Russell Silva
7/15/2018 – 7/29/2018
AMRUPT, Summer ‘18

# Report #3 – Continuous I/Q Extraction from the RTL SDRs without Synchronization Losses

## Overview

The synchronization of data streams was successfully performed using the cross correlation code here [1] as a starting point, which makes use of the FFT convolution [2] to decrease the computational complexity of cross correlation in the time domain. All the import dependencies (i.e. Numpy) can be installed using pip [5], although all the dependencies were already present in the Vanilla Raspberry OS. The python code can be called from a script after making it executable [6]. After successfully cross correlating the data streams from each RTL SDR, several methods have been attempted in the past week to achieve continuous I/Q extraction from each RTL SDR receiver without synchronization losses from each of their data streams after cross correlation.

One could continuously extract samples to a file from an individual core on the Raspberry Pi and use the samples from each growing file. The samples from each file would remain synchronized for a long period of time after performing cross correlation. Also, this method could be in real time using the tail end of samples from each file; however, eventually each file would grow too large in memory. Clearing each file produces timing offsets between the data streams, therefore additional procedures must be implemented to get the data streams to be remain synchronized, or cross correlation would have to be performed for discrete blocks of data (the timing of the samples within the blocks would be determined by the data block’s timestamp and the sampling rate).

## Named Pipes

Multiple commands on the Raspberry Pi (i.e. I/Q extraction, cross correlation, DF algorithms) can be called in series according to a Bash Script [4]. The output of one command can be used as the input of another command in a procedure known as piping [7]. This is done simply in the following fashion: command 1 | command 2.

Kruger’s fastcard [8] README mentions that either samples can be piped from the rtl_sdr executable or by reading directly from the SDR using librtlsdr. This is an important point of consideration for the circular buffer approach (discussed in later section).
The prospect of named pipes seems very promising at first glance for being able to synchronously extract a set byte size from rtl_sdr input commands at any time. Please note that the number of bytes of a file directly correspond to the number of samples extracted, as proven in finite sampling benchmark tests using the wc -c “filename” command to check the number of bytes in a file.

However, when the head command [9] was used to extract a set byte size from a named pipe [10] input, a SIGPIPE kill code is sent to the named pipe sender. A workaround to this is to use a

“trap” command to ignore the SIGPIPE kill code [11]; however, this did not catch the kill code successfully.

The location of the Named Pipe implementation can be found here.

**TCP/UDP**

Attempts to have RTL SDR senders and noncontinuous listeners has been attempted by using TCP/UDP [12] on a localhost IP/port. A UDP connection has been attempted using netcat (a linux application commonly used for UDP/TCP sending/receiving). Piping the netcat listener to a head command to extract a certain amount of data only works for one iteration. Subsequent netcat listeners refuse to function. To handle this situation, custom TCP/UDP protocols have been considered [14], so that listeners can close without severing the TCP/UDP server.

The location of the UDP implementation can be found here.

**Logrotate**

It occurred to me that there must be some mechanism to rotate rapidly growing log files (consider a high velocity stock market data stream that needs to be inputted to a file). Once the RTL SDR input files reach a certain size, a log rotation program such as this one [15] or this one [16] can be used to quickly reroute data streams to a separate log file once the original log file reaches a certain size.

This was only recently considered, so a prototype implementation for this method has not been developed yet.

**Circular Buffer**

Work has been done in setting up a circular buffer in python. The prototype implementation uses this resource [19] as a starting point. Moreover, a testing data stream (sending the current time of day every millisecond) has been piped to this python executable for debugging. Print statements have yielded unexpected contents of the buffer, so configuration is still being performed [20]. Under this implementation, one thread will take in samples from stdin into the circular buffer and a separate thread will be used to push samples from the buffer to a file every tenth of a second. If the stdin datastream can be used without any samples lost in python, the circular buffer will be reliable with an input pipe rather than reading directly from the SDR using librtlsdr.

Kruger’s fastcard implementation [8] uses a circular buffer to constantly stream samples from the RTL-SDR without the overhead of a pipe. There are some implications for a circular buffer in our system, which uses multiple RTL-SDR datastreams (thus multiple circular buffers will be needed). It will be difficult to push synchronized data from the circular buffers. The synchronicity of data within the circular buffers themselves will not be a problem since each circular buffer will grow at the same rate if no samples are lost. However, there is no way (according to my embedded systems knowledge) to push a synchronous window of samples to different files (for each RTL SDR) at a given time.

There are two ways to initiate a push from the circular buffers: polling and interrupts [17]. In polling, running four separate while loops with no sleep will take up a large portion of cpu. Moreover, these while loops will be limited by the Raspberry Pi's 2441 millions of instructions per second [18], which would cause microsecond delays between the while loops, especially with separate processes running on the Pi. Unless the ADC's high sampling rate (currently using ~200 mega samples per second) is interpolated to a lower sampling (the circular buffer intakes every $10^{th}$ sample), there will be sampling offsets with polling. Interrupts with the Raspberry Pi experience the same problem with 500 microsecond offsets [19].

There are two workarounds:

1. Instead of file pushing (we will not be able to signal when to extract a certain amount of data at a given time), log rotation will be implemented to keep the data synchronized. For example, write 10000 samples every 100000 samples are dropped to separate files sequentially.

2. Millions of samples can be written to a file at a time (will allow for polling/interrupts). The large number of samples will allow us to correct for file push offsets by comparing the "head value" of each circular buffer. By writing the "head value" of each buffer to a config file in a separate thread at the time of each file push, we could use windows of data far away from the head values (therefore a large number of samples must be used, to correct for offsets). If the head values are close to certain boundaries, we can discard the data (this can be discussed in further detail later). However, timestamping and sampling rate timing will have to be implemented to differentiate smaller snapshots that will be used in autocorrelation/MUSIC. Timestamps can be appended to each data block as done in Kruger's fastcard implementation here (lines 77-95).

**Resources and relevant Forum Posts**


[1] Cross Correlation Program:
https://github.com/tejeez/rtl_coherent/blob/master/crosscorrelate.py
[2] Explanation of the FFT Convolution: http://www.dspguide.com/ch18/2.htm
Bash Shell:
[3] Including Delays:https://stackoverflow.com/questions/32359374/how-could-i-run-a-shell-script-with-delay
[4] Writing a Shell Script: http://www.circuitbasics.com/how-to-write-and-run-a-shell-script-on-the-raspberry-pi/
Python:
[5] Installing Dependencies: https://raspberrypi.stackexchange.com/questions/71127/how-can-i-install-numpy-and-pandapower
[6] Making Python File Executable: http://www.circuitbasics.com/how-to-write-and-run-a-python-program-on-the-raspberry-pi/
[7] https://ryanstutorials.net/linuxtutorial/piping.php
[8] https://github.com/swkrueger/Thrifty/tree/master/fastcard
[9] http://www.linfo.org/head.html
[10] https://www.linuxjournal.com/content/using-named-pipes-fifos-bash
[11] https://www.tutorialspoint.com/unix/unix-signals-traps.htm

[12] https://support.holmsecurity.com/hc/en-us/articles/212963869-What-is-the-difference-between-TCP-and-UDP-
[13] https://www.computerhope.com/unix/nc.htm
[14] http://www.linuxhowtos.org/C_C++/socket.htm
[15] https://httpd.apache.org/docs/2.4/programs/rotatelogs.html
[16] https://linux.die.net/man/8/logrotate
[17] https://techdifferences.com/difference-between-interrupt-and-polling-in-os.html
[18] https://hackaday.com/2016/03/01/pi-3-benchmarks-the-marketing-hype-is-true/
[19] https://www.safaribooksonline.com/library/view/python-cookbook/0596001673/ch05s19.html
[20] https://stackoverflow.com/questions/30043857/bash-pipe-to-python

Installations:
**sudo apt-get install bc : will enable floating point math in bash**
**sudo apt-get install netcat : for tcp messaging**
**sudo apt-get install buffer : buffering program**
**sudo apt-get install apache2 apache2-utils : logrotate**

Making Bash Script Executable:
```
Make it executable: chmod u+x bash_script_name
Run the Script: ./scriptname.sh
```