3DY4 Project Report

Group 16

Divesh, Rayan, Uzair, Ankur

chudasad@mcmaster.ca, hussar9@mcmaster.ca, jawaidu@mcmaster.ca, kejria1@mcmaster.ca

2021-04-09

1. Introduction:

The objective of this project is to develop a software application that can process FM signals to produce consumable audio. The other primary objective is to also meet physical constraints with regards to computation power and memory to have a system that operates in real time.

2. Project Overview:

In this project, as stated above, many of the basic principles of signal processing were applied such as FM demodulation, filtering etc.,. In this part we will break down the fundamental components of the signal flow.

We pull data in “blocks” or chunks in order to process them in reasonable time to feed back in as close to real time as possible. Block processing requires state saving in various components such as the convolution functions that perform our filtering and even in the case of the PLL. We will discuss more about this in our implementation section.

Starting from the very top we need to low pass filter the bands of information we primarily work with, that is the mono, stereo and rds bands. For this purpose we use a low pass filter with 100 kHz cutoff. When it comes to filtering we generate the necessary coefficients before we begin any processing of the data since these will not change during run time. To low pass filter we use an impulse response filter and a similar approach is taken for band pass filtering.

Once we have the coefficients we can complete the filter by performing convolution using the impulse response coefficients. The data from this point is then broken down into I and Q samples which are viewed as the sine and cosine components (in-phase and quadrature). These components will then be used to demodulate the signal using an arctan and differentiator.

Then the stereo path begins, similar to mono we generate the band pass filter coefficients using the impulse response methodology. The first bandpass is to extract the pilot tone that we lock onto for the stereo channel. The second bandpass is to extract the actual stereo channel. Before we start demodulating, we have to pass the recovered pilot tone to a phase locked loop (PLL) in order to synchronize it. We can then proceed to mix the pilot with the stereo channel to demodulate the double sideband suppressed carrier (DSBSC). Then to eliminate the two sidebands (produced from mixing/demodulation) we simply low pass filter and down sample again in the same way as our mono path.

From this point we generate the 2 channels of audio, the left and right. This can be done by combining the stereo signal with the mono audio. For the left channel we subtract the stereo from mono and for the right we simply add and then divide by 2 in both cases. The product can then be output for consumption.

3. Implementation

3.1 Front End:

For the front end as mentioned we make use of a sampling frequency depending on the mode. This is demodulated by our arctan demodulator function. This in principle computes the arctan and differentiated output all in one stage/calculation to save on computation. Since the regular arctan function can be taxing we take the I and Q samples and take the derivative of I and Q using the previous samples. Then we simply add the I sample and Q sample with dQ and dI respectively and divide by I^2 plus Q^2 to compute the differentiation and arctan in one calculation.

3.2 Mode 0

Within mode 0 we use the basic initial parameters without any upsampling. We start with a sampling rate of 2.4 MHz, and in the end the audio is sampled at 48kHz. We have a pair of low pass filters at 100kHz to extract the radio bands we work with and 15kHz to extract the mono channel. We also have a pair of bandpass filters which we will discuss more in the stereo section.

Once we have the demodulated data we can begin mono processing, as seen in the flow chart above. The first stage is to low pass filter at 15kHz to extract the mono audio and then down sample to the output sample rate of 48 ksamples/sec.

3.2.1 Optimizations:

As mentioned one of the first optimizations was to combine the convolution and downsampling. Since we only take data for some downsample factor we only need to perform convolution at those points. To do this we simply skip the samples in between by increasing the for loop increment to what our downsampling factor is. For this approach we did require a second indexing variable *d* as in our code, to store the convolution output in consecutive memory locations and not have gaps. We faced an issue initially where we used the loop index for our output but since we were incrementing by the downsample factor we found empty samples in between. This is the rationale for having that *d* indexing variable.

The other optimization was also in regards to convolution, this time with the IQ samples. Instead of having to convolutions for each one we combine them and also perform the same decimation technique.

3.3 Mode 1

For mono mode 1, a few initial parameters differ which cascades into design changes for the filtering and convolution compared to mode 0. For mode 1, the intermediate frequency (IF) is sampled at a rate of 250Ksamples/sec instead of 240Ksamples/sec. Because of this, in the non-optimized version of our implementation, we included an 'expander' which increased the sample rate by 24. This was achieved in our Python model first to model the non-optimized behaviour. The data points were separated by 23 zeros which caused the vector length for the data to increase which was then followed by the LPF of 16kHz. However, this changed the sampling rate to 6Msamples/sec (more on this later).

After the LPF, the data points were decimated by 125 to reduce the sample rate to 48Ksamples/sec as required. When implementing in Python, we used the estimatePSD to generate the graphs where we observed the output to not match up with what was expected. The values were much smaller and the LPF was not working correctly. To fix this, we went back to the initial values we had set to make sure they were correct.

After some debugging and reviewing lecture content, we realized we forgot to account for the gain which needs to be added due to the upsampling which was causing the sampling rate issue and incorrect coefficient generation. So by multiplying our audio_taps value and the generated coefficients by 24 (the upsample value) this issue was resolved.

3.3.1 Optimizations:

When creating the optimized version in Python, we faced some issues initially. Our biggest hurdle was understanding how to implement the bounds and calculate the correct index values for the two arrays we would be using in our convolution. The optimized version of Mode 1 required us to combine the decimation and low pass filtering in one function and remove the expander (upsampled padded zeros) completely. This was achieved by using values from our array, which would normally be upsampled in the unoptimized implementation, and using the corresponding values from the second input array which would be needed to compute a convolution. This was done so as to avoid the need for padding our first input array with zeros. For example: [1,0,2,0,3,0] is and array with an upsample value of 2, to optimize the operation, we would work with the array as if it were [1,2,3] but take the values from the second array which correspond with the padded array's indices. We ran into issues when attempting to calculate the indices manually after each pass of a while loop nested within a for loop. We ran into many complications and decided that it would be easier to run 2 nested for loops, where the outer loop iterates by 1 (to iterate through the second input array and resultant array indices) and the inner loop iterates by 24 (the upsample step size to calculate the correct array index for the first array since it would be non-padded in the optimized implementation). Our double nested for loop implementation proved to be successful in Python and allowed us to seamlessly transition to the C++ implementation.

3.4 Stereo

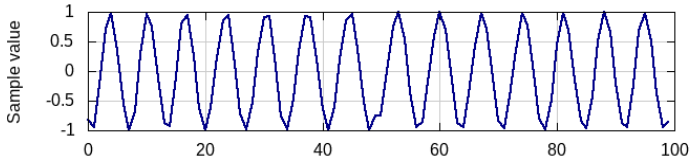
Compared to the Mono signal flow, the stereo signal flow involved using a band pass filter to filter out the stereo channel from 22kHz to 54kHz and the pilot tone 18.5kHz to 19.5kHz. The phase locked loop (PLL) would lock to the pilot tone and produce the 38Khz subcarrier to downconvert stereo channel data to baseband allowing for to continue with the processing step. The signal flow diagram from the project document accurately captures the flow that we followed. We also encountered some issues getting the stereo to run live and heard a lot of noise on the output. We went through all the different steps and stages like our coefficients, our filtering outputs and compared it to the values of our python code. To do this we mainly employed print statements in python and log files in c++. Those turned out to be fairly similar in values and in the end was an issue with the way the output was stored and modified in memory, we cleared the arrays at the end of block processing to address this issue.

3.4.1 Python modelling

The Python modelling was done primarily using the libraries part of scipy for convolution and coefficient generation for FIR impulse response. We tried a single pass approach here and limited it to a couple of blocks in order to speed up the modelling process as using our own convolution function was increasing the run time and slowing down the modelling stage. We verified the output audio file and saved images of the psd of the left and right audio channels and used this as a point of reference for the C++ implementation.

3.4.2 C++ Implementation

Compared to mono processing, stereo requires the use of bandpass filters in order to recover the stereo channel from the demodulated data stream and the pilot tone. After each filter is applied we graph the psd of the output to see if we have frequency peaks at the frequencies in the pass band of the filters as a check to ensure that the filters are working accurately and have significant power. We had issues in mono processing where the shape of the psd was correct but didn't have the right power resulting in issues further down the line.



PLL Block Functionality discontinuity

In order to ensure the PLL is correctly being synced between blocks we graphed the last 50 samples from one block with the first 50 samples of the next block and were looking to see a consistent sinusoid, initially we got a slight discontinuity which can be above. At the 50 samples

mark there is a slight delay between the last sample from the previous block and the first sample of the next block which was later found to be an indexing error which when fixed looked like a continuous sinusoid.

The graphing done above really helped clear any fundamental issues we had in terms of array sizing and filtering parameters being set correctly as the issues would skew the graph and comparing it back to the output graphs from the modelling stage allowed us to locate which part of the program was providing the wrong output.

3.5 Block Processing:

The principle of block processing was used to help the system run in realtime. State saving is important since the next block will often require samples/data from the current block to complete its calculations. In the case of convolution we save the samples at the end of a block for the next one. The size of this piece of state is equal to the transfer function. We also required state for the PLL to track the integrator, phase estimate, I & Q feedback, the trailing value of the nco and most importantly the trig offset. To keep all of these stored and easy to pass to the pll function we used a c++ struct.

3.6 Threading:

The mode 1 path that requires more computation than mode 0 and this is where threading becomes necessary. We implemented a producer and consumer design in our C++ implementation. Where we essentially pull data from stdin from the producer thread, this then handles the front end computations and loads them onto the queue. We make sure to not overload the queue by having a max of about 5 blocks at any given moment, if this is reached we lock the mutex on this thread and wait for the consumer to complete its processes. The data is then processed in the consumer to get mono and stereo audio and write it to stdout. Similar to the producer, we have a lock condition where an empty queue stops the thread temporarily while we wait for the producer to provide some more data. This structure allows 2 separate processes to happen at the same time.

3.7 RDS

We did not complete this portion of the project due to time constraints and were only able to set up a simple single pass model in python. We set up the basic filtering following the project specification to extract the RDS path. Resampling was done following the same method as Mode 1 unoptimized (using an array padded with zeros).

For manchester decoding we simply use the sample points and compare the values relative to each other. So a low bit would be a low followed by high, and inversely a high bit

would be a high followed by a low. As for differential we XOR the previous between with the current for all the bits we got from Manchester. In this process the first may often be wrong.

For frame synchronization we had to use the brute force approach. Since there were no better algorithms available, we used a while loop which continuously pushed the bitstream down by 1 until the 26 consecutive bits synchronized. Then using bitwise matrix multiplication, we used the check block of bits ANDed with the parity matrix to perform multiplication, followed by XOR for addition. The XOR was applied between the resultant bit and the previous AND multiplication value.

4. Analysis and Measurements:

4.1 Mode 0:

Process:	Block size:	Taps:	Decimation:	Multiplies/block
RF FRONT End Processing	51200	151	10	773120
Audio Filtering	5120	151	5	154624
Stereo Carrier Recovery	5120	151	1	773120
Stereo Channel Recovery	5120	151	1	773120
Mixing	5120	1	1	5120
Stereo LPF	5120	151	5	154624

4.2 Mode 1:

Multiplications: 151*2 = **302**
Multipliers: **3**

The first of the two multiplications is from checking the bounds to make sure the value doesn't fall outside of the length of x*24 (data points vector*upsample). The other multiplier is from another bound check of decim*i-j to ensure the correct range.

4.3 Non linear Functions:

cos function: 2*151 = 302
Sin function: 151

4.4 Stopwatch Performance:

Note: Average across 5 test samples

Mode 1 convolution:	0.6472 ms

Mode 0 convolution	2.6447 ms
PLL	2.6862 ms
Fm Demod	0.0325 ms
Impulse response Low Pass	0.4014 ms
Impulse response Band Pass	0.2729 ms
Front end Block processing	11.017 ms
Consumer block processing Mode 1	11.405 ms
Consumer block processing Mode 0	6.0166 ms

5. Potential Improvements:

One area could be to further explore the most optimal number of blocks to load the queue with. In our current end product, we use 5 blocks as the maximum allowed at any given moment during execution. So playing around some value larger than 5 may help reduce some blocking behaviour and let these threads run more asynchronously.

Another area of improvement would be within the RDS portion of the program, specifically the resampler. Given the time constraint, we were unable to implement the optimized version of the resampler and had to resort to using the padded zero array method. Optimizing this would reduce the runtime.

6. Project Activity:

Week	Progress
1 & 2	No progress made
3	Divesh: convert signal flow from lab3 into c++ for front end mode 0, optimize convolution block and with IQ optimization. Adding state for convolution in block processing c++ Ankur: Signal flow graph in a single pass approach and setting up logic for block processing approach and associated debugging. Uzair: assisted with convolution debugging in C++ Rayan: assisted with mode 0 debugging in C++
4	Divesh: Converting python model of stereo into C++, creating bpf in c++, reading and writing data out live Ankur: Stereo Python modelling, C++ translation and optimization and testing on board.

	Uzair: Python modelling for mode 1 non-optimized and optimized versions Rayan: Python/C++ mode 1 non-optimized and optimized debugging
5	Uzair: Mode 1 python debugging for optimized version and converted to C++ for non optimized - debugged issues Rayan: Mode 1 Optimized in C++ completed, converted from python Mode 1 after debugging Divesh: adding threading related producer/consumer functions, adding mutex controls. Integrating Mode 1 code into threads along with mode 0 Ankur: Integrating mode 1 code with non-threaded code and testing/debugging
6	RDS single pass python model (able to see symbols) Divesh: decoding (manchester and differential in single pass) Rayan: Resampler (unoptimized), Frame synchronization with no error checking Ankur: Setting up filters needed to recover RDS data IQ data and modifying the PLL to get quadrature component Uzair: RDS frame synchronization setting up brute force algorithm + syndrome checks

7. Conclusion:

This project was a great learning experience for working with real time systems. More specifically getting to see mathematical and conceptual DSP come into play in the real world was very fulfilling. The programming concepts like modeling, threading, optimizations and even debugging were also eye opening and can be extended to other projects in the software space.

8. References:

FM Arctan Demodulation technique:
https://www.embedded.com/dsp-tricks-frequency-demodulation-algorithms/
Project Specifications Document
Prof Notes (slides)

Help from Peers:

Minhaj Shah - Convolution optimization, explained the concept to us
Olayiwola Bakare - Mode 1 optimization algorithm explanation, rds frame sync how to confirm output