Paul Adams and Lincoln Young

Project Report

August 16, 2016

Report Drafting

INTRODUCTION

This project began as a conceived innovation to the MIT Coffee Can Radar [1], hereafter referred to as the reference design. The reference design demonstrated three different radar modes - Continuous-Wave (CW) Doppler, Frequency Modulated CW, and crude Synthetica Aperture Radar (SAR) - with minor hardware adjustments and using different processing algorithms. The system was interfaced with a laptop in order to acquire a block of data and then process the data offline and show results in Matlab. A block diagram is shown for reference.

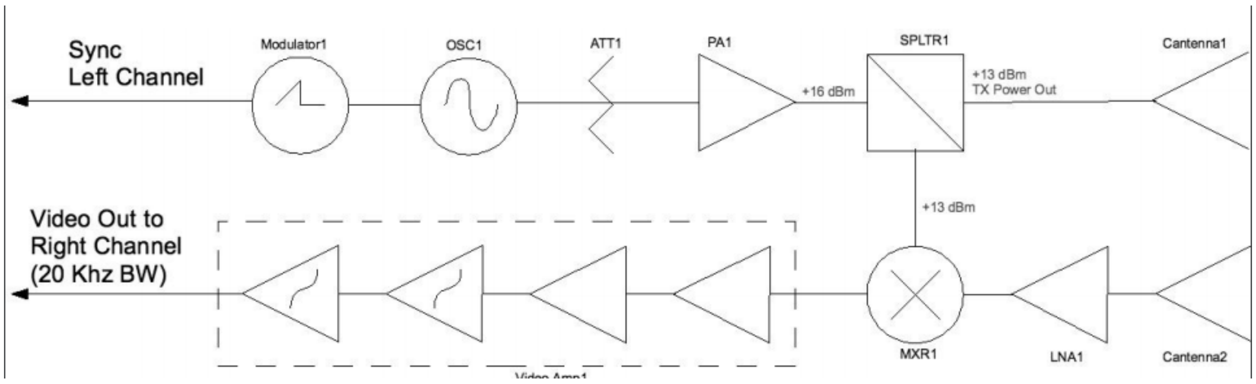

Figure 1: MIT Coffee Can Radar Block Diagram

Our goal was to implement two of these modes using streaming processing and human-in-the-loop mode switching, thus demonstrating a Software-Defined Radar (SDR) system. This project could be extended to interface the control software with waveform tuning hardware and demonstrate Cognitive Radar (CR).

Our initial innovation to the reference design was to design hardware improvements and add a real-time control loop with network offloading of results. The control portion was chosen to be implemented on a Raspberry Pi 2 running Arch Linux for ARM whereas the reference design interfaced with a laptop and used offline batch-mode processing. The modified high-level design is shown below for reference.

Discussion of Project

Design Procedure

The SDR design can be summarized into two components: 1. The design improvements to the reference hardware system and 2. The software design.

Hardware Improvements

Software Design

The reference design provided Matlab scripts for ingesting a block of audio data and then processing to show results. There was no detection or tracking software included in this reference and neither did we attempt to develop sophisticated detection or tracking software. Rather, the data can be transformed and displayed as an image, which then allows the eye to perform the job of a target detector/tracker system.

Initial design began with experimenting and rewriting the reference Matlab code to handle processing the data as a stream and displaying the results in an animated waterfall image. This process was repeated for each of the two radar

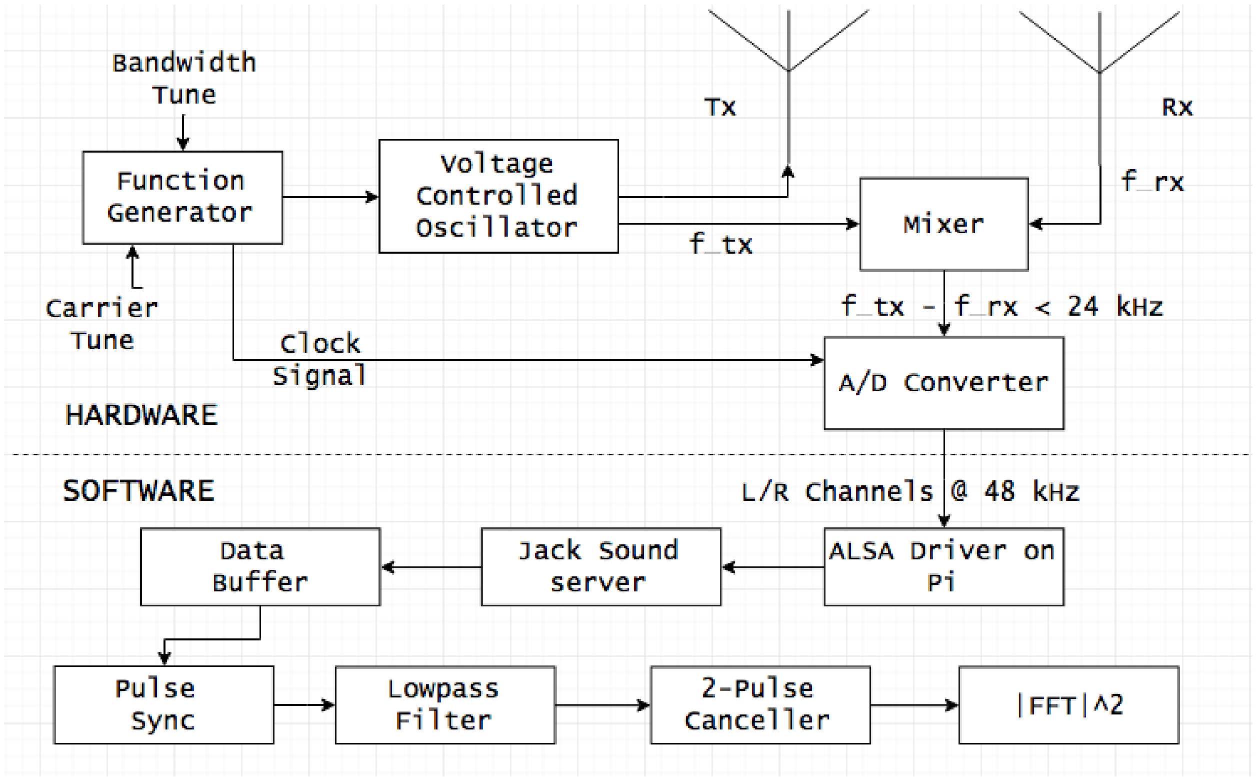

Figure 2: SDR Block Diagram

modes - Doppler and FMCW. These Matlab prototypes then became the reference point for developing Python on the Pi.

This process naturally led to the development of two tools which became essential throughout the development - software audio oscilliscopes that plugged into various source data formats. Eventually this settled into one program used to pull various results from the Pi to the development laptop and display the data in real-time in Matlab figures. The second program is similar except that is fetches data from either the disk or the sound card and performs some processing and then displays those results in a streaming fashion.

Finally, algorithms were migrated to Python on the Linux system and development proceeded over remote shell only. During this phase, publishing results to sockets that could be read from my laptop using the tool mentioned above was critical for testing and integration.

System Description

Specification of the public interface

Inputs

Outputs

The Raspberry Pi does have the capability of running a desktop environment with graphics in order to display results, yet the overhead is significant and so the decision was made leave the Pi running in a headless manner and offload the resulting data over a local network to the development laptop for display. For test and debugging purposes, it was desirable to have the ability to visualize the data as it progressed throught the processing chain. The system output interface is visualized below.

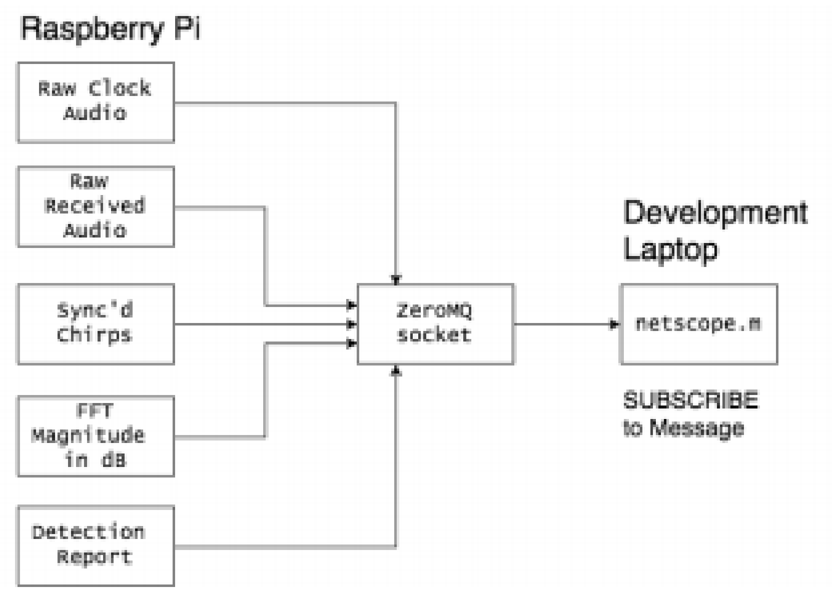

Figure 3: SDR to Development Laptop Interface

As seen, the program `netscope.m` takes a message name string for which a ZeroMQ socket is created which subscribes to that message type. The PUBLISH/SUBSCRIBE topology enables the Pi to artifically put all the data on the wire, yet only transfer the data that is requested. Matlab does not actually have a ZeroMQ implementation, yet it does expose a Python interface, through which ZeroMQ can be reached seamlessly.

In addition to the above the system also prints out status messages to the Raspberry Pi console.

Were the system to progress into a more productized instantiation, offloading reports could be achieved over a ZigBee link. Of course, this would limit the data rate to discrete detection measurements as opposed to the current toolset which allows viewing the data at any stage in the processing chain.

Algorithm Descriptions

Doppler CW

A Doppler CW system is able to measure the instantaneous radial velocity of a moving object. When an electromagnetic wave reflects off of a moving object, say a car, the wave is shifted in frequency by an amount proportional to the wavelength of the RF signal and the projection of the car's velocity onto the line from whence the wave originated. Doppler CW systems use a continuous sinusoid shifted to the carrier frequency. When the received signal is mixed with the transmitted signal, the difference is output. By performing a Fourier Transform on the received data, over some period of coherency, the radar is able to measure the magnitude response of the data at various frequencies. These are then related to speed using the wavelength.

Frequency Modulated CW (FMCW)

In FMCW, a triangle wave is generated rather than a sinusoid. When the triangle is passed through a voltage-controled oscillator (VCO) the ramp produces a linear frequency modulation (LFM) known as a chirp. This enables measurement of object distance from transmitter over the period of the ramp. As with Doppler, a frequency difference is measured and related to range by the speed of light and the bandwidth of the frequency ramp. However, for FMCW the coherent period is constrained to the period of the triangle ramp. Therefore, FMCW signals must be synchronized to the reference clock signal. This enables knowing the time the ramp was transmitted which can be differenced with the peak return in frequency for a measurement of distance.

Another difference is the dominance of stationary objects on the response spectrum. These are collectively referred to as clutter and reside at zero Hz (DC), though the energy bleeds into the nearby frequencies as well. If a radar is primarily interested in objects that move, clutter can be mitigated by taking a slow-time derivative of the data. Slow-time in that the time interval is measured in ramps instead of samples. By subtracting the previous respnse from the current response, much of the DC energy is removed and moving objects are left in the response. This clutter-mitigation strategy is known as a two-pulse canceller.

Timing constraints

For a sensor of any type, the time scale is ultimately driven by the kinematics of the objects of interest in the environment. At some point a simplification must be made and a period extablished over which it is assumed that the environment is stationary. For the case of our SDR, the objects of interest are of the human-walking and car-driving variety. Each of these is slow relative to airborne jets and so our time scale has some margin compared to typical radars.

If we assume the upper bound of stationarity to be a car moving at 35 m.p.h that translates to about 1 foot in 20 milliseconds and 5 feet in 100 milliseconds. Somewhere between 20 and 100 milliseconds we can reasonably assume our sensed environment is static. Our chirp ramps are tuned to last for 20 milliseconds, and so this becomes the fundamental unit of time for processing. Additionally, we can average over 1 to 5 of these pulses to smooth the output results and still have some confidence of the scene remaining roughly stable.

The other time constraint in this case is the ability of processing resources to handle the throughput requirements. Fortunately, the relatively slow speed of the objects of interest coupled with the fact that our system does not have the power to see beyond 1 km for a 10 square-meter target, means the information of interest is contained within a very narrow region of spectrum near DC. In fact, it is within the bandwidth of human auditory sensing. This pleasant coincidence results in an abundance of analog-to-digital converters with the requisite sample rates. This also means that our incoming data rate of 48000 samples/second is manageable for modern processor chips. Ultimately, required processing throughput depends on the data rate and the Raspberry Pi can handle a few audio signal processing operations within the required time frame.

Error handling

We can categorize the classes of errors as those which are induced by unexpected inputs, those induced by uncaught exceptions within the primary Python routine, and those induced by kernel scheduling, causing lag or loss of flow.

The first occur when the routine is unable to synchronize to the reference clock signal. This exception is caught by wrapping the sync block in a try/catch structure. If we are unable to sync, we want the routine to keep trying without falling apart. This reflects the software's inability to control external events, such as low battery power, fried circuits, or some other failure of the signal chain. Similarly, the routines downstream of sync need to predicate their execution on the indication from sync that everything is working. This is achieved with control flags once sync is achieved. Additionally, each pulse iteration checks the period of the sync interval to validate stability and throws a syncLost flag if sync is lost. Therefore, acquireSync is the initial state to which the routine returns if exceptions are encountered.

The second, exceptions within the main Python routine, are due to software bugs and unexpected corner cases. For our prototyping system, these are addressed using the built-in Linux kernel control wrapper called `systemd`. We stress that when the input is operating correctly and the kernel is able to keep up with the data processing and throughput requirements, there have as yet not been uncaught errors within the main loop.

However, in order to handle the unexpected, three `systemd` unit modules were written to indicate what actions should be taken when one of the main programs crashes unexpectedly. The modules can be enabled as services which allow them be started automatically after a reboot once the kernel boot reaches the point where the device drivers have been intialized. Additionally, event actions can be specified, such as OnFailure or OnWatchdog. Precedence may be set such that program two waits until program one has successfully intitialized. In this way, we are leveraging the existing tool set within the wider Unix community to handle simple control flow and error handling. This is opposed to writing a custom routine to interact with the kernel scheduler. For a rapid-prototyped demonstrator, this reduces risk and cost. A produciton system might require more extensive effort to guarantee proper exception protocol and real-time scheduling priority.

This framework also enables handling the third failure mode where the kernel scheduling causes the audio server to fall behind resulting in audio discontinuities. In this case, the audio server, which is implemented using the third-party Jack library with ALSA as the device backend, is controlled by a custom `systemd` service which stops and restarts the server in the case where errors occurs due to overruns.

Hardware Implementation

Software Implementation

For brevity, we will focus on the Python real-time code here and not on the Matlab tools or prototyping routines. A flow chart is presented below for reference.

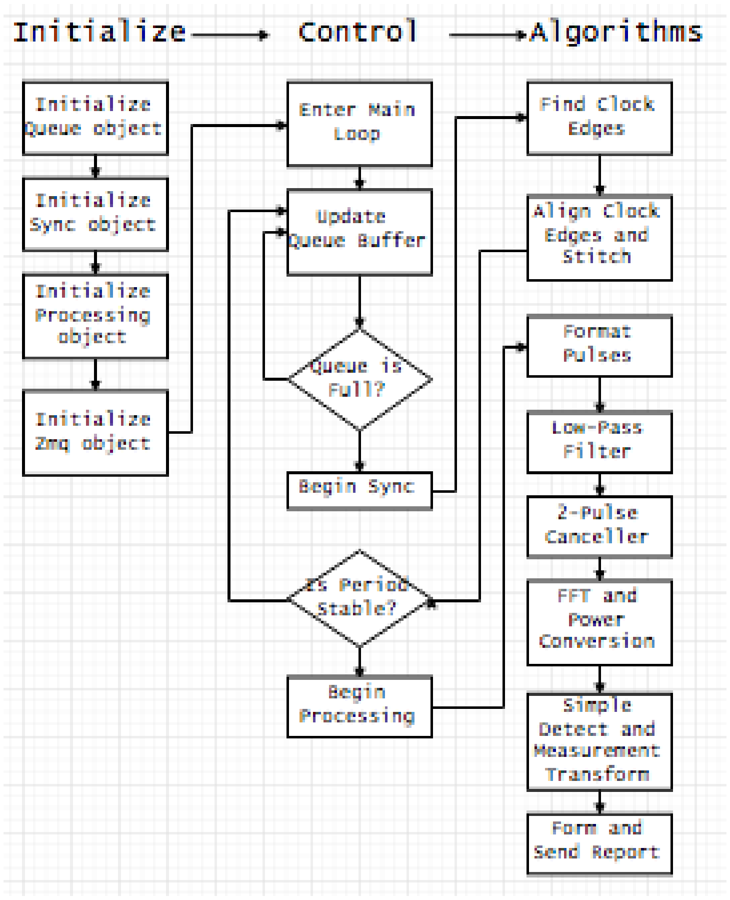

Figure 4: Software Flow Chart

The above diagram illustrates how the main routine, `sdr_main.py`, is structured with four data management classes. The Zmq class provides management of ZeroMQ sockets. The Queue class provides extensible buffers for interfacing with the audio server packets as well as handling formatting of socket byte arrays.

There are two programs that both use these two objects. These are the FMCW application, which detects range, and the Doppler application, which detects radial speed. The Doppler processing is essentially a subset of the FMCW processing and so we will focus on that program.

The primary algorithmic challenge of ranging is synchronizing the data to the transmitted pulses. The Sync class receives numeric data from the Queue and processes the clock signal to find the rising edges of the pulse. Using these indices it fills a matrix with the received samples corresponding to the transmitted chirps. Additionaly, Sync implements methods to ensure the clock period is stable.

Finally, the Processor class performs a series of matrix operations on a buffer of chirp pulses to compute each pulse's range response. These include low-pass filtering, clutter cancellation, and discrete Fourier transform. Minus the Synchronization step and clutter cancellation, the Doppler processing is very similar.

Each of these operate and are update in a loop in the main program. Additionally, there is a second program that interfaces between the sound server and `sdr_main.py`. This program executes a callback when there is new audio data available on the server and then publishes that to a ZeroMQ socket which the main program buffers.

Test Plan

Presentation, Discussion, and Analysis of Results

Testing was performed by walking toward and away from the sensor. However, the more interesting results used cars as the sensed objects. The system was set on top of Paul's car, parked alongside a road in Maple Valley with a fair amount of traffic. The Raspberry Pi received and processed all data in real time and routed the results over a network link to the development laptop. The youtube link below contains two video streams, a recorded video of the traffic on the left, roughly synchronized to the range-time indicator waterfall on the right. Outgoing traffic can be seen as lines angled down and to the right. The oncoming traffic show as lines angled down and to left. The outgoing vehicles are brighter as the field-of-view is focused on these.

FMCW Ranging Demonstration https://www.youtube.com/watch?v=gWq5N82Wgkw

The Doppler demonstration was collected much earlier in the project. In this case, the speed-time indicator waterfall is on the left while the right contains a static image which shows an overview of the entire collect. This video also contains processed audio proportional to the Doppler frequency.

Doppler Velocity Demonstration https://www.youtube.com/watch?v=JB-oInUjbWk)

These results demonstrated real-time processing of received radar data and measurement of environmental phenomena in two dimensions. Achieving the results in the FMCW video required significantly more work than the Doppler. The two challenges associated with FMCW were synchronizing streaming pulses and the 2-pulse canceller. For each, the challenge lay in ensuring the pulses were correctly stitched together over multiple buffers.

On the FMCW display, the vehicle's range response continues out to about 200 meters. Greater distances could be achieved with more transmit power.

Analysis of Any Errors

Analysis of why the Project may not have worked and Efforts made to Identify root cause issues

Summary and Conclusion

We have demonstrated development of a complete radar sensor using off-the-shelf hardware and dynamic programming languages. We were able to demonstrate two separate modes of operation and are able to switch using software only. Future research could take this design, complete the PCB print, and add interfacing inputs to recieve signal inputs from the Pi. In this manner, steps could be made toward a Cognitive Radar that tuned parameters based on measurements.

In concept, the algorithms that perform the kernel of the work are relatively straightforward. One lesson learned was that porting those prototyping code to run in real-time with streaming data was a significant challenge. Additionally, the nature of developing on a unit without a built-in display requires robust connectivity between development and target platforms. In our experience, both the Wi-Fi and Ethernet links typically required manual intervention to operate properly.

Individual Contribution

Paul Adams

o Imaging, set-up and configuration of the Raspberry Pi 2

o All real-time Python software

o All Matlab prototyping and tool software

o All automating and utility bash scripts and Linux service interaction

o Testing and configuration of 3rd libraries on Linux

o All system testing and integration

o Development of all algorithms that deviated from reference design