3DY4 Project Final Report

April 10, 2023

Sava Jankovic- 400292525- jankovs
Lukas Kotuza-Janisch – 400238647- kotuzajl
Parth Kotecha – 400303078- kotechap
Stefan Praniauskas – 400315373 – praniaus

Introduction

The 3DY4 project is focused on developing a software-defined radio (SDR) system capable of real-time reception of FM mono/stereo audio and digital data transmitted via FM broadcast using the radio data system (RDS) protocol. To achieve this, the project will utilize affordable RF hardware such as RF dongles and single-board computers like Raspberry Pi. The primary objective of the project is to consolidate the knowledge gained in electrical and computer engineering and develop a comprehensive understanding of the interconnected concepts and principals involved. The project will involve four different modes of operation, each with specific settings and processing blocks for extracting the mono, stereo, and RDS channels. The software implementation will be optimized for efficient performance on the Raspberry Pi and is expected to run in real-time.

Project Overview

The 3DY4 project is centered around the development of a software-defined radio (SDR) system that aims to enable real-time reception of FM mono/stereo audio and digital data transmitted via FM broadcast using the radio data system (RDS) protocol. This SDR system is designed to utilize affordable RF hardware, such as RF dongles, and single-board computers like Raspberry Pi, making it accessible and cost-effective for a wide range of applications.

The primary objective of the project is to consolidate the knowledge gained in electrical and computer engineering and establish a comprehensive understanding of FM modulation, SDRs, and their interrelated components. This involves delving into the fundamental principles and intricacies of these topics, which are practically connected but may initially seem unrelated. The software implementation of the FM receiver system comprises various essential processing blocks, including FIR filters, FM demodulators, PLLs, and re-samplers. FIR filters are utilized to effectively filter the received signal and eliminate noise and interference, while bandwidth selection through BPF allows efficient filtering of the desired FM signal. FM demodulators are responsible for extracting the audio signal from the FM carrier, and PLLs are employed for frequency synchronization to ensure accurate demodulation. Re-samplers are used to adjust the signal's sampling rate to meet the system's processing requirements, enabling efficient processing on affordable hardware like Raspberry Pi.

An indispensable element of the FM receiver system is the RDS decoder, which employs specialized decoding algorithms to extract RDS data from the FM signal. The RDS data may encompass valuable information such as station identification, song titles, and traffic updates, making it a crucial aspect of the overall system. Signal processing techniques such as convolution and decoding algorithms are used to efficiently extract and interpret the RDS data in real-time. The project also aims to explore optimization techniques such as filter design (LPF, BPF) and signal processing algorithms (convolution, demodulation, PLL) to enhance the performance of the SDR system in terms of signal quality, noise reduction, and frequency synchronization. These techniques will be applied to efficiently process the received FM signal

and extract the desired audio and digital data, including the RDS protocol, in real-time, utilizing the affordable RF hardware and single-board computers like Raspberry Pi. Through these efforts, the project endeavors to develop an efficient and affordable solution for real-time reception of FM audio and digital data, including the RDS protocol, while deepening the understanding of these concepts and principles in the field of electrical and computer engineering.

Implementation Details

Labs

At a high level, the labs served as an imitation of the overall project. For example, in lab 1 python was used to offer an introduction to fundamental signal processing techniques. In lab 2, similar techniques were applied in C++ to achieve a more efficient implementation. In lab 3, the mono path of the project was completed in python, as another preliminary step, using the progress from the previous two labs, to lead into the final project.

Lab 1

The first development step involved prototyping core functions in Python. In the fundamental signal processing techniques implemented in lab 1, the block processing was most important to the final project as the underlying logic of block convolution remained constant from lab 1 to the final project. This logic was not implemented correctly in lab 1 and was later corrected during the debugging of Mono in the final project.

Lab 2

The optimization done in lab 2 of moving from python to C++ mainly concerned the block convolution outlined in lab 1. The framework of this function was initially used in the C++ implementation of the final project. This optimization process was informative in how to maintain the logic of an algorithm, while altering the method of implementation to yield efficiency benefits, a step that would be repeated in later parts of the development process. In addition, the manual implementation of the low-pass filter was completed. This function followed the algorithm presented in lecture of generating coefficients for a sinc function, which represents a step function in the frequency domain.

Lab 3

Lab 3 was both a consolidation and prototyping step, as it brought together concepts from the past two labs to complete the mono path in python, though it would later have to be translated into C++. In this lab, debugging had to be done at each processing step in order to obtain the correct output. The main new processing step in this lab involved the demodulation step, which required research into the efficient method which utilized derivatives instead of the computationally taxing alternative which used arctan.

Project

RF Front End

The prototyping of RF front end began with the labs, where key functions such as convolution and inpulseResponseLPF were written in python. Lab 3 consolidated this knowledge into a functional python program, which was then ported to C++ for the project. Because the RF front end processing is depended upon for all the following paths, it had to b e debugged carefully before any further progress could be made. Debugging this part of the code was difficult because when initially developing the mono path, there was no working product to test the code against. Ideally, Lab 3 would have served this purpose, but the project had various bugs which prevented it from exporting audio files. Due to this issue, this step was skipped in favour of immediately integrating the code with the dongle and APlay, which was a relatively simple task, as the Producer Consumer Sample Code on A2L was comprehensive. The main debugging tool that was used in this step was printing values from the project and qualitatively comparing them with the values printed in lab 3. One of the main optimization steps taken at this stage involved downsampling. Initially, the full signal was convolved and then the downsample values were removed. In order to avoid calculating the values that were to be discarded immediately after their computation, the convolution pointer was incremented such that these unnecessary values were skipped. One of the main issues in this stage was finding vector functions that would not cause problems during the runtime of the program. Initially, the assign function was used, but this would overflow the memory and crash the program. It was later determined that the resize function was a safe alternative.

Mono

The mono path reused most of its functions from RF front end, making implementation simple. The main feature that mono added was upsampling. The initial method of upsampling was just adding zeros, then passing the signal through a LPF scaled by the upsample coefficient to interpolate the data. This method was inefficient as it computed many multiplications by zero, which were unnecessary. In order to optimize this step, the convolution counters were looped in such a way that these repetitive zero calculations were skipped. This step was essential in getting the program to the efficiency required to process radio signals in real time. The main error when optimizing convolution was segmentation error due to array out of bounds. The main debugging technique used to solve this issue was printing out the index until the program crashed, allowing the behaviour of the indices to be tracked and therefore more easily corrected.

Stereo

The stereo path included all the steps of mono, with some added complexities. Due to time constraints, this path was implemented directly in C++ with few iterations. This made debugging more difficult, but did save time overall, due to the reductions of steps required. It is also worth noting that this approach was made possible by the fact that much of the new code in stereo was either provided on A2L or outlined in great detail in the lecture slides. The PLL code was ported from the provided python implementation, with the main added feature being

the state saving. It was then discovered that a common method of saving multiple variables was to use struct. This circumvented the necessity of passing each state variable as an argument to the PLL function in favour of simply passing one struct variable that contained all the required states. The mixing and stereo combining steps were implemented using the transform function from the C++ std library, allowing for a straightforward approach to the arithmetic required by these steps.

RDS

The RDS path was partially completed using much of the reused code from the previous sections. There was not much debugging done on the RDS section, as the code written for it was focused on sketching an outline of its the main steps as opposed to diving too far into specific implementation details. As for the iteration stages, RDS was approached similar to stereo, where the final product was targeted from the start, foregoing extensive iteration. Development began with studying lecture material and determining what could be reused from stereo and what required further research.

The steps for RDS channel extraction and carrier recovery were repurposed from stereo and were implemented without much trouble. The main addition in this step was the addition of the APF, as unlike in stereo, the RDS channel was not filtered alongside the pilot frequency. The PLL and mixing steps were also implemented similarly to stereo with some parameter changes in PLL as outlined in the lecture slides. The squaring nonlinearity was achieved using the transform function, similar to the mixing step.

The CDR function remained in the conceptual design stage, as there was not sufficient time to deeply explore aspects such as the python plots, though an outline of its function was designed based on the topics mentioned in lecture such as implementing s synchronization anchor with the local min, max values.

The main debugging tool used for RDS was verifying the logic of the code with the lecture material to ensure that conceptually, the various blocks of RDS were correctly implemented, thus providing a strong base which could be debugged more thoroughly should further development time be available.

Multithreading

The multithreading was implemented following example105 from the Producer Consumer Sample Code provided on A2L. This approach was selected for its reliability and simplicity, as it did not require the use of mutex. Instead, the queue would be indexed with atomic variables, which provided well-defined read/write access from different threads. The indexing used specific offsets for each thread which ensured that no data would be overwritten by RF Front End before it was processed, and no data would be skipped by the Audio and RDS threads before being processed. Though the sample code provided all the necessary logic to integrate multithreading, further research was done concerning the details of certain functions from the atomic class using the C++ documentation to consolidate understanding of this step.

Analysis and Measurements

We were unable to complete RDS, and thus we do not have an executable thread for RDS. As a result, Run Times and multiplications and accumulations per one bit for RDS were not able to be calculated.

Table 1: Analysis of Multiplications and Accumulations per Sample:

Path and Mode of Operation	Multiplications and Accumulations per Sample	Input (assume 100,000)	rf_decim	expander	audio_decim	audio taps
Mono, 0	1111.00	100000	10	1	5	**101**
Mono, 1	1111.00	100000	6	1	5	101
Mono, 2	1200.32	100000	10	147	800	101
Mono, 3	1420.18	100000	4	49	320	101
Stereo, 0	2121.00	100000	10	1	5	**101**
Stereo, 1	2121.00	100000	6	1	5	101
Stereo, 2	2299.64	100000	10	147	800	101
Stereo, 3	2739.37	100000	4	49	320	101
----------	----------	----------	----------	----------	----------	----------
Mono, 0	143.00	100000	10	1	5	**13**
Stereo, 0	273.00	100000	10	1	5	**13**
----------	----------	----------	----------	----------	----------	----------
Mono, 0	3311.00	100000	10	1	5	**301**
Stereo, 0	6321.00	100000	10	1	5	**301**

Table 2: Non-Linear Operations:

Path and Mode of Operation	Non-Linear Operations
Mono, 0	0
Mono, 1	0
Mono, 2	0
Mono, 3	0
Stereo, 0	20
Stereo, 1	20
Stereo, 2	21.7687075
Stereo, 3	26.122449

As we expected from the multiplications and accumulations, generally there is not a noticeable change between modes 0 and 1, but much more apparent change in modes 2 and 3.

*Table 3: Run Times for all functions (Mono and Stereo Paths, All Modes):
*Note: Modes 0 and 2 have an input IQ rate of 2.4M/sec, Mode 1 has an input rate of 1.44M/sec and Mode 3 has an input rate of 1.152M/sec. **Ran with Taps = 101**.

Function	Path	Runtime Of Mode 0 (ns)	Runtime Of Mode 1 (ns)	Runtime Of Mode 2 (ns)	Runtime Of Mode 3 (ns)
Front End LPF Coeff	Mono	29742.64356	25813.76	32140.91176	42032.9009
Front End LPF Coeff	Stereo	28660.5514	26690.72816	44879.35644	39707.52427
Front End Convolve	Mono	3625154.534	4188530.704	7575726.223	210043310.6
Front End Convolve	Stereo	3721414.912	3780724.505	524008717.4	210970591.4
Front End Demod	Mono	87016.58491	86757.85849	182034.1058	4537548
Front End Demod	Stereo	91334.22642	92483.30841	11122074.46	4537611.971
Audio LPF Coeff	Mono	152609	26184	3004102	609920
Carrier BPF Coeff	Stereo	73315	75536	77203	39592
BPF Coeff	Stereo	54166	77185	22981	41036
LPF Coeff	Stereo	40611	40444	1914257	622400
Convolve	Mono	384420.0885	366435.6786	8494026.194	274185946.6
Carrier Convolve	Stereo	1311441.943	1395096.887	189383168.1	77752259.29
Queue	Mono	28013843.45	28520029.65	41475685.98	869272981.6
PLL	Stereo	605522.0769	686439.4563	97001232.08	39668262.39
Extraction Convolve	Stereo	1378799.539	1301795.282	190018023.3	76240517.46
Mixing	Stereo	67492.95146	55353.68269	15394098.36	6012637.919
Final Filter/ Conversion	Stereo	439090.6408	379054.1765	837137530.6	266954698.9
Combiner	Stereo	16574.02941	11112.15238	1111708.693	345233.2315
Front End Queue	Mono	19777.59048	15154.98077	28840.2451	751088.1584
Front End Queue	Stereo	16749.04762	17678.71429	1822393.126	763096.9

*Table 4: Effect of Num Taps:
*Note: All tests below were conducted in Mode 0, as instructed.

Function	Path	Runtime With Taps = 13 (ns)	Runtime With Taps = 101 (ns)	Runtime With Taps = 301 (ns)	Change (101 to 13, 101 to 301) Positive Means larger taps is slower
Front End LPF Coeff	Mono	12978.75238	29742.64356	63738.0283	**16763.89118, 33995.38474**
Front End LPF Coeff	Stereo	11871.33654	28660.5514	60938.94286	**16789.21486, 32278.39146**
Front End Convolve	Mono	577541.8846	3625154.534	10051171.85	**3047612.6494, 6426017.316**
Front End Convolve	Stereo	547009.7019	3721414.912	10278900.93	**3174405.2101, 6557486.018**
Front End Demod	Mono	80596.64151	87016.58491	85245.81731	*6419.94339566038, -1770.76759566038*
Front End Demod	Stereo	84941.93269	91334.22642	83774.00952	*6392.29372509434, -7560.21689509433*
Audio LPF Coeff	Mono	6056	152609	104147	146553, -48462
Carrier BPF Coeff	Stereo	20259	73315	118017	53056, 44702
BPF Coeff	Stereo	11296	54166	144813	42870, 90647
LPF Coeff	Stereo	8944	40611	102203	31667, 61592
Convolve	Mono	53475.2243	384420.0885	1029915.519	**330944.8642, 645495.4305**
Carrier Convolve	Stereo	208024.2056	1311441.943	3571814.886	**1103417.7374, 2260372.943**
Queue	Mono	28372172.5	28013843.45	27191390.1	*-358329.05, -822453.35*
PLL	Stereo	652574.9151	605522.0769	613565.5094	*-47052.8382, 8043.4325*
Extraction Convolve	Stereo	208746.8679	1378799.539	3513207.953	**1170052.6711, 2134408.414**
Mixing	Stereo	57963.65094	67492.95146	64755.2243	*9529.30052, -2737.72716*
Final Filter/ Conversion	Stereo	52852.7757	439090.6408	1016570.876	**386237.8651, 577480.2352**
Combiner	Stereo	15416.04762	16574.02941	14428.63462	*1157.98179, -2145.39479*
Front End Queue	Mono	16148.16346	19777.59048	18461.38462	*3629.42702, -1316.20586*
Front End Queue	Stereo	16121.88462	16749.04762	18617.30769	*627.163, 1868.26007*

When varying the taps, the run times get proportionally larger with what was identified for the multiplications and accumulations for the functions we expect (see table 1). Specifically noting the massive impact taps have on convolution specifically. Additionally, see the “Change” column in table 4 for more in detail on which functions are affected by taps change. Bolded items in the column are functions that were direly affected by an increase in taps, contents that are italicized saw a negligible change (change was most likely measurement error and caused by not using a large enough sample size for testing).

Proposal for Improvements

Although there were many concepts throughout the project that were done well, which led to a successful product, there was still room for improvement. The following points outline the main improvements that could be done to improve the project flow next time:

First of all, we could’ve implemented a good model through python code. Although the product did contain strong C++ code for all parts of the project, there is not a lot of Python code

accompanying it. From the beginning of the course, python was essentially used as a modelling tool that would show us how the outputs would look graphically to the user. Without python modelling for most of the project, the user wouldn't be able to have a clear idea of what each part is effectively doing. Another improvement that could be added to the project which would decrease the running time is using a different technique for the phase locked loop block incorporated in the stereo part of the project. We have used a built in C++ arctan estimation function, which outputted the expected results, but the running time was considerably larger than expected. The improvement would be to instead implement a custom Arctan function that would be more time efficient. When we did this throughout the first three labs, it proved to be a very useful function and was time efficient.

Another improvement that could be made is to work on GitHub branches. Throughout the project, there were a few instances where the local code that we have each written for the functions in Mono and Stereo did not match the ones on the brain branch. This was mainly due to different teammates pushing their own versions of the code that would overwrite the previous progress. By using the GitHub branches, everyone would be able to work on their own part of the code and therefore in the end would be able to come along and combine as well as debug the code efficiently by merging the branches.

Project Activity

Week	Progress	Contribution
February 14 – February 21	- Reading and understanding project document - Going over lectures and labs for filtering, convolution, and demodulation	All members discussed project specifications, delegated major work areas, studied respective project topics
February 21 – February 28	- Adding functions for convolution and low pass filters - Added RF front end function - Added demodulation	Stefan and Lukas begin implementation. All members continue to study respective content
February 28 – March 7	- Finalizing mono - Adding all modes functionality to mode.cpp and mode.h - Starting stereo, created BPF - Started PLL	Lukas and Stefan work on C++ implementation. Sava and Parth work on modelling in Python

	- Testing Mono with front end	
March 7 – March 14	- Finished PLL - Added mixing and channel combination - Started threading support - Finished APF	Stefan and Lukas continue with implementation and integration of mono and modes. Sava and Parth design supporting C++ functions for stereo
March 14 – March 21	- Finalizing stereo - Finalized Multithreading - Tested stereo and real time audio	Lukas and Stefan finish multithreading and stereo. Parth and Sava finished supporting functions
March 28 – April 4	- Last edits to stereo - Presentation preparation - Cross exam preparation - Continuing working on RDS functions - Added support to limit user to modes 0 and 2 for RDS	All members presented in class and attended cross examination. Specific RDS functions were delegated to group members
April 4 – April 10	- Report writing	Work divided between all members

Conclusion

Overall, the 3DY4 project had been an experience that not only taught us about implementation of concepts such as Fourier Transform, Convolution among other things, but also helped us organize ourselves as a team and work with each other on creating the best output possible. Implementation of FT, Convolution, Band Pass Filters, PLL's had been given to us in other courses, however this project taught us the principle of representing it in C++ code and henceforth putting Stereo, Mono and RDS blocks together. Throughout the project, we have also developed our coding skills through libraries such as SciPy and NumPy. These have helped us not only write different code, but also visualize it through signal plotting. Finally, the way that the project had been broken down into different sections made it easier for us to complete as we were able to divide each other efficiently and henceforth finished almost all the tasks on time.

References

- https://en.cppreference.com/w/
- https://cplusplus.com/doc/tutorial/
- 3.11.3 Documentation (python.org)
- Avenue Lecture Slides
- Relevant Provided C++ and Python Code Files
- https://stackoverflow.com/questions/876901/calculating-execution-time-in-c
- c++ - Elegant way to pass multiple arguments to a function - Stack Overflow
- Previous Lab code