COE3DY4 Project Report

Group 07

Caleb Gannon, Ben Stephens, John Wyeth, Sawoud Al-Los

gannonc, stephb5, wyethj, alloss

April 9, 2021

1 Introduction

The objective of this project is to create a radio data decoder system that receives data via a dongle, and passes it on to a C++ application. The relevant channels for mono, stereo, and RDS get extracted and processed. The live data is played, as the new data is coming in. In the list below some explanation will be provided into the related concepts required for the understanding of this project:

- Software-defined radio: takes the components that are implemented in a traditional radio's hardware and implements them in software.
- Frequency Modulation: a modulation technique that relies on modifying a carrier wave's differential phase (frequency) based on the amplitude of the message signal.
- Block processing: taking a long stream of data and processing it in smaller defined chunks.
- Finite impulse response filters: are filters that only allow certain frequencies in a signal to pass via impulse vector and convolution.
- Phase locked loops: generates output whose phase is the same as the inputs.
- Re-samplers: uses intelligent sample spacing to perform the optimal number of convolution operations while changing the sampling rate by a rational amount.

2 Project overview

The project is a minimum-viable-product approach to software defined radio. The Raspberry Pi 4 Model B handles the majority of compute operations, and is fed I and Q (in-phase and quadrature) RF samples by the NESDR RTL-SDR (Realtek RTL2832U chipset) RF dongle. The project data-flow encompasses four major sub-sections: front-end processing, mono audio, stereo audio, and RDS data recovery.

Front-end processing receives I and Q samples from the RF hardware. The role of this module in the signal-flow process is to extract the IF (intermediate frequency) FM channel. It works by demodulating the I and Q samples using their phase differential. To reduce required operations, the sample rate is decimated by a factor of 10. The input rate is either 2.4MSamples/sec or 2.5MSamples/sec. The IF band spans only 100KHz, and in account with the Nyquist theorem a sampling rate of only 200KHz is required (to recover RDS a squaring non-linearity rate of 114KHz is required, and as such a Nyquist rate of 228KHz is encompassed by both 240KHz and 250KHz). Anti-aliasing filters are used before the decimators, with a cutoff of 100KHz.

Mono audio is outputted at 48KSamples/sec, and extracts the mono-audio channel from the IF spectrum (0 - 15KHz). This pathway operates in two different modes: mode 0 corresponds to an IF frequency of 240KHz, mode 1 corresponds to 250KHz. The difference between the mode processing-paths lies in the requirement of a resampler for mode 1, whereas mode 0 requires only a decimator. Mode 0 receives samples at 240KSamples/sec, and therefore a decimator of factor 5 brings it to 48KSamples/sec. Mode 1 is more complicated: using a rational resampler, the IF up-samples to 6MSamples/sec (factor of 24), and then down-samples by a factor of 125 to 48KSamples/sec. The resampler is used to optimize this up/downsampling data-flow. Both paths require anti-aliasing filtering, however the resampler requires additional anti-imaging filtering due to upsampling.

Stereo audio also receives IF samples in either mode 0 or mode 1 sample-rates. This processing path outputs 48KSamples/sec, but combines a separate IF channel with the above extracted mono data. The separate channel contains the difference between the left and right audio channels, and spans 23KHz - 53KHz. Extraction of this channel begins by locking a PLL (Phase-Locked-Loop) onto the pilot tone located at 19KHz. Bandpass filtering extracts the pilot tone, and the PLL output is modified by an NCO (Numerically-Controlled-Oscillator) to produce a phase locked sinusoid at 38KHz. This is modulated (mixed) with the bandpass-extracted stereo channel, and the output is processed similarly to the mono path. The resulting 48KSamples/sec output is combined with the Mono data to extract the left and right audio channels.

RDS processing recovers binary data containing information about songs playing, and the radio station. The 19KHz tone is not always phase-locked to the RDS channel, therefore additional processing is required. The RDS channel is bandpass-extracted, and pointwise squared to recover a phase locked signal at 114KHz. This signal is used to lock the PLL, and the NCO outputs a locked sinusoid at 57KHz. This output is mixed with the RDS channel, the result is low-pass filtered at 3KHz (RDS is transmitted at 2375 symbols/sec). The output is rationally resampled to a multiple of 2375Hz, and then passed through a root-raised cosine filter to retrieve the RDS symbols. The symbols are then processed in order to synchronize to the transmitted clock rate. The symbols come in *High-Low* or *Low-High* pairs, representing binary values. These binary values are differentially encoded. Once decoded, the binary data can be error-checked and clock-synchronized by detecting transmitted syndrome codes via the Galois field matrix multiplication of 26-bit strings and a parity array.

3 Implementation details

Labs

The labs were concerned with the early tools required to begin working towards a full software model. We build a convolution algorithm first in Python, then in C++. This algorithm would act on FIR coefficients produced by our implemented FIR algorithm. We worked with models of the SciPy library, and explored the creating of both low-pass and band-pass filter coefficients.

We formed a foundational understanding of block processing, and how in reality a single-pass approach to convolution is impractical. So we built block-processing algorithms first in Python, then C++, relying on state saving methods to eliminate block-edge-case errors. These models were tested on RAW audio files in both Python and C++, and were pivotal in building the Mono path.

Mono

The Mono implementation was initially inspired heavily by the Lab 2 C++ block processing approach. Python modelling was handled in depth throughout the labs. It was optimized by merging the decimation and anti-aliasing filter operations, via a combined method of nomenclature *convolveFIR_N_dec.* This was accomplished by only calculating every Nth output value, so as not to waste time anti-aliasing values the decimator would throw away.

This pathway appeared simple, however it took some time to debug several critical errors:

- Some array sizes had been set incorrectly and the data became mismatched
- The block-selection algorithm used to grab I and Q samples was grabbing double the block size
- The *fmDemodArctanBlock* method was not correctly state-saving, and did not account for NaN

Stereo

By this section, we had implemented command-line-arguments (CLA) to distinguish calls for mode 0/1, the mono/stereo path, and to distinguish between binary and float32 input types. Python modelling of stereo was an addition to the Mono modelling from lab 3. Bandpass filters were derived using Scipy methods, and the PLL method distributed by Professor Nicolici was implemented in a black-box setting. Most troubleshooting involved PLL state saving and phase. The largest bug in modeling was the SciPy argument pass_zero=False being required to omit the zero-frequency response. Once SciPy was correctly implemented, stereo audio was recovered. The conversion of the stereo model to C++ was very straightforward.

Most issues were around troubleshooting, such as:

- ➢ Matching both audio and stereo data to *short int*
- ➢ Incorrectly initializing the band-pass coefficients

These issues were resolved by writing blocks of data to .dat files, and comparing to .out files from the Python model either visually or numerically.

Mode 1

The most important aspect of getting the Mono mode 1 audio path working was integrating the upsampling into the convolution and decimation function used in Mono mode 0. Through an optimized method that takes advantage of the upsampling, we run through a method that performs the convolution on only specific indices of arrays. Running through N phases, where N is the upsampling value, we increment the indices of the impulse and data array by the lowest common divisor between the upsampling size and the decimation value.

Errors that arose during the debug process came from how we iterated through all the N phases. Since the phases repeat multiple times throughout the array indices, we decided to complete all of each phase before moving on. This led to errors with iterating array indices based on the phase, more specifically what the starting index for the impulse array was and adding N each iteration. Debugging our implementation consisted of carefully tracking how we iterate each phase, printing how the for loops changed the indices was our main tactic when troubleshooting.

Threading

Threading was accomplished in three phases:

- ➢ **Phase 1:** Front-end ↓ (reader/writer)
  - ○ ↑ Audio
- ➢ **Phase 2:** Front-end ↓ (reader/ 2x. writer)
  - ○ ↑ Audio
  - ○ ↑ RDS
- ➢ **Phase 3:** Front-end ↓ (reader/ 2x. writer)
  - ○ ↑ Audio
  - ○ ↑ RDS_Binary ↓ (reader/writer)
    - ■ ↑ RDS_Sync

Initial front-end to audio threading was accomplished using the standard mutex reader/writer approach covered in classes for a single queue. It was an effective method, however the overhead of threading required a slight decrease in the number of front-end taps.

RDS + Audio threading encompassed a few trials. Initial whiteboarding sessions lead to a possible single-queue semaphore based implementation. The C++ standard library semaphore class was tested, along with additional other mutex-based semaphore implementations, however none seemed to work according to our hypothetical design. After multiple semaphore implementations failed, we switched to a multiple-queue mutex-based implementation.

This design was met with initial issues, often ending in deadlock. The GDB debugger was useful in identifying the fundamental issues in this design: when one reader-queue filled, the reader would lock prematurely and never reach the notify stage so the system would deadlock. We modified our approach using notify_all and achieved a working multi-queue design.

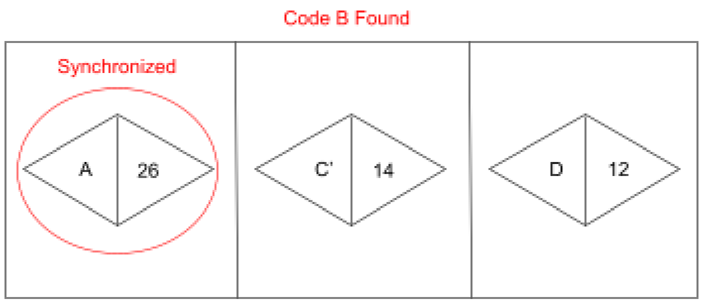
To split the RDS path into two threads, the pre-differential binary data was appended to a third queue, which a secondary RDS thread would process to detect semaphores. This path used a standard mutex-based reader writer approach based on the front-end to audio method.

RDS

RDS was modelled in Python, appending onto the existing Stereo block-processing model. The model initially was a straightforward approach, following the standard directive:

- ➢ Extraction of the RDS channel, pointwise squaring of the array, extraction of the RDS carrier, PLL NCO output of a 57KHz phase locked sinusoid, and mixing to bring the RDS channel to the origin.
    - ○ The PLL was phase shifted appropriately by plotting IQ constellations
- ➢ Output is filtered at 3KHz, inefficiently upsampled by 19, filtered for alias+imaging, and downsampled by 80.
    - ○ This results in a sample frequency of 57KHz
- ➢ Root-Raised Cosine coefficients are generated and convolved with the output.
- ➢ A zero-finding algorithm is used to select the initial offset when sampling symbol arrays of size 24. The algorithm sets the initial offset, but the difference between the last sample and the end of the block is saved and used to compute the offset in the next block.
    - ○ This maintains a constant separation of 24 samples between sampling points.
    - ○ The zero-finding algorithm is set to re-run in the case of excessive errors. This can compensate for selecting a new station mid-run, or errors in the first block due to PLL locking-time.
- ➢ As our chosen block size is 5120 *fm_demod* values. This results in 1216 samples of RDS data (after resampling). *1216 / 24 = 50 + ⅔*. This results in incomplete symbol 24-groups, unless state saving is implemented.
- ➢ We noticed that to avoid saving entire 24-groups of RDS data, we would take advantage of the sampling pattern and save only a single sampled value when necessary.
- ➢ The pattern that results from the above mathematical relationship is that every three blocks will contain two sample sets of size 51, and one set of size 50. The configuration depends on the initial sampling point.
- ➢ Two samples are required to detect a bit, so sets of size 51 will have a spare-sample that must be saved.
    - ○ Unless, there is already a spare-sample available for use (*52 / 2 = 26*)
- ➢ State saving logic is used to detect the availability of a spare-sample, and the output of this logic is a binary array of size either 25 or 26.
    - ○ The logic also contains an error-detection algorithm, in the case of a High-High or a Low-Low detection, an error is flagged so the system can respond.
    - ○ Depending on error tolerance settings, the error either triggers a reset-response that offsets the sampling pattern by one symbol, or a differential algorithm is used to compensate for a noisy/weak signal and estimate the transmission for a pre-set allotment of erroneous symbols.

- ➢ The binary array is then processed by a manchester-differential algorithm, and each resultant bit is fed to a 26-bit-running-queue-like array.
- ➢ The running array is processed on each iteration via Galois-field-multiplication with a parity array.
- ➢ The resultant 10-bit-syndrome output is compared against syndrome results for transmitter offset codes.
- ➢ A global array *found_array* is used to track correct syndrome code detection. The array is a 2-dimensional-int-int array. Each sub array contains 2 elements: One is the ASCII of the detected syndrome, the other is the number of iterations since detection. Each detection of a syndrome is appended to *found_array*, and on each iteration the array is checked to see a sub-array counter has reached 26 iterations, in which case it is deleted from the *found_array*. However, if the correct subsequent code is detected when a sub-array counter is 26, the system is declared synchronized.



- ➢ Once the system is synchronized, it checks the syndrome only every 26 bits to detect errors and maintain synchronization.
- ➢ However after a given error-tolerance of consecutive missed syndromes, the system decides it has lost synchronization and resets—checking every new bit against a running 26-bit total to establish resynchronization.

The most time consuming aspect of the RDS path was integration of each of these modules in C++ and troubleshooting the resultant deviations from the Python model. There were a few major bugs:

- ➢ The resampler used in the audio processing path had to be remodelled to work effectively in the RDS path.
- ➢ Threading posed significant challenges across different implementations of RDS.
- ➢ Error thresholds had to be tuned to compensate for the lower quality output of *fm_demod* in comparison with the Python model (w/o atan derivative approximation)

Note when testing: Our code is set to print detected syndromes only after it has synchronized. To see all detected syndromes, uncomment lines 332, 350, 368, 386, and 404 of filterRDS.cpp.

Debugging

To debug the final product, the GDB debugger was used, alongside the debugging strategies mentioned in the respective sections above.

4 Analysis and measurements

Blocksize = 102400 = BSZ | i_samples size = Blocksize / 2

Front End Path

2x Convolution/decimation + 1x Arctan Demod

- ❑ 2 * (Taps$_{FE}$ * *BSZ* / (2*decimation)) + *BSZ* / (2*decimation)
- ❑ 2 * (Taps$_{FE}$ * *BSZ* / 20) + *BSZ* / 20 = **10240*Taps$_{FE}$ + 5120 operations**

Mono 0 sample

1x Convolution/decimation

- ❑ Taps$_{Audio}$ * *BSZ* / (20 * decimation)
- ❑ Taps$_{Audio}$ * *BSZ* / (100) = **1024*Taps$_{Audio}$ operations**

Therefore a Mono 0 sample requires 10240*Taps$_{FE}$ + 5120 + 1024*Taps$_{Audio}$ operations.

Stereo 0 sample

2x Convolution + 1x PLL + 1x Pointwise + 1x Convolution/decimation + 2x Pointwise

- ❑ (2 * Taps$_{Audio}$ * BSZ / 20) + (BSZ / 20) + (BSZ / 20) + (Taps$_{Audio}$ * *BSZ* / (20 * 5)) + (2 **BSZ* / (20 * 5))
- ❑ *Above* = **11264*Taps$_{Audio}$ + 12288 operations**

Therefore a Stereo 0 Sample requires 10240*Taps$_{FE}$ + 12288*Taps$_{Audio}$ + 17408 operations.

Mono 1 sample

1x Resample

- ❑ Taps$_{Audio}$ * (24/125) * *BSZ* / 20
- ❑ *Above* = **984*Taps$_{Audio}$ operations**

Therefore a Mono 1 sample requires 10240*Taps$_{FE}$ + 5120 + 984*Taps$_{Audio}$ operations.

Stereo 1 sample

2x Convolution + 1x PLL + 1x Pointwise + 1x Resample + 2x Pointwise

- ❑ (2 * Taps$_{Audio}$ * BSZ / 20) + (BSZ / 20) + (BSZ / 20) + (Taps$_{Audio}$ * (24/125) * *BSZ* / 20) + (2 * (24/125) * *BSZ* / 20)
- ❑ *Above* = **11224*Taps$_{Audio}$ + 12208 operations**

Therefore a Stereo 1 Sample requires 10240*Taps$_{FE}$ + 12208*Taps$_{Audio}$ + 17382 operations.

RDS bit (Mode 0)

1x Convolution + 1x Pointwise + 1x Convolution + 1x PLL + 1x Pointwise + 1x Resample + 1x Convolution + 1x Pointwise (sampling) + 1x Pointwise(symbol analysis) + 1x Pointwise (manchester)

- ❑ (2 * Taps$_{RDS}$ * BSZ / 20) + (3 * BSZ / 20) + (Taps$_{RDS}$ * (19/80) * BSZ / 20) + ((19/80) * BSZ / 20) + (2 * (19/80) * (1/24) * BSZ / 20)
- ❑ *Above* = **11456*Taps$_{RDS}$ + 16677 + ⅓ .**

The additional ⅓ is on account of the fact that bits are calculated on a rotating state-saving pattern of 25-25-26-groups.

Therefore an RDS bit requires 10240*Taps$_{FE}$ + 5120*Taps$_{RDS}$ + 28133 + ⅓ operations.

Runtime Analysis

Code Section	Runtime**		%diff.* (Time)		%diff.* (Analytic)	
Front End Path 0|1	3.95272 ms	4.10945 ms				
Mono 0	0.363866 ms		+9.21%		+10.0%	
Stereo 0	4.61042 ms		+116.6%		+110.4%	
Mono 1	0.40652 ms		+9.9%		+9.6%	
Stereo 1	4.71014 ms		+114.6%		+110.0%	
RDS 0|1***	4.88981 ms	0.006502 ms	+123.7%	+0.16%	+112.5%	+0.1%

** Percent difference is relative to FE runtime, analytic is computed w/ 151 Taps*

*** Average over 10 Trials w/151 Taps (FE, Audio and RDS).*

**** RDS 0|1 refers to separate RDS threads, not Mode 0 and 1*

Differences between theoretical and measured values can be largely attributed to the way the analytic section does not consider the exact difference between MAC and pointwise operations, as well as the overhead from the large if-conditional segments in the RDS path.

5 Proposal for improvement

The user experience can be optimized; a graphical user interface (GUI) to show the user the name of the radio station, artist, and song can be added. This addition would round off our system to act more like the modern radio found in the majority of vehicles and homes - while opening up the opportunity to change radio stations through the GUI at runtime.

A developer productivity boost can be achieved, making the code base easier to work with by putting each thread in a file of its own (so the experiment.cpp file file only has the logic for the main function). This would make the code overall easier to read, understand and edit. This change is realistic and viable since it only requires the threads to be moved to other files and the make file will just have their names declared in the "SOURCES" variable.

To improve the runtime of the system, even when it is run on hardware of lesser performance, the number of if-statements can be decreased since evaluating a huge amount of them takes significant processor resources. Wherever appropriate, switch statements can be used instead of if-statements since their evaluation time is much lower when considering a large number of conditionals. Also if the hardware has less storage space, the *shrink_to_fit()* command can be applied more often to release memory resources. The number of functions used from the math library can be decreased since these functions are usually slower. For example, an arctan approximation can be implemented in the PLL pathway. Necessary reduction in computation can be offset by applied Look-Up-Tables.

6 Project activity

Week	Overall Progress	Individual Contributions	
Mar 1-Mar 6	Mono Python Model	**John:** Mono Path **Sawoud:** Mono 1	**Caleb:** Mono Path **Ben:** Mono Path
Mar 7-Mar 13	Mono C++ Complete Stereo Model	**John:** Mono Mode 1 **Sawoud:** Mono 1	**Caleb:** Stereo Path **Ben:** Stereo Path
Mar 14-Mar 20	Stereo C++ Complete Mode 1 C++ Complete Threading started, no RDS implementation	**John:** Mode 1 Optimization **Sawoud:** Mode 1 Optimization	**Caleb:** Threading **Ben:** Threading
Mar 21-Mar 27	RDS Threading started RDS Python Completed	**John:** RDS Demodulation Path **Sawoud:** RDS Demod/CDR	**Caleb:** RDS Modelling **Ben:** RDS Modelling
Mar 28-April 3	RDS Threading Main-exec. debugging RDS C++ Complete	**John:** RDS Demod + CDR **Sawoud:** RDS Man. Differential	**Caleb:** RDS C++ **Ben:** RDS C++
April 4-April 10	Project report compilation, code analysis, etc.	**John:** Project Report **Sawoud:** Project Report	**Caleb:** Project Report **Ben:** Project Report

7 Conclusion

Throughout this project period there were two major learning experiences: Interfacing with project specific technologies such as the RTL dongle, and managing the development of a project with this complexity. When it comes to understanding how to work with the data stream the RTL dongle produces, we had to utilize python for its ease of modelling and C++ for its quickness and real time implementation. Following the project document and applying various optimization techniques, the group was able to perform various signal processing techniques (ie. convolution, impulseResponses, upsample/downsample, etc) to manipulate and stream radio station audio in real time.

8 References

https://www.scipy.org/

https://en.cppreference.com/w/

https://www.embedded.com/dsp-tricks-frequency-demodulation-algorithms/

CompEng 3DY4 Lab 1, 2 and 3 Manuals

Dr. Nicola Nicolici and teaching assistants