COE3DY4 Project Report: Real Time SDR
Group 62

Raul Singh Sidhu, Syed Ali Raza,
Zuhaib Mohammed Quraishi, Khushi Akbari
sidhur26@mcmaster.ca, razas32@mcmaster.ca,
quraiz2@mcmaster.ca, akbarik@mcmaster.ca

04/05/2024

Introduction

The focus of this project was the development and implementation of a software-defined radio (SDR) system, capable of handling real-time FM audio streams, including both mono and stereo formats. Different operational modes were integrated to adjust sampling frequencies and associated parameters dynamically. The system utilizes an RF dongle attached to a Raspberry Pi 4 for receiving FM signals. These signals are then processed and rendered through the Raspberry Pi. The core development was undertaken using C++, complemented by Python for creating models that facilitated verification and testing of the implemented logic.

Project Overview

The essence of our project lies in the implementation of a SDR system, specifically tailored to handle Frequency Modulation (FM) audio signals. FM is a method of conveying information over a carrier wave by varying its frequency and is widely used in radio broadcasting. Our project's core objective was to design an SDR that could process these FM signals in real-time and decode them into clear, audible audio.

Unlike traditional radios with fixed hardware structures, SDRs employ software to perform most of the signal processing, offering remarkable flexibility and adaptability. In this project, we explored how different software modules can replace what was traditionally accomplished by hardware components.

Central to our system are several key building blocks, each playing a pivotal role in the FM reception and decoding process. Finite Impulse Response (FIR) filters are crucial in this setup, isolating specific frequency bands and mitigating unwanted noise or interference. The FM demodulator, another critical component, extracts the audio information from the modulated carrier wave. We also utilized Phase-Locked Loops (PLLs), which are essential in maintaining the alignment of the output signal with the reference signal, ensuring stability and accuracy in frequency extraction.

The project also incorporates re-samplers necessary for adjusting the signal’s sample rate to match the processing and output requirements. Additionally, the Radio Data System (RDS), a standard for sending small amounts of digital information alongside the traditional FM radio broadcast, was considered for integration, aiming to enhance the system's functionality.

When these components are chained together in a signal-flow graph, they form the foundation of an FM receiver in a software environment. The flow starts from capturing the FM signal, progresses through filtering and demodulation, and concludes with the precise tuning and decoding of the audio content. This requires the integration of digital signal processing (DSP) techniques and the adaptability of SDRs for different broadcasting standards and operational modes.

Implementation Details

Building Blocks and Initial Steps

Our journey through this project was marked by a progressive and iterative approach, aligning with the principles of SDRs and the intricacies of FM processing. Starting from the foundational work laid out in our laboratory exercises, we embarked on a complex task to implement an SDR system capable of handling mono and stereo FM signals. These lab components included implementing discrete Fourier transformations to convert signals between time and frequency domains and designing low-pass filters which are essential for isolating desired frequency components. We employed sinc functions to generate impulse responses and filter coefficients, providing an in-depth understanding of these filters.

As we began to work with FIR filters, our focus was on achieving effective convolution between the signals and filter coefficients. This was accomplished using nested loops, with an outer loop for phase computation and an inner loop iterating through filter coefficients. A critical technique we adopted was block processing, where we processed data in real-time without running into array sizing issues in C++. State saving was a technique we integrated in Python during these labs, ensuring continuity in data processing across successive blocks.

RF Front-End

The RF front-end of our project involved using the ‘readStdinBlockData’ function to capture the data stream, followed by splitting I & Q data. The logic divided the incoming data into I and Q vectors, laying the groundwork for further processing. We utilized the ‘impulseResponseLPF’ function to calculate the impulse response and applied convolution to both I and Q samples respectively using our custom ‘decim2’ function, which also included state saving logic. These downsampled I & Q data streams are then passed into our ‘fmDemodulation’ function to give us a final Intermediate Frequency (IF) signal. This signal is extremely important for further processing and concludes the RF front end.

Mono Path

Our approach for the mono path relied on variables set according to our predefined operational modes. Data passing through low-pass filters and subsequent downsampling were crucial steps. Notably, modes 2 & 3 were slightly more complex due to the additional upsampling component. These values are determined by analyzing the ratios between output and input sample rates. To ensure accurate audio reproduction, the IF is passed into our custom ‘decim4’ function. The naming convention can get slightly confusing, so here is the difference; ‘decim2’ does downsampling and convolution while ‘decim4’ does downsampling, convolution, and upsampling. To synchronize the timing between the mono and stereo paths, an All-Pass Filter was introduced to the mono path. The mono output was then stored for further processing or writing to aplay.

Stereo Path

In our stereo path, the IF signal goes through a detailed carrier recovery process. We use a band-pass filter function (‘impulseResponseBPF’), tailored to isolate the 19 kHz pilot tone essential for stereo signal identification which selectively filters the IF signal. Following this, a Phase-Locked Loop (PLL) effectively locks onto the pilot tone frequency. Important to the PLL is a Numerically Controlled Oscillator (NCO), which works to stabilize and maintain the correct phase and frequency of the 19 kHz pilot tone, producing a clean, noise-reduced carrier signal.

In the process of stereo carrier recovery, we next focus on isolating the stereo channel, employing a band-pass filter with a frequency range of 22 kHz to 54 kHz. The filtered output, combined with the carrier signal refined by the PLL, is directed into a mixing function (stereo_mixer). This step involves blending the two signals through a sample-wise multiplication, effectively normalizing the signal.

The final phase of the stereo path combines the previously isolated mono audio signal with the freshly processed stereo information using a custom function called ‘stereo_combiner’. The digital filtering and sample rate conversion uses the same logic as the mono path (decim4) to prepare the stereo channel for the last stage. A stereo combiner then brings together the mono and stereo signals, creating distinct left and right audio channels. The output of this combiner gives us the completed stereo audio at a sample rate corresponding to the mode of operation, ready for playback.

Validation Strategy

A critical aspect of our project was ensuring the accuracy and reliability of our implementation. Our validation strategy involved a blend of modeling, iterative testing, and direct comparison with theoretical models.

The first step of validation was the modeling of filters and other key components in Python. This approach provided a clear visual reference, allowing us to compare our C++ implementation against reliable Python models. We extensively used matplotlib to plot our C++ vectors and compared them with Python-generated graphs. This method was particularly effective in validating our filter designs, ensuring that the low-pass filter mimicked the ideal sinc function and that the bandpass filter exhibited the characteristics of subtracting two low-pass filters.

Another cornerstone of our validation strategy was ensuring the accuracy of our block processing logic. For this, we utilized the custom Python script, blockprocessing.py, as a comparative tool. This script replicated the block processing steps we implemented in C++. By running parallel tests and comparing outputs, we could confidently validate the integrity of our block processing in the real-time environment of our SDR system.

Our approach to testing was iterative, with continuous cycles of development, testing, and refinement. This allowed us to quickly identify and fix conceptual problems, such as incorrect filter settings, and implementation-specific errors like excessively clearing arrays in C++, and incorrect state saving logic. By addressing issues in small increments, we maintained a high level of control over the project's progress.

During this project, we faced many challenging debugging scenarios, each requiring a unique approach to resolve. A particularly perplexing issue was the I/Q data blocks that were exponentially growing in size with each iteration. As we processed each block, the time for processing became longer, indicating a problem in the data handling logic. Upon a closer examination, it was discovered that the loop responsible for splitting the incoming I/Q data was incorrectly structured, leading to an exponential increase in vector sizes. By analyzing the sizes of each vector after every block processing and comparing these values, we identified the issue. The solution was straightforward but crucial; a reconfiguration of the for loop logic ensured that the I/Q data was split accurately, preventing the vectors from growing.

Another issue that emerged was the occurrence of underruns, where our processing speed could not keep up with the real-time data stream. To tackle this, we used the chrono library to benchmark our functions, identifying the ‘decim2’ function as the main reason. This function was taking a large amount of time, leading to gaps in the continuous flow of data. We came across the underrun issue once more, when the compiler flag was changed to access GDB. After use, this flag was not changed back to O3, and was pushed to GitHub as o3. Although this is quite a silly issue, it took significant time to resolve.

Furthermore, we dealt with bad allocation errors, specifically, memory leaks. This error was a bit trickier to diagnose. We used GDB to see the order of execution of our program. It turned out we were passing incorrect parameters to the ‘decim2’ function during stereo processing. By passing the wrong values, we wrote data out of the bounds of our allocated memory and tried to modify an array that was incorrectly sized, leading to random behavior and crashes.

In summary, our validation strategy encompassed theoretical modeling, practical testing, and iterative refinement. This approach ensured that our final implementation was functional and aligned closely with theoretical expectations, guaranteeing the effectiveness of our software-defined radio system.

Analysis & Measurements

In the RF front end, two parallel paths of in-phase (I) and quadrature (Q) components of the RF signal have varying sampling rates depending on the mode of operation. Each path begins with a LPF, followed by a downsampling process indicated by rf_decim, reducing the rate by a factor of the downsampling rate. The outputs are then fed into an FM Demodulation block, converting the FM signal back to an audio signal.

Mode 0

The number of multiplications/sample for mono can be detailed using the formula below:

((2*IF_Fs*n_taps) + (Audio_Fs*n_taps)) / Audio_Fs = multiplications/sample for mono.

((2x240000x101) + (48000x101)) / 48000 = 1111 multiplications/sample for mono.

The number of multiplications/sample for stereo can be detailed using the formula below:

(2*rf_Fs*n_taps/rf_decim + 2*if_Fs*n_taps + 2*if_Fs*n_taps*upsample/audio_decim)/audio_Fs = multiplications/sample for stereo.

(2*2400000*101/10 + 2*240000*101 + 2*240000*101*1/5)/48000 = 2222 multiplications/sample for stereo.

Mode 1

((2x120000x101) + (40000 x101)) / 40000 = 707 multiplications/sample for mono.

(2*960000*101/8 + 2*120000*101 + 2*120000*101*1/3)/40000 = 1414 multiplications/sample for stereo.

Mode 2

((2x240000x101) + (44100x101)) / 44100 = 1200.32 multiplications/sample for mono.

(2*2400000*101/10 + 2*240000*101 + 2*240000*101*147/800)/44100 = 2400 multiplications/sample for stereo.

Mode 3

((2x360000x101) + (44100x101)) / 44100 = 1749.98 multiplications/sample for mono.

(2*1440000*101/4 + 2*360000*101 + 2*360000*101*49/400)/44100 = 3499.96 multiplications/sample for stereo

Runtime Analysis

Function / Mode	Mode 0 (ms)	Mode 1 (ms)	Mode 2 (ms)	Mode 3 (ms)	Mode 0 (13 taps)	Mode 0 (301 taps)
RF Front End						
LPF (RF FRONT END)	0.095554	0.08624	0.083258	0.086462	0.02674	0.162184
IQ Splitting	0.426718	0.200683	0.530458	0.269719	0.390444	0.511072
Decim2 (I)	5.20428	8.48467	8.9756	10.5068	0.54861	14.6257
Decim2 (Q)	4.59169	8.50764	5.02099	9.59031	0.601758	13.9016
FM Demod	0.149165	0.348867	0.048259	0.086129	0.045629	0.042815
Mono Path						
LPF Mono	0.045648	0.04061	0.039759	0.037944	0.007889	0.095426
Decim4 (Mono)	0.81138	0.899621	0.874177	1.12312	0.092037	2.85879
Stereo Path						
LPF Stereo	0.042111	0.039759	0.038241	0.037241	0.006685	0.11287
Decim4 (Stereo)	0.816659	0.899566	0.870917	1.12366	0.094204	2.70658
BPF Extraction	0.071185	0.307942	0.084499	0.082795	0.017074	0.165425
BPF Carrier	0.05511	0.348089	0.055463	0.052647	0.008852	0.138851
Decim2 (Carrier)	4.60194	8.4908	5.02373	9.81655	0.632184	14.1622
PLL	0.988805	2.04985	1.0574	2.00481	1.63966	1.1124
Decim2 (Extraction)	4.60223	3.31	5.01845	9.6385	0.644554	14.4361
Mixer	0.014778	0.100036	0.015388	0.030111	0.021741	0.014925
All Pass Delay	0.00924	0.007204	0.010555	0.017037	0.011871	0.009778
Combiner	0.007407	0.00787	0.007481	0.009648	0.085611	0.008537

In mode 0 of our system, altering the number of filter taps from the standard 101 has notable effects on both the system's runtime performance and the audio output quality. Reducing the filter taps to 13 simplifies the filter, leading to fewer calculations per sample and thus a better runtime on the Raspberry Pi. This reduction can significantly enhance the system's real-time processing capabilities. However, this comes at the cost of audio quality; the broader transition band of a simpler filter may result in a noisier signal, with potential aliasing and a less distinct cutoff, compromising the clarity of the FM broadcast.

Increasing the filter taps to 301 improves the audio quality by better isolating the signal. Yet, this sharper filter requires a greater number of calculations, increasing the potential for underruns and computational demands on the hardware. While the audio might be clearer and more precise with 301 taps, the performance trade-off on lower-spec hardware could be substantial, leading to delays that negatively impact the real-time aspect of the system. Therefore, finding the right balance between the number of filter taps is essential to achieving optimal audio quality without sacrificing efficiency.

Proposal for Improvement

To enhance the functionality of our SDR system, we could introduce a graphical user interface (GUI). This GUI would eliminate the need for users to enter complex commands and would allow us to tune into radio stations with ease. Using a Python framework such as Tkinter, we would design an intuitive interface with buttons for station selection, volume control, and mode switching. Implementation would involve creating event-driven functions that interact with our SDR system, transforming user actions into corresponding system commands. By simplifying user interaction, we would become more efficient and productive, while improving user experience and also allow for non-technical users to interface with the SDR. Another feature we could integrate is an automatic station scanner. This scanner would assess frequencies for signal strength, automating the process of finding active stations. We could use the Fast Fourier Transform (FFT) function to analyze the frequency spectrum in real time, highlighting peaks that represent potential broadcasts. A list of possible stations could then be presented to the user, who could select any to listen to.

Additionally, we could consider optimizing our data structures and memory management practices. By using more efficient data containers that minimize memory allocation, and by carefully managing memory reuse, we can decrease the runtime and make the software more suitable for hardware with limited resources. These improvements include reduced computational complexity and more efficient memory usage. By breaking the convolution into smaller, manageable blocks and reusing memory where possible, we can lower the CPU load and memory, enabling the system to maintain real-time processing speeds even on lower-end hardware platforms.

Project Activity

	Raul	**Ali**	**Zuhaib**	**Khushi**
Week 1	N/A	N/A	N/A	N/A
Week 2	N/A	N/A	N/A	N/A
Week 3	N/A	N/A	N/A	N/A
Week 4	- Read project document	- Read project document	- Read project document	- Read project document
Week 5	- Implement low pass filter function, convolution decimation function - Implement FM demodulation function and writing function - Tuning the Mono portion of the project	- Implement reading function + split I/Q data - Implement FM demodulation function and writing function - Tuning the Mono portion of the project	- Implement low pass filter function, convolution decimation function - Implement FM demodulation function and writing function - Implement faster design combining convolution and decimation -Tuning the Mono portion of the project	- Implement reading function + split I/Q data - Implement FM demodulation function and writing function -Implement faster design combining convolution and decimation -Tuning the Mono portion of the project
Week 6	- Implement all the conditions for mode 0,1,2,3 and declare all the required variables - Implement fast convolution, decimation, and up sampling function - Stereo channel extraction: implement all- purpose filter function	-Implement all the conditions for mode 0,1,2,3 and declare all the required variables - Implement fast convolution, decimation, and up sampling function - Stereo channel extraction: implement all- purpose filter function	- Implement all the conditions for mode 0,1,2,3 and declare all the required variables - Stereo Processing: Implement mixer and stereo combiner function	- Implement all the conditions for mode 0,1,2,3 and declare all the required variables - Stereo Processing: Implement mixer and stereo combiner function
Week 7	- debugging stereo - RDS Carrier Recovery: Implement squaring nonlinearity function	- debugging stereo - RDS Carrier Recovery: Implement squaring nonlinearity function	- debugging stereo - RDS Carrier Recovery: Implement squaring nonlinearity function	- debugging stereo - RDS Carrier Recovery: Implement squaring nonlinearity function
Week 8	- project interview - project report	- project interview - project report	- project interview - project report	- project interview - project report

N/A - Unable to start due to multiple exams and project deadlines for all the group members.

Raul:
One complexity I encountered during this project was getting the mono signal to match the stereo signal. The stereo channel extraction required goes through an additional bandpass filter that the mono does not. It is important to recognize this difference as it causes a delay in the stereo signal. We resolved this issue by delaying the mono signal to compensate for the delay in the stereo signal. This was done by implementing an additional function named 'all_pass_filter.'

Ali:
The development of the decim4 function for our project was a complex task, particularly in efficiently combining downsampling and upsampling to ensure high-quality real-time audio playback. The intricacies of implementing a fast convolution mechanism were a significant challenge. We overcame this by upsampling, downsampling convolution using filter coefficients in one function, as well as state saving to handle negative indices. This achievement was crucial as it directly influenced our ability to process audio signals in real-time.

Zuhaib:
One significant challenge I found was within the PLL implementation in our project. More specifically, involving the phase detector error to synchronize the NCO output with the fluctuating frequency and phase of the incoming signal. I addressed this by implementing 'atan2' to determine phase error between the input and feedback signals. This was essential for maintaining synchronization and accuracy of the demodulation process.

Khushi:
Addressing the complexities surrounding the PLL within our project, one challenge I encountered was managing the PPL's state across processing blocks, which posed a hurdle. We needed a solution to preserve the continuity of phase and frequency adjustments. We tackled this problem by implementing a state vector that kept track of essential PLL parameters at the end of each block, ensuring seamless signal processing across blocks. State saving was crucial in ensuring seamless real-time signal processing.

Conclusion

In conclusion, taking on this project where we built a SDR system was completed by applying our theoretical understanding to practical application. Through the hands-on challenges of integrating software and hardware to capture, demodulate, and process FM signals in real-time, we learned the significance of FIR filters, the complexities of FM signal demodulation, and the crucial need for adaptability in engineering designs. This project not only tested our technical skills but also highlighted the dynamic nature of engineering problem-solving. Working together, we navigated through errors we ran into, adjusted our strategies to meet the project's demands, and experienced firsthand the satisfaction of turning theoretical concepts into a functioning reality. It was a comprehensive lesson in the importance of the depths of digital communications, and the practical application of our studies, preparing us for future challenges in the evolving field of engineering.

References

1. Nicola Nicolici and Kazem Cheshmi, Computer Systems Integration Project at McMaster University, Canada Project document, (all other class notes provided on Avenue)