COE3DY4

Computer Systems Integration Project

Instructors: Dr. Kazem Cheshmi, Dr. Scott Chen

Akash Sharma—L01—shara98—400300570
Isaac Shoong—L01—shoongi—400312914
Samarth Mehta—L01—mehtas30—400260946
Andrew Ye—L01—yeh31—400318465

Introduction

In this project the goal was to develop a radio system using industry specifications. Such as engineering the three objectives of a mono audio processing block, a stereo processing block, and a radio data system (RDS) processing block. Utilizing tools such as Python and C++, as well as concepts learned in class, such as block processing, resampling/convolution, phase-locked loop (PLL), filters and more.

Overview

The big picture of the project was to implement the knowledge gained from electrical and computer engineering, such as convolution, wave processing and data processing from first principles to the complex interrelationships between them.

The overall goal of this project is to create a software-defined radio (SDR) system. Traditionally a radio system was created using more hardware components, such as physical filters for physical electronic waves. Where as a SDR implements software based components to process and convert digital signals. SDRs are much more flexible than traditionally hardware systems.

There are four stages to our SDR. RF front-end, mono processing, stereo processing, and RDS. Concerning the RF block, an RF signal is captured by an antenna and converted into the digital domain by producing an 8-bit sample in-phase (I) and another 8-bit sample quadrature component (Q). That is all for the RF front-end of the stereo and mono processing system. However, for RDS, RF front-end it is different, where it extracts the FM channel through a low-pass filter (LPF) and decimation to develop the signal in IF frequency.

The mono block extracts the mono audio channel, which lays between the frequencies of 0 kHz to 16 kHz. It also reduces the mono data sample rate to 48 kilosamples per second (kS/s). The input to the mono block is the demodulated FM signal, with an IF sampling rate that depends on the mode, where for mode 0 it is 240 kS/s. The output of the mono block is a 16-bit sample which is the sum between the left and right audio channel. This sample can be listened to, but obviously does not have the depth that stereo does.

The stereo block extracts a 19 kHz pilot tone from the signals with frequencies between 23 and 53 kHz. This is done using a band-pass filter (BPF) and downsampling, as well as a PLL. The input of the stereo block is the same as the mono block, but due to the difference in processing, such as completing stereo channel extraction and stereo carrier recovery. The output is the difference between he right and left audio channels, which is the product of the extraction and recovery data. Thus using the mono and stereo data, it is possible to isolate for the left and right audio channels.

Lastly, the RDS block recovers the subcarrier from the RDS channel between 54 and 60 kHz. It is then converted through a digital communication method and resampled, then clock and data recovery is preformed. A bit stream is then generated and frame synchronization is done in the data link layer to produce information. The input is the same as the mono and stereo data, where the output is now in terms of bits and words in radio data.

Implementation

In the first laboratory assignment, we were tasked with writing our own discrete Fourier transform (DFT) and finite impulse response (FIR) coefficient generator for a low-pass filter (LPF) in Python. To validate our custom implementation, we compared the results of our own functions to that of pre-existing functions from the NumPy and SciPy signal processing libraries. The purpose of this was to familiarize ourselves with digital filtering and convolution, which would ultimately represent the majority of all the computation for this project. In addition, we slowly transitioned away from single-pass convolution towards block processing, which also proved to be fundamental for streaming.

In the second laboratory assignment, we reimplemented our custom signal processing functions in C++, as this was the language of choice for the final product. We validated our LPF once again, this time by extracting the first harmonic of a square wave with randomized parameters. We also began to use gnuplot to visualize the time and frequency domain signals—this would become a useful tool in debugging later on.

In the final laboratory assignment, we moved towards frequency demodulating the input signal. This involved putting our LPF functions from the previous lab to use, first extracting the FM band (up to 100kHz), demodulating, then using another LPF to extract the mono audio band (up to 16kHz). We also reimplemented a power spectral density function in C++, but never ended up using it during the course of the project. Our demodulation and power spectral density functions were checked against pre-existing functions from a provided library, `fmSupportLib.py`.

Our first step towards the project involved adapting our code from lab 3 to read raw data from a file. This was achieved using a snippet of code provided at the 11:15 mark [1]. This allowed us to test our code with consistent input data. A similar piece of code provided around the 17:15 mark was used to write the audio data to standard output [1]. The first working version of the SDR could only play mono audio and operated in mode 0, which meant no upsampling was required. This was the most basic form possible, which we used to debug the integration of our LPFs, demodulation, and I/O. One of the first problems that plagued our implementation was memory allocation. Our mono-only SDR had some stuttering the in the audio but otherwise proved that our code worked. But as we added more and more components, we ran in to errors that indicated corrupt memory and that vectors were being freed twice. We later discovered that we had not used `std::vector::reserve()` to allocate memory for each vector, so some functions were writing into the memory blocks of other vectors. Once this was realized, we placed a block of code at the top of every function that would clear, allocate memory for, and resize all the relevant vectors.

Table 1—Input, intermediate, and output sampling rates for four modes in kilosamples per second.

	Mode 0	**Mode 1**	**Mode 2**	**Mode 3**
RF f_s (kS/s)	2400	1152	2400	2304
IF f_s (kS/s)	240	288	240	256
Audio f_s (kS/s)	48	48	44.1	44.1

Expanding the mono path to other modes (with different sampling rates, as outlined in Table 1) meant implementing upsampling. Our original method involved padding zeros between each sample and then passing it though a LPF. Here we ran into a second roadblock. Because of the zero-padding, the subsequent convolution with the FIR coefficients took an unreasonable amount of time. We first tried to optimize the convolution by iterating through the output and FIR coefficients instead of the output and input, but that was not enough. We were then directed towards a resampling algorithm. The idea was that since most of the data was now zeros, we did not need to perform the products of the samples with zeros. This reduced the computational load by two orders of magnitude. In addition, since we were to downsample immediately afterward, we also did not need to compute the samples that would not make it to the output. This again reduced the computational load by another three orders of magnitude (for modes 2 and 3). Overall, this resulted in a very fast filter and seamless audio.

In the stereo block there were three tasks that needed to be completed. The first of which is to complete stereo carrier recovery. We implemented a BPF and validated it against an existing Python signal processing library, like how we validated our LPF. Both our LPF and BPF FIR coefficient generators were implemented using pseudocode provided in [2]. Once that was completed, we were able to use the resampling function to convolve the coefficients with the input data. The BPF had a cut off range of between 18.5 kHz to 19.5 kHz. This extracted the 19 kHz pilot tone needed for frequency shifting, however, it was quite noisy and out of phase at this point. We used a PLL and numerically controlled oscillator (NCO) to generate a filtered and in-phase pilot tone to be fed into the mixer. Initially, our stereo audio contained what can be described as a Geiger counter clicking. We tried several ideas, like increasing the block size and filter taps, validating block state-saving, and re-recording our input data sample. Eventually, after comparing with a colleague, we realized that we had forgotten to implement state-saving within the PLL itself. This was an easy fix—we just needed to maintain the last element each time the PLL was called.

The second task is stereo channel extraction. This involved using a BPF with cutoff frequencies at 22 kHz and 54 kHz to extract that band from the demodulated data.

The final task is stereo processing. The first step is to perform a point-wise multiplication between the stereo channel data and the 19 kHz pilot tone. However, because the mono and stereo data had been passed through different filters at this point, the samples no longer lined up. Thus, the mono data must be delayed by a certain number of samples in order to multiply the correct samples together. This delay was found experimentally, by trying different values and listening to which one produced the greatest separation between the left and right channels. This was done by using a test input file that had drastically difference audio in each ear. Then the data is resampled to get our desired sampling rate of 48 kS/s, using the resampling function. The last step is to combine both the stereo audio and mono audio to isolate for left and right channel audio. This was done by exploiting the fact that mono is transmitted as the sum of the left and right channels, and stereo is transmitted as the difference between the left and right channels. By adding mono and stereo, we get the left channel, and by subtracting them, the right channel. After halving the amplitude, we got our final output data that could be played as audio.

Multi-threading was implemented to process two blocks of data in parallel, in order to speed up our SDR. We had two working threads, one for the RF front-end and one for audio processing, as well as an incomplete thread for RDS. We decided to follow the producer/consumer model. Our producer was the RF front-end thread, which pushed each block of demodulated data to a queue, and our consumer was the audio processing thread, which popped blocks from the queue. In order to avoid the two threads accidentally pushing and popping to the same location, a mutex (or mutual exclusion) lock was used. This was locked whenever a thread was accessing the queue. Without this lock, variables may be accessed and written to at the same time, which could result in undesired behaviour. We used a condition variable to notify other threads whenever the queue was free to access. We encountered an issue involving function arguments when creating a thread. We later found out that arguments had to be passed using `std::ref()` but also must be read into the function by reference using the & operator, which we initially overlooked.

Analysis and Measurements

Mono

Table 2—Multiplication/accumulation analysis for the mono data path.

Mode	Block Size 256*audio_decim*rf_decim	Taps	Multiplication/ Accumulation	Non-Linear
Mode 0	12800	51	305	0
Mode 1	6144	51	330	0
Mode 2	2048000	7497	40854	0
Mode 3	5898240	22491	130612	0

Stereo

Table 3—Multiplication/accumulation analysis for the stereo data path

Mode	Multiplication/Accumulation	Non-Linear (PLL+NCO)
Mode 0	560	12800
Mode 1	636	6144
Mode 2	41100	2048000
Mode 3	130908	5898240

Radio Data System

We did not reach a stage in RDS where there is a valid measurement for analysis.

Function Runtimes

Table 4—Runtime of various sections in SDR, averaged over blocks 11 through 20.

Function	Average runtime (ms)
Reading and splitting I & Q	0.1912572
Resample (RF)	0.8865253
FM demodulation	0.0199991
Resample (Mono audio)	0.1489678
Mono delay/APF	0.0028444
Resample (Stereo channel extraction)	0.6700436
Resample (Stereo carrier recovery)	0.366669
PLL	0.4347466
Mono/Stereo mixer	0.0260758
Resample (Stereo audio)	0.1597459
L & R separation	0.0060574
Writing to standard output	0.0079151

The run times correlate to the number of operations done. This can be seen from PLL being one of the largest times as it has the greatest number of operations to do as it takes in the demodulated data without downsampling.

Proposal for Improvement

The are many places for improvement on the project. Some of which are for the design itself. When it comes to a user interface, there is none. The program is run via the Linux command line interface. Thus when it comes to design improvements, the process of starting the radio, developing an output and selecting modes could be made more user friendly, such as implementing a GUI for the user or using physical buttons that can start the process.

In terms of maintenance and expansion improvements. The entire program could be more modular to help with maintenance. As most of the time, when it came to debugging, only one function was causing issues. Thus, if the program is very modular these issues can be dealt with quickly, without affecting the entirety of the program. Especially when it comes to processing the different blocks. As mono may work but stereo does not, thus we only want to perform maintenance on the stereo block and not the mono block. In terms of expansion, we could utilize more of the cores, as there are 4 available and only 2 are used. This is so more data could be processed at a faster rate. Another expansion improvement could be to include more channels, as the program only goes up to RDS which ends at 60 kHz. However, the FM band occupies up to 200 kHz and many more SCA subcarriers can be developed to broadcast more services.

When it comes to improving the speed of the system. There are two main things that can be done. The convolution can be switched from standard convolution to fft convolution. This will make the program much faster as the number of calculation completed is much lower. However, this method is much more complicated and must be researched more by the group to implement, as the risk of overlapping samples is quite large. Another method of improving speed is to use more cores. Where currently, two cores are used but four are available. This can allow more blocks to be processed at once, essentially cutting the processing time in half.

Lastly, to improve voice quality there are multiple things that can be done. First of which is to run the signal through more filters. This can filter out more noise, making the desired signal to be clearer. Improving the quality of filter coefficients would also help. Another way would be to improve the PLL, as it recreates the carrier signal. If the recreation is more accurate, the voice output would be much clearer. Next would be to make the program run faster, as currently the number of taps and block size is limited to the speed of the program. If it ran faster, the umber of taps and block size could increase, resulting in more accurate readings and outputs. Lastly, the hardware that collects the RF data can be improved, as this is the main process that collects the actual data.

Project Activity

Akash Sharma	
Week 1 (February 13th):	Introduced ourselves to the project
Week 2 (February 27th):	Reading documentation on mono implementation
Week 3 (March 6th):	Helping with troubleshooting the C++ mono code
Week 4 (March 13th):	Reading documentation on stereo
Week 5 (March 20th):	Coded the stereo processing in Python
Week 6 (March 27th):	Started reading up on threading and implementing basic producer consumer models
Week 7 (April 3rd):	Implemented a solution to threading for the project and tested mono C++
Week 8 (April 10th):	Worked on the final project report activity and conclusion

Andrew Ye	
Week 1 (February 13th):	Introduced ourselves to the project
Week 2 (February 27th):	Reading documentation on mono implementation
Week 3 (March 6th):	Started to implement Lab 3 mono code, and implement into C++
Week 4 (March 13th):	Mono debugging
Week 5 (March 20th):	Adding C++ upsample and C++ mono debugging
Week 6 (March 27th):	Fixing C++ mono code as it broke during refactoring
Week 7 (April 3rd):	Finished up the threading and ensured mono working as intended. Worked on the all-pass filter for RDS as well.
Week 8 (April 10th):	Worked on the final project report implementation and run times

Isaac Shoong	
Week 1 (February 13th):	Introduced ourselves to the project
Week 2 (February 27th):	Reading documentation on mono implementation
Week 3 (March 6th):	Reading documentation on stereo implementation while helping with the mono implementation.
Week 4 (March 13th):	Created an initial python stereo implementation up to processing
Week 5 (March 20th):	Finished the stereo in Python
Week 6 (March 27th):	Debugged the C++ stereo code and started RDS implementation in Python
Week 7 (April 3rd):	Translated python RDS prototype into C++
Week 8 (April 10th):	Worked on the final project report introduction, overview and areas of improvement

Samarth Mehta	
Week 1 (February 13th):	Introduced ourselves to the project
Week 2 (February 27th):	Reading documentation on mono implementation
Week 3 (March 6th):	Debugged the C++ code for mono
Week 4 (March 13th):	Reading into stereo documentation
Week 5 (March 20th):	Created the stereo C++ code and debugged it
Week 6 (March 27th):	Finishing debugging C++ stereo code and read about threading
Week 7 (April 3rd):	Worked on the implementation of multi-threading in C++
Week 8 (April 10th):	Worked on the final project report analysis and measurements

Conclusion

In this project, students applied theoretical concepts related to signals and computer systems to the real-life application of radios. They utilized a variety of skills, including Fourier series/transforms, Python, C++, and control systems, to gain practical experience in in digital signal processing and debugging. The project was completed as a group, which allowed students to develop a stronger understanding of team dynamics and become better collaborators. The use of GitHub to share code and track progress also provided valuable experience for real-life development work. Overall, the project gave students an opportunity to apply their theoretical knowledge to a practical application, enriching their learning experience.

References

[1] S. Chen, Avenue to Learn. *SDR Project – Technical Details (MUST WATCH!!)*. (Feb 2023). Accessed: Mar 9, 2023. [Online Video]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/528090/viewContent/4085409/View

[2] S. Chen, K. Cheshmi. (2023). COE3DY4 Project Real-time SDR for mono/stereo FM and RDS [PDF]. Available: https://github.com/3dy4-2023/3dy4-project-group04-prj-tuesday/blob/main/doc/3dy4-project-2023.pdf.