling factor and n is the output vector index. The step size is the upsampling factor which accounts for the resampling process by selecting and applying filter coefficients to match the sample rate. There is also a corresponding input sample index that is calculated which is dx, which is calculated using this equation dx = (ds*n-k)/us, this ensures that each output sample has the correct input samples that are needed according to the resampling rate. If the input sample index goes beyond the input vector the function uses the state vector instead of the input vector. State saving is done by taking the last 101 values of the input vector, and saving it for the next block; initially, the vector is filled with zeros.

Stereo

Implementing the stereo path is very similar to the mono path because a lot of the functions are reused at some point in the stereo path, this being the conv_ds_fast, conv_rs and the LPF impulse response function. New functions that had to be implemented included the Bandpass coefficients, the PLL function, and the delay block. The bandpass impulse response pseudo-code was given to us and Python versions of the PLL and delay block were given to us. The initial model was created in Python for mode to utilize the NumPy library. After changing the PLL function to include state saving, on the first try at running the code, the left and right channels were not splitting correctly, this was tested by writing both left and right channels to their separate .wav files. After testing this first try, I realized that the size of the state for the delay block was incorrectly set to num_taps meaning that the space delay for the all-pass filter would not be correct. After changing this subtle detail, the left and right channels were separated correctly. To interleave in Python we used NumPy hstack which interleaves two arrays into a new array. Moving on to C++ the implementation into C++ was a lot easier to debug. After implementing the bandpass in C++ a simple test for mode of testing the output vectors for this function was a lot easier to implement. The whole process included implementing a function and then comparing the outputs from the Python version to the C++ version, this gave us a lot better results as it allowed us to see which functions were incorrect, instead of implementing it and then debugging afterwards. The PLL was implemented with a struct vs the Python implementation with a dictionary. This differs from our implementation in mono because we get the output audio and then start the debugging. This method is easier in terms of finding issues but slows down the process a little bit. After verifying that all the vectors are correct we did the same thing we did in Python, we wrote the left channel and checked if the output had split channels. The first run again did not work at first but this is because we had a segment error for our downsampling for mode 0, this was due to not initializing the state vector with 0's. This error was easily fixed using print statements. After this, the audio split correctly. When interleaving on the VM, the audio was not splitting correctly still, this was due to a Windows issue playing mono audio instead, and when testing on the RPI, the audio was split correctly.

Multithreading

Implementing multithreading is important as it allows multiple concurrent events to run at the same time. This is ideal for a system such as a radio because it allows information like the track name and artist name to be seen while the music is playing on an FM station. Concerning the project, we mainly used threading for the RF front-end to produce and push the FM demodulated data to a queue while the audio thread

processes data for stereo and mono audio. The RF thread runs much quicker than the audio thread so we push data to the queue quicker than it is used by the audio thread. We started by creating a ThreadSafeQueue class where we created methods for pushing, popping, checking if the queue is empty, and printing the elements of the queue out. After creating the class, the RF front-end thread was created and all functions and variables used to generate the FM demodulated data were inserted into the thread function in the code. The audio thread was also created where everything related to mono and stereo separation was added. From the lecture notes, we learned about mutex and how locking the queue would be important since we don't want both threads accessing the queue at the same time. While pushing the data to the queue, the queue is locked so the audio thread cannot access it and vice-versa. The FM demodulated data was popped from the queue and used for mono/stereo audio. When we first tested our implementation, around 60 blocks were processed before heavy static and white noise. It was discovered that the functions we had created for debugging to check if the queue was empty and print the contents were locking the queue and then not unlocking it resulting in a lot of white noise. Upon testing, there was a constant underrun at block 35 for all modes (mono and stereo). A queue size of 3 was implemented since we realized the queue size was always increasing. With this new addition, the queue size could reach a max value of 3 before the RF thread stops pushing more data into it. However, the underrun issue persisted. Upon block size manipulation, it was noted that the underrun would change depending on how the block size was changed. For example, a 2x change would make the underrun happen at half the amount of time, meanwhile, a 0.5x change in block size would make the underrun occur at 2x the time (block 70). This makes sense since it takes 2x the number of processes if there is a larger block size. Threading was removed and the issue was traced back to our mono mode. Without multithreading, the underrun would still occur at block 35. There was an attempt at reducing the runtime for mono to try and remove the underrun issue by not using modulo in the splitting of the I and Q samples function. With all of these attempted fixes, there was suddenly no underrun on March 28, 2024, before submitting the final code for the project. A rough skeleton was created for the RDS thread as well, but it is not fully implemented as RDS was not finished. The variables and all functions used for processes such as channel extraction, carrier recovery, and demodulation were added to the thread but it was commented out because as it wasn't fully functioning.

RDS

Many Aspects of RDS are very similar to Stereo in terms of the theory and the processes that go on in both paths. The main difference is the far greater length of RDS and the RDS path only really affects Mode 0 and Mode 2. To start the process, we first need to do channel extraction. We obtain the RDS Band centered around 57kHz by convoluting a BPF filter to the output of the front end, which we call demod. We set the sampling rate for this filter to be the intermediate frequency, which is 240 kSamples/sec for both Mode 0 and Mode 2. We obtain this value from RFfront.cpp by taking the input sampling frequency of 2400kHz for both Mode 0 and Mode 2 and downsampling by 10. We convolve with a range of 54kHz to 60kHz and receive an output which is entered into the float vector rds_filtered.

We then square rds_filtered nonlinearly to get a pseudo pilot tone at 114kHz, as well as other harmonics of decreasing amplitude at multiples of 57kHz. We similarly do this to a stereo mixer by running a for loop in which pointwise multiples each element in rds_filtered by itself and adds this output to a float vector called rds_nonlin. We then extract this pseudo pilot tone by creating a BPF filter with a range of 113.5kHz to 114.5kHz and the same sampling rate as the first BPF(240 kSamples/sec and convolving this filter with rds_nonlin. Just like in stereo we have to process and down-convert this pilot tone using a PLL and NCO. In RDS we change our Norm Bandwidth to a far smaller value of 0.003 because of the RDS signal having a very narrow bandwidth compared to other audio signals. This is due to the RDS signal having only 3kHz of available space to transmit information compared to stereo and mono having 16kHz and RDS encoding being very efficient. We also set the NCO scale to 0.5 to make sure to down convert the 114kHz output of the PLL to 57kHz, making sure the pilot and the RDS band are the same value so that when the mixer is run we get the RDS band on the baseband (0kHz). This

PLL+NCO process outputs the float vector rds_NCO_outp. As said previously, the pilot is then mixed with the RDS band to get the RDS band on the baseband resulting in the float vector dem_mixer. This happens because mixing two signals results in the frequency of the 2 signals being subtracted and added to create two different signals centered at the respective sum and difference. In this case, we have two copies of the output of PLL at 57kHz+57kHz and 57kHz-57kHz = 0kHz. We then run an LPF with 3kHz and a sampling rate the same as the BPFs to extract the positive part of the RDS band centered around 0kHz and receive an output of dem_rds_LPF. We then ran a resampler on it to get the symbols per second to 2375 and the samples per symbol to the value specified for Mode 0(30) and 2(43); We made sure to make the filter cutoff between 240kHz / 2 * (upsampling factor / downsampling factor) and 240kHz / 2. We have the min to ensure that the cutoff frequency doesn't go past the Nyquist, which changes based on the resampling values and could be less than 240kHz/2. We also have to make sure the filter size is equal to the upsampling factor * N_taps. We also choose symbols per second for the rational resampler to be 2375 and not 1187.5 like in the PySDR because each of our symbols contains 2 bits, therefore we have to multiply 1187.5*2 to make sure to not lose any bits. Shown below is the method to find the upsampling and downsampling factors.

mode 0 output sample rate is 24 * 2375 = 57000

- 2375 is the symbol rate and 24 is the SPS for group 99 / mode 0
- Check the unique SPS constraints for your group in modes 0 and 2

- Input sample rate is 240000 (same for mode 0 and 2 for all groups)
- GCD(240K, 57K) = 3000
  - **U** = 57000 / 3000 = 19
  - **D** = 240000 / 3000 = 80

The output of the rational resampler, dem_rds_resamp_filtered, is then put into a Root Raised Cosine filter using the sampling rate we found in the rational resampler to ensure that Intersymbol Interference doesn't mess up the Clock and Data Recovery. We finally extract the bits from outp_rrc by using an algorithm that runs for the first SPS values of the outp_rrc float vector and finds the local maximum or minimum; It then increments SPS elements through the rrc_outp float vector and detects whether it should be an LO(local minimum) or a HI(local maximum). There are a few errors with this approach though, as the local maximum and minimum values can get derailed without any error correction. For this problem, we tried to implement a peak correction system that would check +- n elements around the SPS increment to find and recenter the local maximum and minimum. We chose a value of 3 for n due to our previous n values being far too high compared to the total SPS value. We didn't want n to be greater than SPS/4 due to the fears of it detecting small dips as local minima or maxima. These minima and maxima are theoretically located at SPS/2 after a peak, so if we choose an n less than that value we should be able to reduce the number of false positives we receive. We weren't able to complete more of the project from this point due to a subpar graph coming into CDR and troubleshooting of the peak correction system that went nowhere.

One feature that can significantly improve the user experience is the addition of an interactive hardware interface. Currently, our solution needs to utilize the command line to be able to emulate or execute the radio in real-time, which requires specific parameters to be manually entered by the user. We can tackle this by adding additional hardware to the Raspberry Pi to have buttons that can switch between paths (Mono, Stereo and RDS) and modes (0, 1, 2, & 3). The process of implementing this feature is as follows. First, we can research Raspberry Pi's existing modifications or extensions that can connect via the general-purpose input/output (GPIO) ports within the Raspberry Pi. Ideally, we search for an upgrade that can handle a minimum of 7 button actions. Afterwards, we can learn how to utilize the GPIO ports and program them to perform certain actions. In this case, we would program the buttons when actuated, to switch between paths, as well as switching between the modes as well. From this, the user does not have

to memorize the command line execution and now only has to press a button to be able to listen to real-time radio audio.

A feature to improve our productivity is to utilize another method of plotting our data, which will decrease the debugging time of our validation process. There are many methods of generating plots for data, but in C++ it may be challenging to easily do so. The improvement plan can be done by experimenting with external libraries within C++ other than Gnuplot. With initial research, we can see that there exists a matplotlib, but for C++. After installing the library, we can look into the documentation and learn how to use certain vectors or any variables containing information that we want to plot. Then we can finally incorporate the library that works within our project and generate more advanced plots to debug more efficiently.

To further improve our runtime performance, we can upgrade the hardware component from a Raspberry Pi 4 to the newly released Raspberry Pi 5. This provides a significant performance boost in terms of CPU processing power. Benchmarks show that the single-core and multi-core performance is significantly improved [1]. From this article, we can observe that the Raspberry Pi 5 has almost double the performance in terms of processing power for single-core and multi-core usage. This means that we can process the RF Hardware component faster and therefore allow us to be closer to real-time processing, while simultaneously reducing the workload of the hardware component of our project.

Project Activity

	Ryan	Japman	Jeremy	Shaiyaan
Week 1	Reviewed labs and read over project document			
Week 2				
Week 3			Refactored the rest of Lab 3 python to cpp (Nicolicci homework over reading week was to convert python functions from Lab3-> cpp)	Refactored Lab 3 python to cpp
Week 4	Worked on Mono, fixing syntax errors in project.cpp file, getting the code running in the project file and setting up the header files.	For the RF front end, I split the I and Q samples from the input stream. Debugged mono and found that in the “fwrite” for mono the final vector size was the block size which is too big for mono. It kept printing the first block over and over because I was sending in the block data instead of the downsampled data. Refactored the demodulation function.	coded Mode 0 and 1 in Stereo and started debugging the issue of stereo combiner output being zero, understanding the theory of the code, and attempted PLL with state saving.	Worked on the upsampling and downsampling, setting up filters
Week 5	Debugged multithreading with Japman where audio would play for 60 blocks then white noise would play. Found that the .empty() and	Tested stereo audio and created the RF front-end thread, and audio thread, and implemented a class to perform queue operations such as pushing and popping.	Debugged PLL function by dereferencing ncoOutput, caught everyone up to speed on code and theory of	Aided creating the class for queue operations. Set up delay

	.printContents() methods would unnecessarily lock the queue and not unlock it.		Stereo, Started working on RDS implementation in python and C++, making sure variables aren't used in wrong places, working RDS in python till rational resampler.	blocks, errors would occur mainly due to phase mismatch.
Week 6	Helped Jeremy with RDS GNU plot, logged the PLLin, NCOout, before RRC and after RRC	While testing with multi-threading, I found there was an underrun issue at block 35 every time we ran. I removed threading and noticed the issue also is in mono. Tried to improve runtime for mono, implementing changes to the queue, and manipulating the block size to find reasoning for underrun occurring. Finally, I created a rough skeleton for the RDS thread with athe ll functions and variables used in RDS path but not working.	Refactoring Python code for RDS to C++, Started to graph RDS with Ryan's Help using GNUPlot, created algorithm for bit detection, tried to improve bit detection and graph coming into RDS CDR	Debugging and aiding in multi-thread ing.
Week 7	Worked on the final report			

Ryan

I implemented the fast downsampler that also does convolution inside of the function. The fast downsampler logic was not too difficult of a challenge. Implementing the resampler with convolution inside was a pretty difficult task as you had to access the correct element in both the h vector and the x vector. The lecture notes provided a good gateway into resampling mixed with convolution, and after doing a lot of calculations across multiple y elements, you could calculate the correct x element that you wanted to pull from. The mono path took longer than stereo because we had to develop the downsampling and resampling, whereas while working in stereo we already had these functions available to us. Resampling was a huge challenge because I either messed up the element indexing or messed up with the state saving. A major challenge was getting started on multithreading due to the amount of understanding needed behind it. In the end, working with Japman on multithreading allowed us to combine our problem solving skills and debug the white noise issue we were running into while working with multithreading. This was due to looking at the size of the queue at every block number for the RF front and we were locking the mutex. This was solved by commenting the line where we call the size of the queue.

Japman:

For this project, my main contributions were implementing multithreading and debugging the underrun issue that was seen at block 35 in multithreading that I traced back to mono and tried improving runtime to fix it. When I started multithreading, I began working on the RF thread first. I wanted to go step-by-step in multithreading by making sure one part works before moving on to the next. Because of this, I made sure that the RF thread was correctly pushing the data to the queue before moving on to the audio thread. I made a ThreadSafeQueue class and created methods for pushing, popping, checking if the queue is empty, and printing out the contents for the queue. The first challenge was making sure I was printing out the correct elements because I was passing in the actual data to the queue. When I called the .printContents()

method the actual data was being printed and it was difficult to interpret. I spent some time trying to understand it then I decided to push the address to the data instead. By doing so, the queue elements were easy to see once I printed them and it assured me all the blocks were being pushed to the queue. Another major challenge was attempting to solve an underrun issue at block 35 when we implemented multithreading. Our queue size was defined with a size when we first created it and we realized that the queue size kept getting bigger when we printed the size because the RF thread was pushing quicker than the audio thread was popping. I defined a queue size of 3 and also tried changing the block size to understand the issue better. If I scaled the block size up, the underrun would happen quicker and vice-versa. I decided to remove the threading and go back to test stereo to see if the issue persisted. To my surprise, the underrun would still occur. I went back and tested mono and still saw an underrun. I tried to make the implementation quicker by making the splitting of I and Q samples branchless which didn't have much of an impact. Upon trying all these fixes and testing the code before the deadline, the code ran with no underruns.

Jeremy

I primarily worked on setting up Mono and Stereo in Python and C++ and working through RDS up until Manchester encoding. Although setting up the state saving function for Mono was fairly easy due to the python code for Lab3 being fairly similar, I had some difficulty with Stereo PLL state saving. Although I realize the error now, I was not dereferencing ncoOut or PLLin because I thought that they didn't change in the function. This led to the ncoOut variable being cloned in the function and instead of the global ncoOut variable being changed only the copy would change. Since this copy would terminate after the function finished, this led to the ncoOut being whatever I set the initial value to. My other group members quickly realized the error and continued to code Stereo with me. My other team members ended up changing PLL a bit when multithreading was implemented, so I didn't fully understand the code going forward into the project but I understand after reviewing it now. It was streamlined so the values that are changed in the function are put in a struct in the filter.h folder and are dereferenced and called in the PLL function arguments, making it easier to read and work with.

Peak correction for RDS Manchester encoding was another challenge that I faced while doing this project. Because there would sometimes be a false positive for local minima and maxima I tried to implement a peak correction system that would recenter the maximum of the peaks, and I had to pick a value that would go +- n elements around the supposed peak and find the actual peak or through. Picking this value of n was difficult, but I set a limit of less that n/4 due to the issue of detecting a false local minima/maxima from the dip between a HI and a HI or a LO and LO. This dip occurs at n/2, so I set the limit to n/4 to be safe.

The input to the CDR was also a large issue, as the graph was sometimes too smooth to see the difference between two HI's or two LO's next to each other. Looking at the code after the final commit I realize I made a silly error of calling the size of the filter as min{(u*d) * 2375/2, 2375/2} instead of 16kHz. I remember talking with other people working on the project while working on RDS about the resampler cutoff and how it would be 16kHz if you were filtering and resampling as opposed to just resampling, where you'd actually use the equation min{(u*d) * 240000/2, 240000/2}, because having a cutoff larger than 16kHz would introduce unwanted noise, as seen with the cutoff frequencies for mono and stereo . I also mistaked symbols per second for sampling frequency when writing the min equation. There may be more of these wrong values that were committed before I could take a thorough look though the code.

Shaiyaan

Throughout this project, my main contributions were focused at the beginning of the project. These contributions consist of working on the mono and stereo. The first few sessions consisted of me refactoring lab 3 from python to cpp and fixing issues along with the lab as there was feedback given back from the TAs regarding our lab 3. After completing this task I moved onto working on the mono portion

of the project helping Ryan with any issues he found and help create the filters from the pseudo code provided in lecture. Eventually we had a few issues implementing the upsampling and downsampling such as an incorrect calculation of ratios and incorrect use of filters causing deletion, distortion and shifting. Once mono was complete I moved onto stereo which was mostly the same concepts in mono being translated. An issue eventually arised that the delay block had an phase error as the first elements would not be padded with 0. In the final weeks of the project I was not as involved with the project mostly helping with debugging and aiding Ryan and Japman with multi-threading.

Analysis and Measurements

Mode #	# accumulations for front end per 1 block	# accumulations for mono output per 1 block	Number of samples per 1 block	Num of accum per mono
0	**5120*2*101**	**1024*101**	**1024**	**1111**
1	**12000*2*101**	**1500*101**	**1500**	**1717**
2	**5880*2*101**	**1080*101**	**1080**	**1200.777**
3	**6000*2*101**	**2205*101**	**2205**	**650.660**

Mode #	# accumulations for front end per 1 block	#accumulations for stereo output (output of combiner so includes mono and stereo path) per 1 bock	Number of samples per 1 block	Total number of accumulations per sample
0	**5120*2*101**	**2*1024*101+5120*2*101**	**1024**	**2222**
1	**12000*2*101**	**2*1500*101+12000*2*101**	**1500**	**3434**
2	**5880*2*101**	**2*1080*101+5880*2*101**	**1080**	**2401.55**
3	**6000*2*101**	**2*2205*101+6000*2*101**	**2205**	**1301.319**

Front End Runtime

5 second Audio clip: RUNTIME IN ms	Mode 0 13 taps	Mode 0 301 taps	Mode 0 101 taps	Mode 1 101 taps	Mode 2 101 taps	Mode 3 101 taps
Generating LPF coefficients	0.151665	0.208865	0.086927	0.146316	0.159187	0.199835
readStdinBlockData	1.353	0.72576	0.79836	0.79836	0.84491	1.2953
Convolution and demodulation	0.471	12.135	4.3499	4.775	4.7576	5.229

Mono Runtime

5 second Audio clip: RUNTIME IN ms	Mode 0 13 taps	Mode 0 301 taps	Mode 0 101 taps	Mode 1 101 taps	Mode 2 101 taps	Mode 3 101 taps
Generating LPF coefficients	0.060203	0.19798	0.01387	0.040832	5.71974	4.86384
Popping queue	0.575111	0.263	0.05409	0.13013	0.11547	0.15992
Resampling and convolution	0.09118	2.50397	0.903129	0.67513	1.0777	1.6348
TOTAL RUNTIME	0.70407	2.793	0.99313	0.83476	1.2332	1.8224

Stereo Runtime

5 second Audio clip: RUNTIME IN ms	Mode 0 13 taps	Mode 0 301 taps	Mode 0 101 taps	Mode 1 101 taps	Mode 2 101 taps	Mode 3 101 taps
All Pass Filter(Delay Block)	0.0823519	0.041426	0.0614334	0.0526607	0.0700812	0.0700346
BPF Convolution with pilot tone	0.484245	11.851245	4.04454	4.6695	4.54879	3.87989
PLL	1.1779056	1.071463	1.09989	1.27818	1.254399	1.041798
BPF Convolution with Stereo Band	0.5983176	11.8497	4.091347	4.70668	4.62863	3.8712
mixer	0.029641	0.018769	0.018509	0.0206636	0.02666	0.02055
Resampling after mixer	0.092686	2.4519399	0.818459	0.58875	0.972137	1.55686

interleaving	0.00730407	0.005450172	0.006485	0.0038878	0.006112	0.007928
Stereo Combiner(Extracting channels)	0.0131708	0.010803133	0.012289	0.006049	0.01332	0.01342
TOTAL RUNTIME	3.07069	29.89515	11.2627	12.17	12.6363	12.2444

The runtime agrees with the calculations when the input size of the vector that is being convolved with the h vector is larger, which means that more multiplications have to be done. The runtime of only the mono path, because we only measured between the beginning of the delay block and at the end of the final mono resampling function called. The stereo path took approximately double the time of the combined RF and stereo vs RF and just mono, this makes sense due to the number of calculations needed to be done for the stereo path, the mixing and much more. As can be seen, the convolution is a huge bottleneck for this specific project, and the impulse response generations and a lot of the other functions do not need as many resources,

Conclusion

Throughout the project, we each had to deal with our hardships and challenges as well as challenges implementing each of our sections into a cohesive working system. Although the technical aspects of the project were difficult, the interpersonal connections and teamwork that we had to create and maintain throughout the project were vital to our continued success. Without it, we wouldn't have been able to spend all the time and effort we did separately and together to work on the code. Although we were only able to finish the Mono and Stereo section of the radio receiver, RDS was 60% complete, and with a few more days of consistent effort, we would have achieved a fully working system. A challenging yet enlightening project, our group thanks the COMPENG 3DY4 staff and Professor Nicolici and Professor Kazem for their continued support and for the opportunity to work on this project. Thank you from Group 4.

References

[1] https://pimylifeup.com/raspberry-pi-5-benchmark/
[2] “COE3DY4 Project Real-time SDR for mono/stereo FM and RDS.”
[3] “COE3DY4 Class Notes.”
[4] “COE3DY4 Class Python Code.”
[5] M. Lichtman, “A guide to SDR and DSP using python,” PySDR: A Guide to SDR and DSP using Python, https://pysdr.org/ (accessed Apr. 5, 2024).