CONVERTING RADIO SIGNALS TO DATA PACKETS

Examination of Using GNU Radio Companion for Security Research and Assessment

May 15, 2014

Presented by:

INGUARDIANS, INC.

Security Research and Guidance

TABLE OF CONTENTS

1.0 Introduction

InGuardians, Inc. (InGuardians) has leveraged wireless assessment methodologies since our inception. Wi-Fi assessments involving Enterprise wireless deployments have slowly grown to include analysis of radio technologies leveraging common and custom deployments of Zigbee, Bluetooth, and mesh networking implementations within the Industrial, Scientific and Medical (ISM) radio bands. InGuardians has seen these technologies expand rapidly with the rapid deployment of Smart Grid-based technologies and the proliferation of embedded devices in many industries including utilities, oil and gas, retail, grocery, aviation, and medical.

InGuardians' approach has always been to take the current intelligence from other researchers and expand upon their guidance and theories. Sometimes we have been lucky enough to stumble upon new developments of our own and lead the discovery of new vulnerabilities and issues. Most of the time, like many other security professionals, we have used the data discovered by others to define and refine our techniques while periodically augmenting it with our own information, expertise, and programming skills. InGuardians has continued this learning and leadership role in the radio analysis field by sharing our research, assessment, and development skills with the security community and associated industries as often as possible. Our analysts have presented on these topics at ShmooCon[1], Black Hat[2], DefCon, S4, and other industry-specific venues.

In our quest to fully understand the techniques behind radio assessments, InGuardians has determined there is a lack of specific step-by-step guidance demonstrating some of the many radio analysis techniques. The biggest gap appears to be centered on the use of GNU Radio Companion (GRC) to completely analyze a signal from capture to the data the signal contains. This is partially due to the vast number of implementations that are possible for any device configured to implement a radio as a method of communications and data transfer. But, as with all information technologies, understanding the steps behind one or more scenarios opens doors for other research and tools. To this end, InGuardians offers the following step-by-step guide which outlines our experiences and approach to radio analysis using GRC. These efforts have also resulted in a custom script, GRC Bit Converter[3], to help assist security researcher and radio enthusiasts with their personal and professional projects.

```
DISCLAIMER: Several InGuardians have held HAM certifications for many years.
However, none of our security analysts have degrees relating to Radio
Engineer. The information provided here is accurate to the best of our
knowledge. These are techniques gleaned from Mike Ossmann's radio analysis
course[4], extensive individual research, and during actual radio analysis
assessments. The terms and technique are accurate to the best of our
knowledge. This paper will be updated to address any major inaccuracies.
Please notify us with any input or clarification.
```

[1] Hop Hacking Hedy: http://code.google.com/p/hedyattack/downloads/list
[2] Looking into the Eye of the Meter: http://www.youtube.com/watch?v=hXfpEauUCto
[3] GRC Bit Converter: https://github.com/cutaway/grc_bit_converter
[4] Great Scott Gadgets: https://greatscottgadgets.com/

2.0 Installation and Configuration

There are many tutorials and guides for purchasing radios, installing GRC, and installing supporting spectrum analysis software. Repeating these sources would be futile. Therefore, the following is a list of hardware and software that InGuardians analysts use as a starting point for research and security assessments. The links provided will change over time and new software and hardware will eventually be released. Thus these resources are only current at the time this guidance was written.

- Hardware Resources
  - USRP – http://home.ettus.com/
  - HackRF Jawbreaker – https://github.com/mossmann/hackrf/wiki (shown in Figure 1)
  - bladeRF - http://nuand.com/
  - RTL-SDR – http://en.wikipedia.org/wiki/List_of_software-defined_radios
  - CC1111EMK - http://www.ti.com/tool/cc1111emk868-915
  - Ubertooth – http://ubertooth.sourceforge.net/
  - Atmel RZUSBstick - http://www.atmel.com/tools/rzusbstick.aspx
- Software Resources
  - GNU Radio Companion – http://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion
  - SDR# - http://sdrsharp.com/
  - GQRX – http://gqrx.dk/
  - Baudline - http://www.baudline.com/
  - RFcat – http://code.google.com/p/rfcat/
  - Ubertooth - http://ubertooth.sourceforge.net/
  - KillerBee - http://code.google.com/p/killerbee/

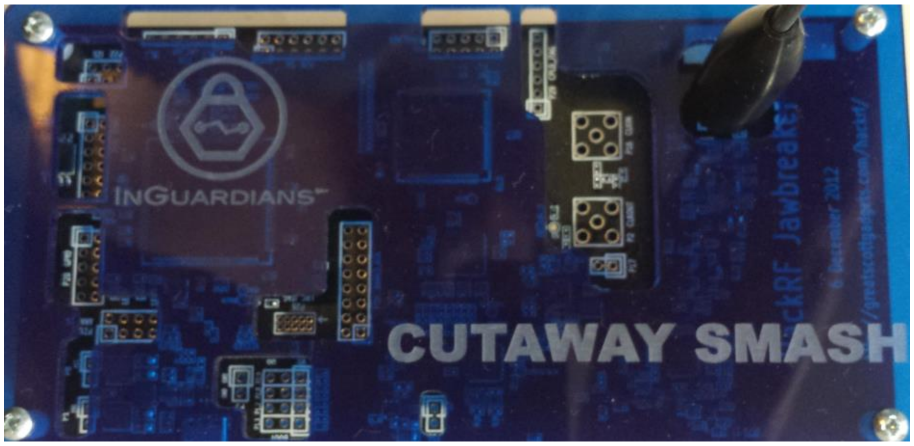
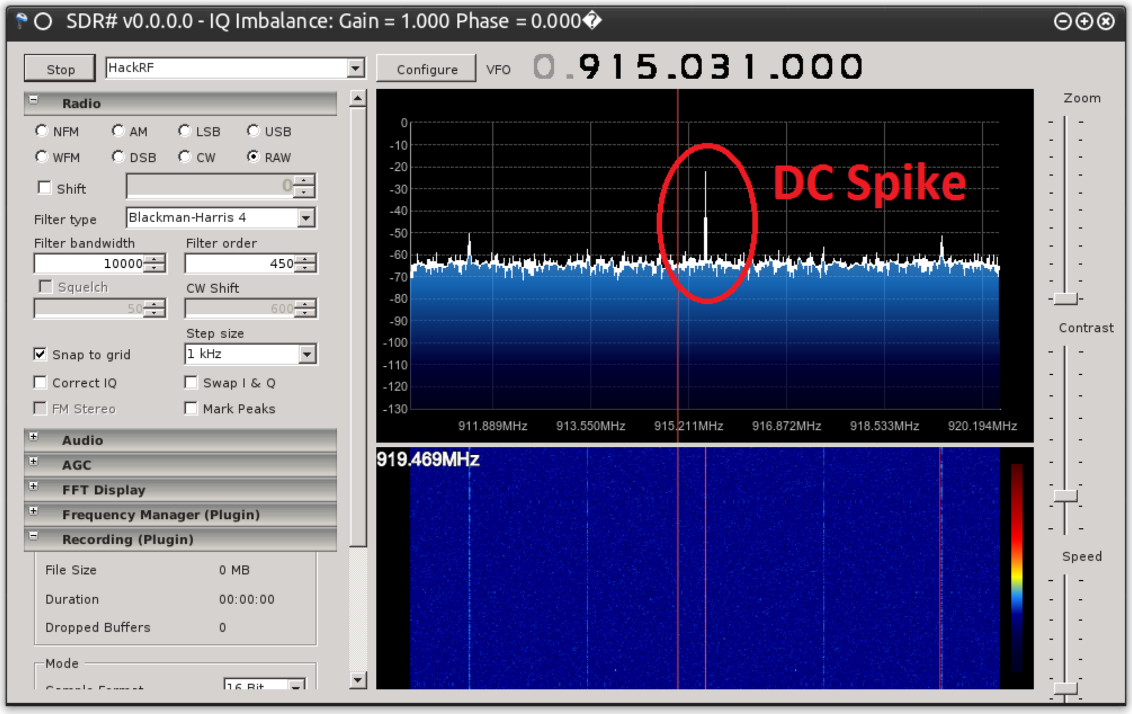


Figure 1 HackRF in Custom Case

Getting started with GRC is fairly easy. The GRC GUI provides the user an easy-to-use interface designed to implement the complicated radio functionality provided by this program. GRC relies heavily on configurable “blocks” to perform a specific action. Blocks are linked so that a signal flows from one block to another being slightly or majorly modified along the way. Again, there are many tutorials describing the basic functionality of GRC and the primary blocks used for capturing and displaying. Gaining an understanding of getting started with GRC is an exercise left up to the reader. InGuardians recommends that the read install and “play” with the software previously listed. Leveraging the spectrum analyzers (GRC’s FFT, SDR#, and GQRX) will provide the reader with the basics of tuning and visualizing radio transmissions.

3.0 MANAGING DIRECT CURRENT SPIKE

Users of the HackRF, bladeRF, or one of the many RTL-SDR dongles are going to see a large spike at the location where the radio has been tuned. This is the Direct Current (DC) spike, seen in Figure 2, which occurs naturally in radios that have not specifically accounted for this spike via hardware / firmware. The HackRF team does a great job explaining why this occurs and what you can do with it in their Frequently Asked Questions (FAQ).[5]



SDR# v0.0.0.0 - IQ Imbalance: Gain = 1.000 Phase = 0.000

HackRF

VFO 0.915.031.000

Radio

NFM AM LSB USB WFM DSB CW [✓ RAW]

Shift

Filter type: Blackman-Harris 4

Filter bandwidth: 10000

Filter order: 450

Squelch: 50

CW Shift: 600

Step size: 1 kHz

[✓ Snap to grid]

Correct IQ

Swap I & Q

FM Stereo

Mark Peaks

File Size: 0 MB

Duration: 00:00:00

Dropped Buffers: 0

Mode

Sample Format: 16 Bit

Figure 2 HarkRF Jawbreaker's DC Spike

As described in the HackRF FAQ, the DC spike, shown in Figure 2, can be ignored in most situations. However, to successfully demodulate a captured signal and obtain the transmitted data, it may be necessary to have as clean a signal as possible. Therefore, to avoid the DC Spike a radio capture can be configured to avoid it using the "DC Offset" method. This method entails capturing the radio's transmission by tuning the radio to a frequency just outside of the transmission's bandwidth. Capturing with an offset is easy to accomplish with the current GRC blocks. The challenge is to select an offset that moves the DC Spike outside of the source's transmitted signal but not too far which could force us to capture more bandwidth than we need.

Two methods can be implemented to determine the configuration of the DC Offset. The first is to conduct data sheet analysis to determine the radio and device's capabilities. These documents will outline "Channel Spacing." The channel spacing is the distance between the center frequencies of two transmission areas configurable by the radio. While this helps, and is often enough information to adjust for the DC Spike, the channel spacing is not necessarily related to the size of the transmission. We see this in Wi-Fi which has fourteen (14) channels but the transmission of a wireless adapter will engulf approximately six (6) of those channels. Reviewing the bandwidth of the transmission with a spectrum analyzer helps compensate for this possibility and is also the second method. Figure 3 (a display of radio

[5] HackRF FAQ: https://github.com/mossmann/hackrf/wiki/FAQ

transmissions over a five minute period) is a good example of Channel Spacing used in a radio-based mesh network implementing Frequency Hoping Spread Spectrum (FHSS). Using the "Peak Hold" option in the GRC FFT, the DC spike can be seen in relation to the center most transmission in this capture.

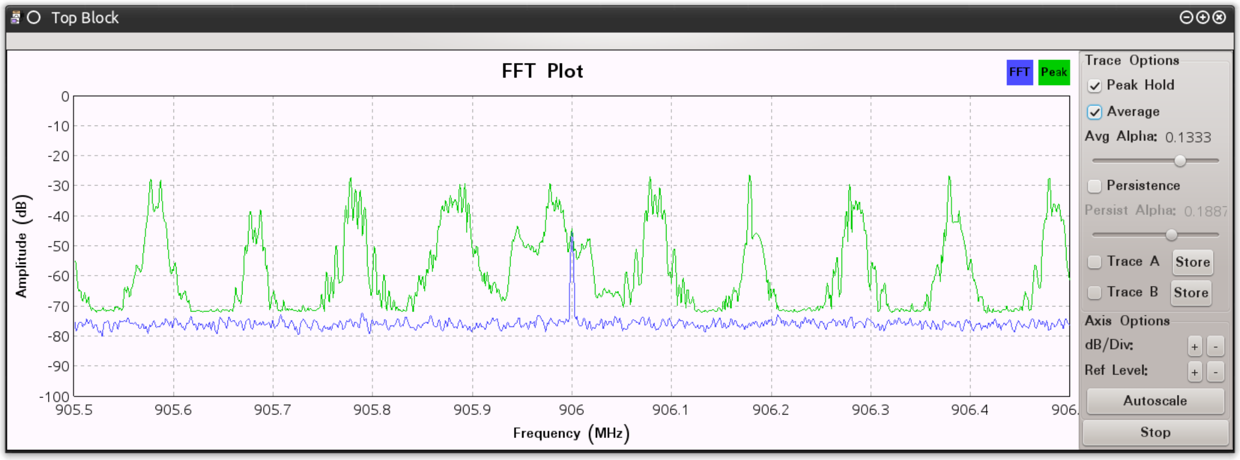


Figure 3 FHSS Transmissions Demonstrating Channel Spacing

Radios use Channel Spacing so that multiple radios can be deployed in close proximity and reduce the likelihood that one transmission interferes with another transmission. The default Channel Spacing will vary by manufacturer and radio type and can be unique in any deployment. Because it is a defined configuration setting for all radios, it can also be found in manufacturer documentation for that radio. The information provided by these documents will be associated with the configuration options rather than the actual value being used by an implementation. Knowing the specific values for a specific implementation requires documentation from the device's vendor. This may or may not be available. Using online reconnaissance can turn up additional information in source code, patent filings, and forums.

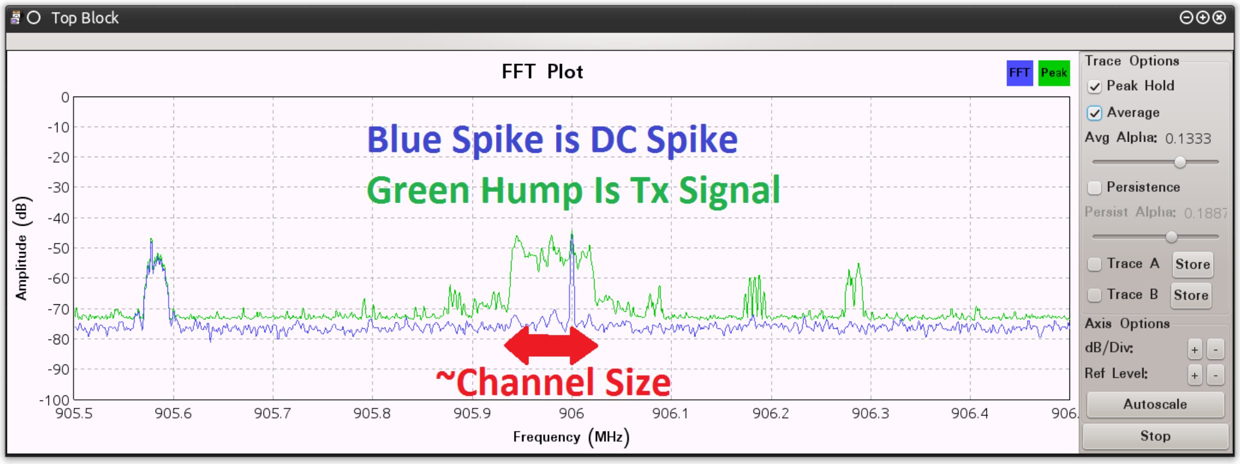
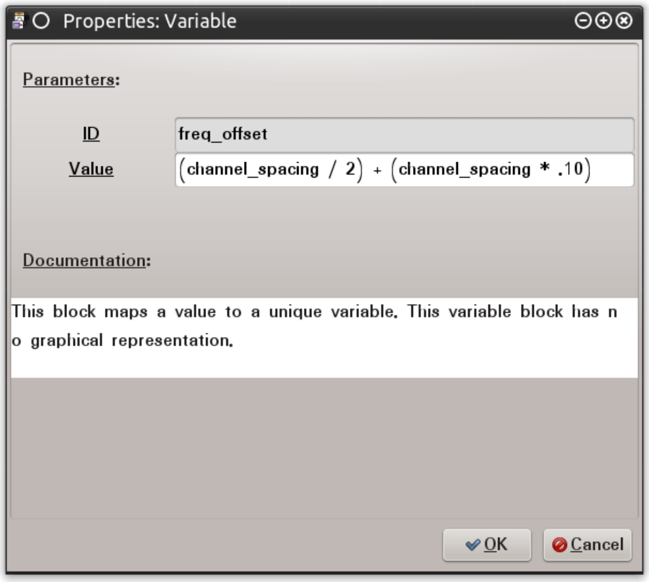


Figure 4 DC Spike Inside TX Signal

When documentation is lacking, or settings need validation, visual analysis is a simple technique to obtain the approximate Channel Spacing values. Figure 4 is an example of visually determining the Channel Spacing by observing the distance between both edges of the transmission spike. It has been labeled "Channel Size" in this image to help understand the measurement of the transmission in the absence of other transmissions.

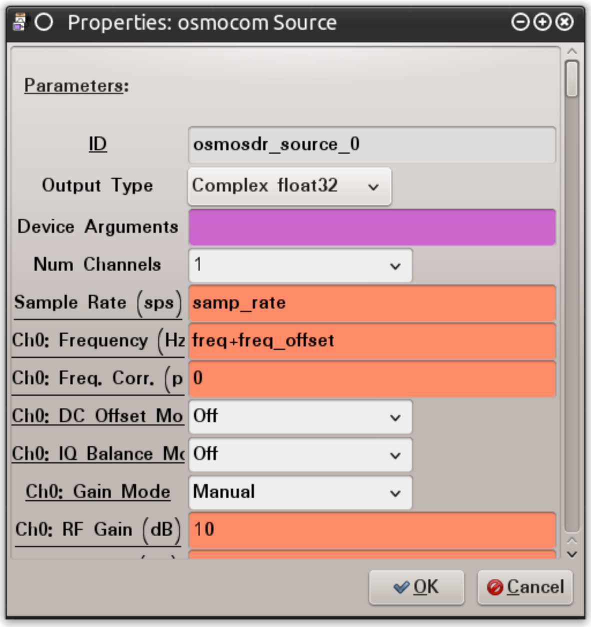
GRC uses variable blocks, like the default "samp_rate" block, to configure settings that may be modified during research. In the following examples the Channel Spacing value will be defined as "channel_spacing" variable block. Once defined this variable block can be used to configure other blocks. To help leverage Channel Spacing to manage the DC spike, a variable block named "freq_offset" will be configured with the equation: *(channel_spacing / 2) * (channel_spacing * .1)*. Breaking this equation down, dividing the Channel Spacing by two will move the DC Spike to the very edge of the transmission. Adding an additional 10 percent to this offset "should" move the DC spike completely outside the transmission. Figure 5 demonstrates the "freq_offset" configuration and Figure 6 shows how it is implemented within the source block.



ID	freq_offset
Value	(channel_spacing / 2) + (channel_spacing * .10)

This block maps a value to a unique variable. This variable block has n
o graphical representation.

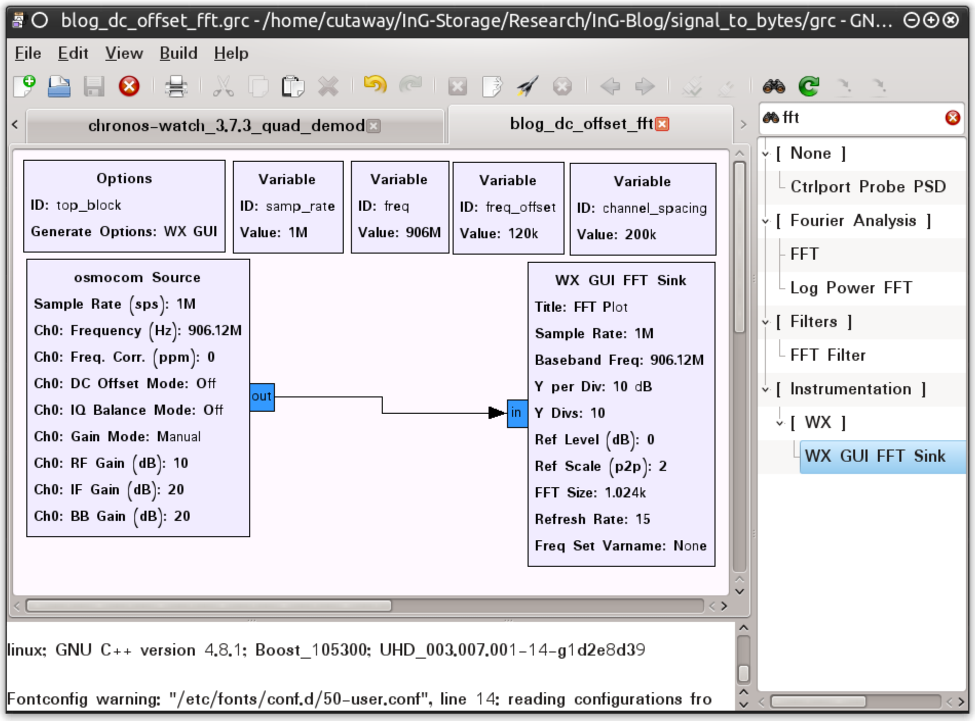
Figure 5 GRC "freq_offset" Variable Block Configuration



ID	osmosdr_source_0
Output Type	Complex float32
Device Arguments	
Num Channels	1
Sample Rate (sps)	samp_rate
Ch0: Frequency (Hz	freq+freq_offset
Ch0: Freq. Corr. (p	0
Ch0: DC Offset Mo	Off
Ch0: IQ Balance Mo	Off
Ch0: Gain Mode	Manual
Ch0: RF Gain (dB)	10

Figure 6 GRC "osmocom Source" Block Configuration

Once these blocks have been configured, each block in the primary GRC window configured with these equations will display the resulting values. The calculated values are show in Figure 7.


blog_dc_offset_fft.grc - /home/cutaway/InG-Storage/Research/InG-Blog/signal_to_bytes/grc - GN...
File Edit View Build Help
chronos-watch_3.7.3_quad_demod
blog_dc_offset_fft
fft
[None]
Ctrlport Probe PSD
[Fourier Analysis]
FFT
Log Power FFT
[Filters]
FFT Filter
[Instrumentation]
[WX]
WX GUI FFT Sink
out
in

Options
ID: top_block
Generate Options: WX GUI

Variable
ID: samp_rate
Value: 1M

Variable
ID: freq
Value: 906M

Variable
ID: freq_offset
Value: 120k

Variable
ID: channel_spacing
Value: 200k

osmocom Source
Sample Rate (sps): 1M
Ch0: Frequency (Hz): 906.12M
Ch0: Freq. Corr. (ppm): 0
Ch0: DC Offset Mode: Off
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Manual
Ch0: RF Gain (dB): 10
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20

WX GUI FFT Sink
Title: FFT Plot
Sample Rate: 1M
Baseband Freq: 906.12M
Y per Div: 10 dB
Y Divs: 10
Ref Level (dB): 0
Ref Scale (p2p): 2
FFT Size: 1.024k
Refresh Rate: 15
Freq Set Varname: None

```
linux; GNU C++ version 4.8.1; Boost_105300; UHD_003.007.001-14-g1d2e8d39

Fontconfig warning: "/etc/fonts/conf.d/50-user.conf", line 14: reading configurations fro
```

Figure 7 GRC Capture Configuration

The resulting Top Block, as seen in Figure 8, shows that the DC Spike has been shifted outside the transmission.

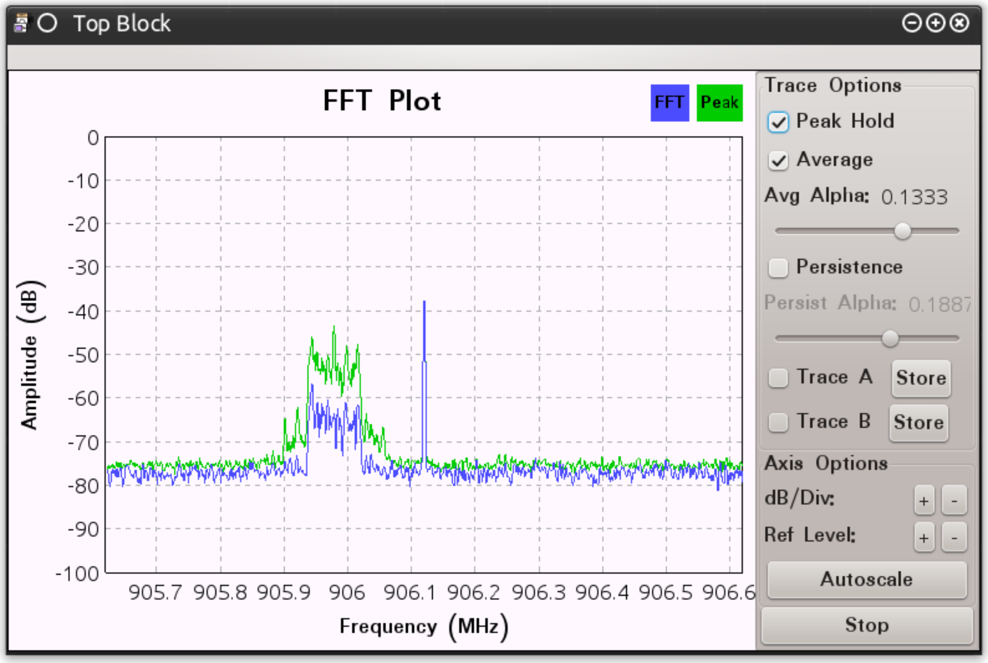


Figure 8 GRC Top Block With Offset DC Spike

4.0 Isolating The Transmission

Once the DC spike has been addressed the transmission can be processed without its interference. The basic concept is to zero in on the transmission and isolate it from all of the other signals. GRC includes two different blocks to help with this isolation: the "Low Pass Filter" (LPF) and the "Frequency Xlating FIR Filter" (FXFF) blocks. Both of these blocks provide the same primary functionality. The FXFF, however, provides a few more options to help isolate and begin to manipulate a transmission. The LPF will be explained first as its variables are also leveraged by the FXSS.

The first configurable option provided by the LPF is the "Decimation" value. This allows us to modify the sampling rate of the incoming signal. Decimation is particularly valuable when taking a transmitted signal and modifying it for output to audio files as seen in many GRC examples for FM demodulation. Configuring this variable is usually done using the equation-method as shown when calculating the Frequency Offset. Specifying the input sampling rate, which is usually automatically identified by the "samp_rate" variable block, and dividing by the intended (a.k.a output) sampling rate will result in this block outputting the signal with the intended output sampling rate. While leveraging decimation for some demodulation is necessary, it does not need to be used for this example. Therefore the Decimation value can be left at the default value of "1" which indicates no change to the Sampling Rate. Not decimating also helps avoid any issues associated with breaking the Nyquist–Shannon sampling theorem rule.[6] This rule simply states that a signal must be sampled at twice the data rate to avoid data lose.

The next value in the LPF block to take into consideration and modified in the LPF block is the "Window" value. The default value for this variable is "Hamming". However, Balint Seeber's explained in his talk "Blind signal analysis with GNU Radio"[7] that this value should be changed to the setting "Blackman" as it is the superior algorithm.

With the Decimation and Window parameters configured the "Sample Rate," "Cutoff Frequency," and "Transition Width" can be configured to isolate the transmission. Sample Rate is simple as it is the incoming sample rate and it should be set to "samp_rate" by default. Sample Rate should be set to the incoming Sample Rate even if Decimation is used to modify the output sample rate. The "Cutoff Frequency" is the bandwidth (Channel Size) of the transmission extending out from the center frequency. "Channel Spacing" is specified by the "channel_spacing" variable block and this value should be configured with the name of this variable block.

"Transition Width" is one of the parameter values that require a bit of testing to configure correctly.[8] Research and experience will help determine where to initially set this value depending on center frequency, bandwidth, and radio type. This may be a bit confusing, however consider the radio signal itself. The transmission signal is not always going to be exactly centered on the intended center frequency. Many things are going to impact this signal: weather, other signals, power, etc. Therefore a radio has to compensate for anomalies to ensure that "all" of the signal is received. For research and assessment purposes we will describe the Transition Width as a value that is used by the LPF to reduce the edges of the signal slowly so that issues, such as atmospheric jitter, are accounted for when receiving the signal. Using this logic, a steep transition will isolate the center frequency very well but, depending on conditions, it may generate some signal and data loss. On the other hand, a slow transition will include too much extra signal and ruin the ultimate goal of signal isolation. From our experiences, InGuardians

[6] Nyquist–Shannon sampling theorem:http://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem
[7] Blind singal analysis with GNU Radio: https://www.youtube.com/watch?v=pltmBkQSy7w
[8] The actual reasoning behind the Transition Width may not be accurately represented in this document. The information presented here is for logically configuring this parameter quickly during research and assessment activities. See Disclaimer in Introduction.

has determined that setting the "Transition Width" to a value between forty (40) and fifty (50) percent of the Channel Spacing results in an output signal that works the best in the follow on demodulation blocks.

Figure 9 shows the captured signal leveraging the following inputs for the LPF that will be used in this example.

- Frequency Offset: 120,000
- Channel Spacing: 200,000
- Channel Transition: 80,000
- Window: Blackman

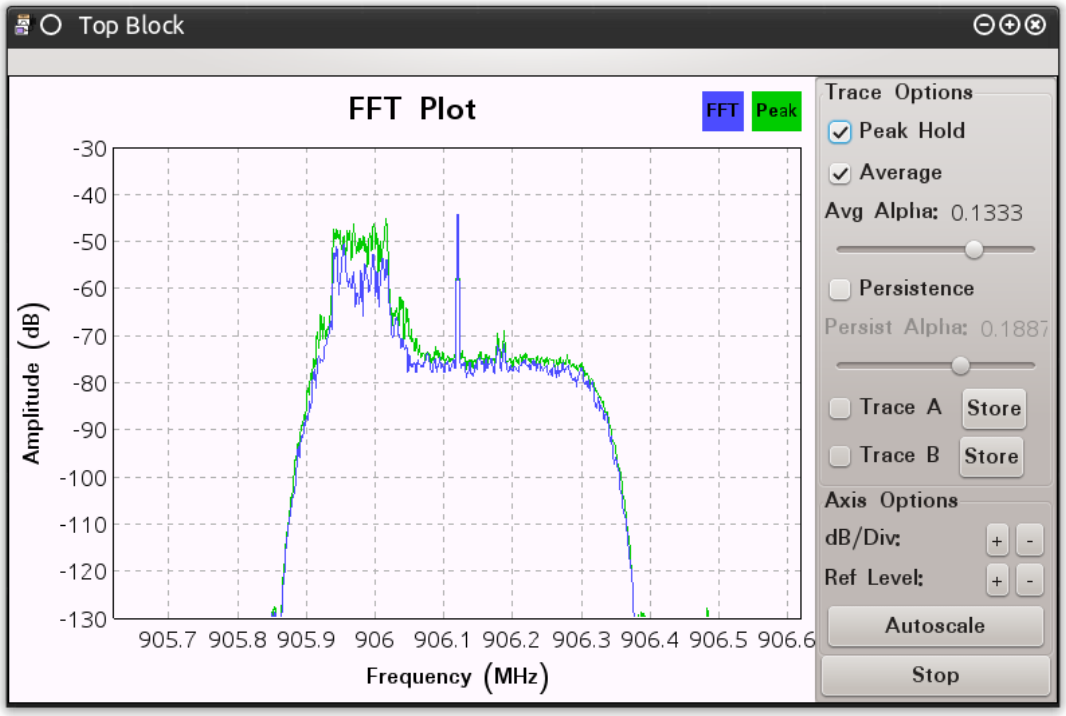
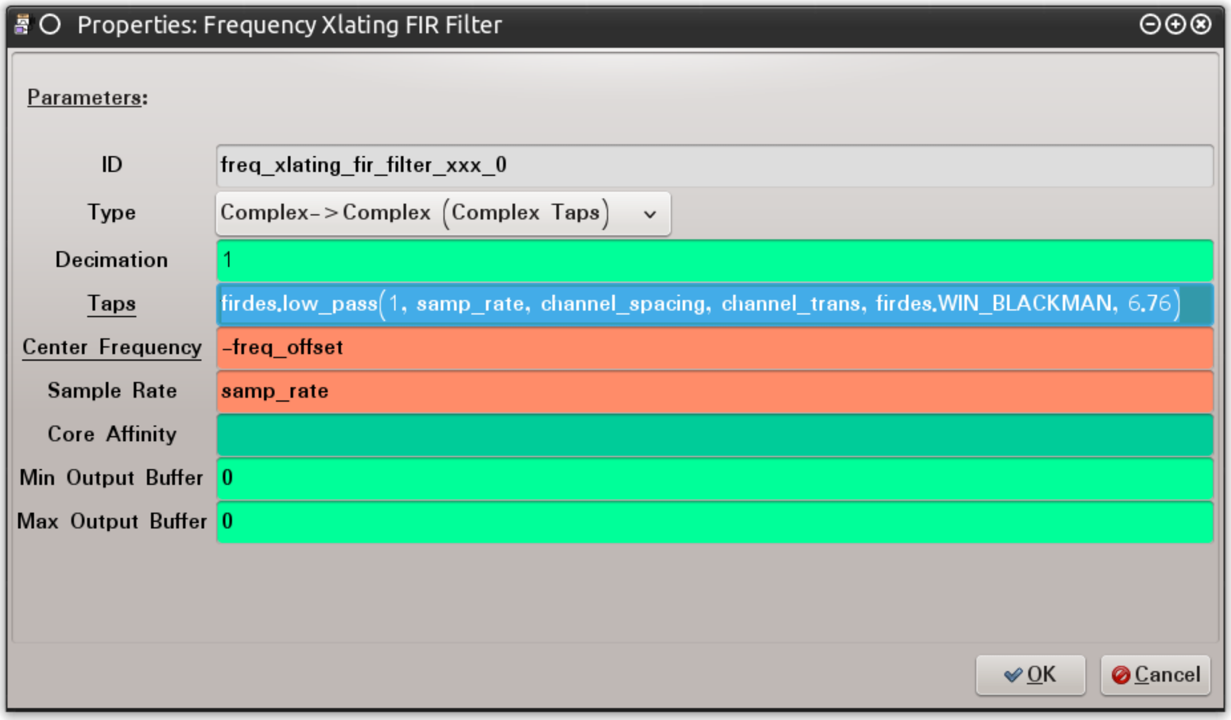


Figure 9 GRC Top Block Using Low Pass Filter

While isolated this signal is not located at the center of the FFT Plot. This situation, if left unaddressed, will negatively impact the rest of the demodulation efforts. For proper demodulation this transmission needs to be centered. Currently the radio is configured so that the DC Spike is located at the center frequency and thus requiring that the signal be adjusted so that the transmission is located at center frequency. Using mathematics it is possible to implement several GRC blocks to make this modification. However, GRC has included the "Frequency Xlating FIR Filter" (FXFF) block to incorporate this action.

The core variables that configure the LPF also properly configure the FXFF block. The FXFF has "Decimation" and "Sample Rate" like the LFP and they are configured the same way. The FXFF block also has the "Center Frequency" variable. This is the variable that re-centers the signal as adjusted for the DC Offset. If no DC Offset was used then this value should be left as zero(0) else it is configured with the "freq_offset". The final setting to configure is the "Taps" variable. InGuardians uses the guidance provided by Dragorn in his blog post "Playing with the HackRF – Keyfobs." This blog post provides an equation for this variable that uses the "Window," "Channel Width," and "Transition Width" parameters in the same manner as they are configured in the LPF. Configure the Taps value with the settings used in this example, as shown in Figure 10, results in the following equation: "*firdes.low_pass(1, samp_rate, channel_spacing, channel_trans, firdes.WIN_BLACKMAN, 6.76)*."


Properties: Frequency Xlating FIR Filter

Parameters:

ID	freq_xlating_fir_filter_xxx_0
Type	Complex->Complex (Complex Taps)
Decimation	1
Taps	firdes.low_pass(1, samp_rate, channel_spacing, channel_trans, firdes.WIN_BLACKMAN, 6.76)
Center Frequency	-freq_offset
Sample Rate	samp_rate
Core Affinity	
Min Output Buffer	0
Max Output Buffer	0

Figure 10 GRC FXFF Block Configuration

Running with this configuration isolates the signal and centers the transmission as desired. This is shown in Figure 11 which demonstrates that the signal is centered and ready for demodulation.

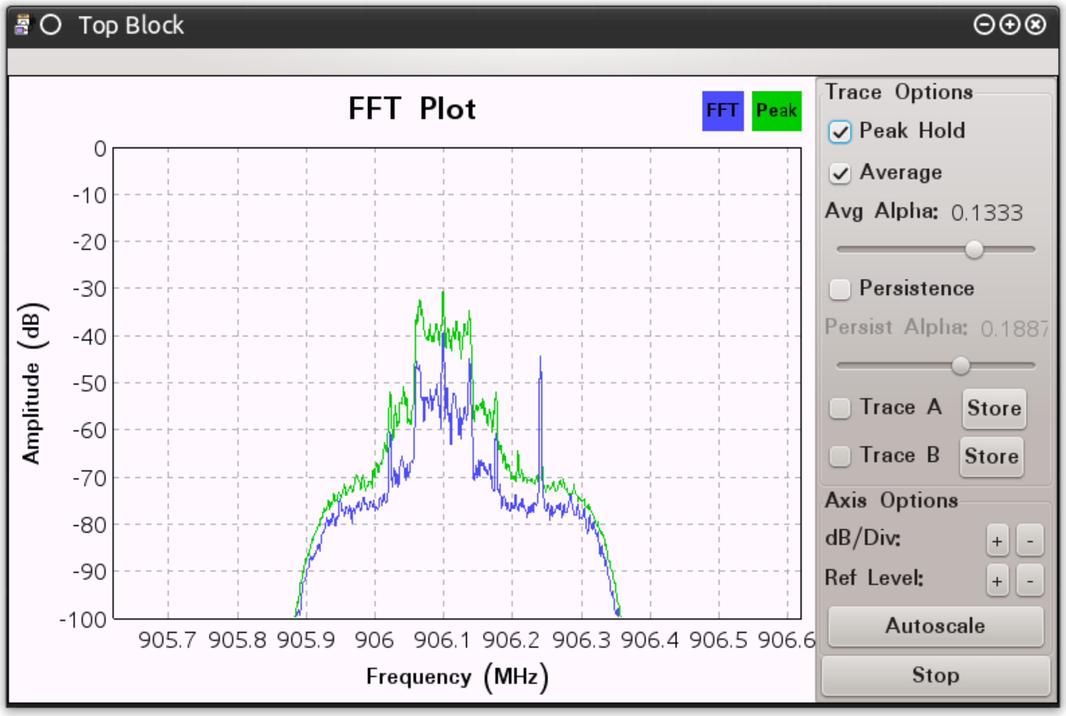
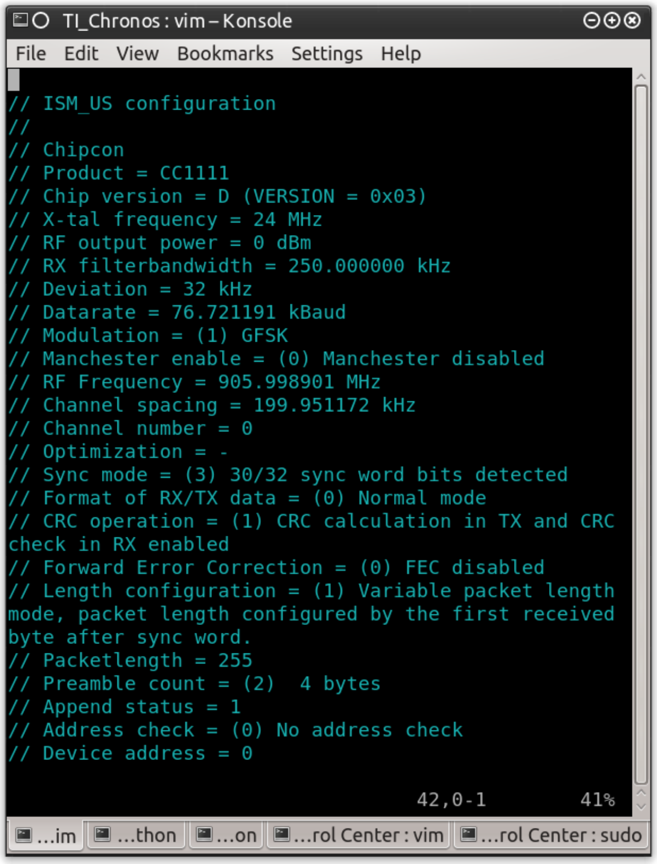


Figure 11 GRC Top Block Using FXFF Block

5.0 Actual Demodulation

Isolateingthe signal with the LPF or FFX allows for the effective demodulation of the signal. Mike Ossmann explains the concepts and math behind how to proceed with demodulation from this point in his radio analysis classes. He provides the specific reasonings behind the blocks necessary for demodulating different tytes of frequency-shify key (FSK) and amplitude-shift key (ASK) modulation. The basics are this: the "Complex to Mag" or "Complex to Mag ^ 2" blocks are used to demodulate ASK transmissions and the "Quadrature Demod" block is used to demodulate FSK transmissions (specifically 2FSK and Gaussian FSK (GFSK)).

The signals generated for this example were transmitted using GFSK modulation. This was determined by researching the devices that transmit and receive the captured signal. The transmissions shown in the previous images are the results of a Texas Instrument (TI) Chronos Watch communicating with the TI Chronos black dongle. The TI Chronos Dongle has a TI Chipcon CC1111 radio. The datasheet and even the source code for the dongle are available on TI's website. Often times the radio's datasheet is sufficient to provide the extra information we need. Source code is usually better because every radio can be configured in many different ways. A vendor will configure the radios in their products to perform optimally for their primary functionality. Datasheets narrow the possibilities while source code and radio configuration settings provide specifics. In the case of the TI Chronos Dongle a quick review of the source code produces the values used to configure the radio for interaction with the TI Chronos Watch as seen in Figure 12.

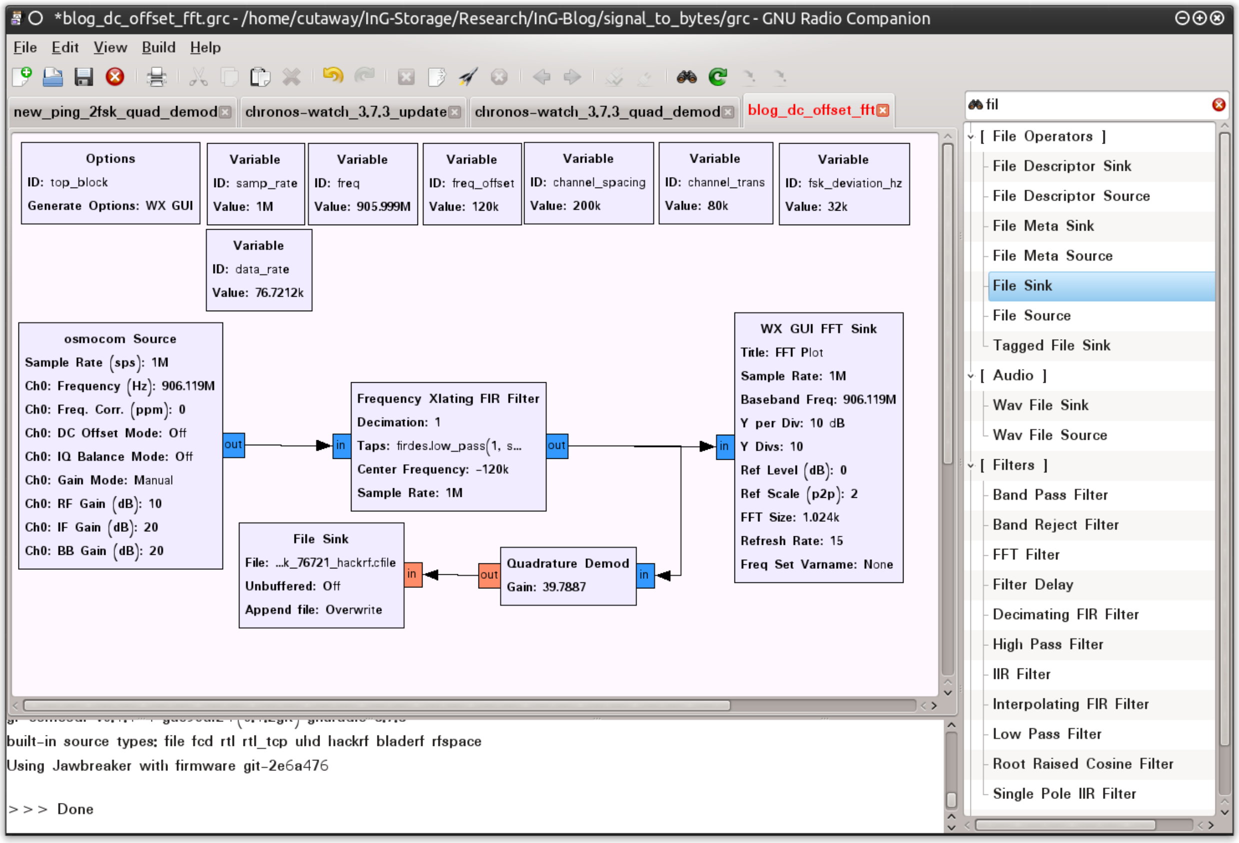

TI_Chronos : vim – Konsole
File Edit View Bookmarks Settings Help
...im
...thon
...on
...rol Center : vim
...rol Center : sudo

```
// ISM_US configuration
//
// Chipcon
// Product = CC1111
// Chip version = D (VERSION = 0x03)
// X-tal frequency = 24 MHz
// RF output power = 0 dBm
// RX filterbandwidth = 250.000000 kHz
// Deviation = 32 kHz
// Datarate = 76.721191 kBaud
// Modulation = (1) GFSK
// Manchester enable = (0) Manchester disabled
// RF Frequency = 905.998901 MHz
// Channel spacing = 199.951172 kHz
// Channel number = 0
// Optimization = -
// Sync mode = (3) 30/32 sync word bits detected
// Format of RX/TX data = (0) Normal mode
// CRC operation = (1) CRC calculation in TX and CRC
check in RX enabled
// Forward Error Correction = (0) FEC disabled
// Length configuration = (1) Variable packet length
mode, packet length configured by the first received
byte after sync word.
// Packetlength = 255
// Preamble count = (2)  4 bytes
// Append status = 1
// Address check = (0) No address check
// Device address = 0

                                    42,0-1         41%
```

Figure 12 TI Chronos Dongle Source Code Radio Configuration Settings

Reviewing these settings shows some of the configuration values that have already been used in this example's block variables. Configuration of the rest of the GRC blocks, in this example, will leverage these values. For now demodulation only requires knowing the modulation type and "Deviation." The modulation type, as mentioned, is GFSK. The other value, which will be used to configure the "Quadrature Demod" block is the Deviation.

When using the "Quadrature Demod" block it requires the "Gain" variable to be configured correctly. By default this block is preconfigured with the following equation to determine Gain: *samp_rate/(2*math.pi*fsk_deviation_hz/8.0)*. This requires a new variable block with the name of "fsk_deviation_hz". This variable block should be configured with the value "32000" as defined for Deviation in the source code. Figure 13 shows these modifications with the addition of the "Quadrature Demod" and "fsk_deviation_hz" variable blocks.



Options
ID: top_block
Generate Options: WX GUI

Variable
ID: samp_rate
Value: 1M

Variable
ID: freq
Value: 905.999M

Variable
ID: freq_offset
Value: 120k

Variable
ID: channel_spacing
Value: 200k

Variable
ID: channel_trans
Value: 80k

Variable
ID: fsk_deviation_hz
Value: 32k

Variable
ID: data_rate
Value: 76.7212k

osmocom Source
Sample Rate (sps): 1M
Ch0: Frequency (Hz): 906.119M
Ch0: Freq. Corr. (ppm): 0
Ch0: DC Offset Mode: Off
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Manual
Ch0: RF Gain (dB): 10
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20

Frequency Xlating FIR Filter
Decimation: 1
Taps: firdes.low_pass(1, s...
Center Frequency: -120k
Sample Rate: 1M

WX GUI FFT Sink
Title: FFT Plot
Sample Rate: 1M
Baseband Freq: 906.119M
Y per Div: 10 dB
Y Divs: 10
Ref Level (dB): 0
Ref Scale (p2p): 2
FFT Size: 1.024k
Refresh Rate: 15
Freq Set Varname: None

File Sink
File: ...k_76721_hackrf.cfile
Unbuffered: Off
Append file: Overwrite

Quadrature Demod
Gain: 39.7887

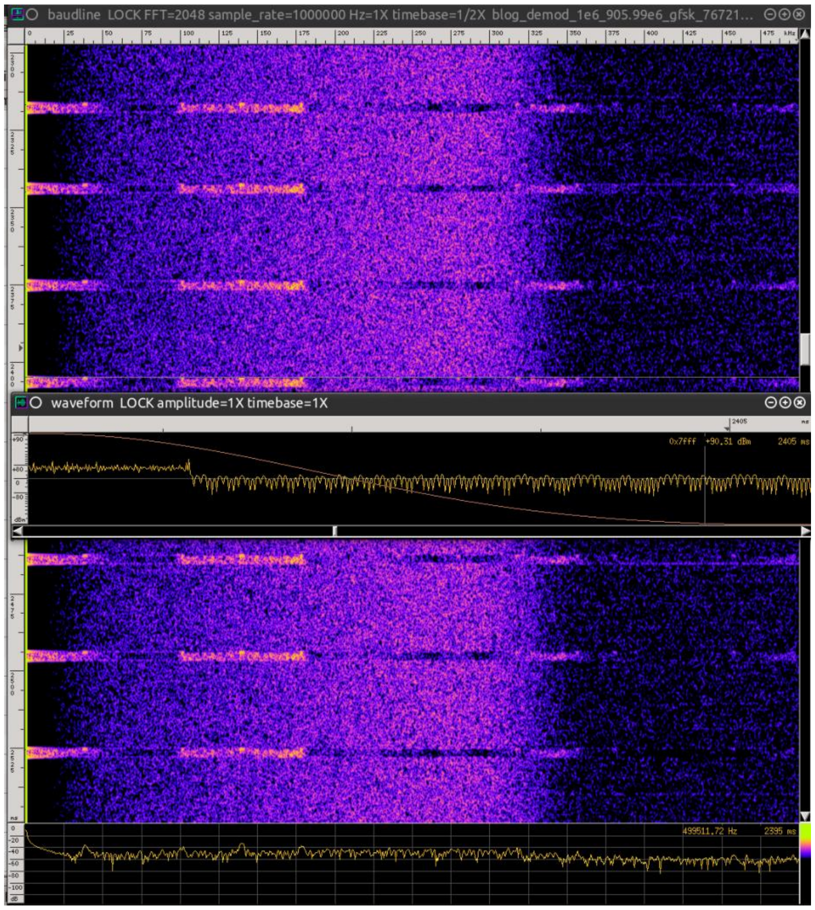
```
built-in source types: file fcd rtl rtl_tcp uhd hackrf bladerf rfspace
Using Jawbreaker with firmware git-2e6a476

>>> Done
```

Figure 13 GRC Quadrature Demod Configuration

As shown in Figure 13 the "Quadrature Demod" block is outputting to a "File Sink" block. This could be set to output to another "FFT" block but signal displayed during an active demodulation will not provide any useful information. For continued analysis of the demodulated signal, other wave analysis tools are necessary. But before discussing these tools some guidance about naming output files should be covered. There are several things that should be taken into consideration when outputting a signal to a file. The first is that wave file generated by capturing a radio transmission can get very large very fast. Capturing straight from the HackRF, or any other radio, to a file with a sample rate of 1,000,000 will generate approximately one gigabyte every forty (40) seconds. This file size increases dramatically as the sampling rate is increased. For these instances, writing to the hard drive fast enough to capture all of transmission needs to be considered. To increase performance capturing straight a file in the "/dev/shm" directory, provided by Linux operating systems, may be necessary. The "/dev/shim" directory is RAM Disk and it can be written to faster than the hard drive (system dependent, of course). In this case, however, demodulated data does not require the same considerations for write speeds although some attention should be paid to file size to ensure that the primary file system or "/dev/shm" is not filled.

In addition to write speeds and file sizes, special consideration should be given to the naming of the file. When capturing a radio transmission a record should be made of the capture settings. Without these settings, future analysis or replaying the data will not be possible. To address this, I note all of the settings that are important for analysis in the file name. For this example the file name contains references to demodulated data, the Sample Rate, the center frequency, the modulation type, and the data rate: "/tmp/blog_demod_1e6_905.99e6_gfsk_76721_hackrf.cfile."

Running the GRC script shown in Figure 13 will capture the data transmitted on the desired frequency, demodulate the transmission, and output the demodulate signal to a file. The resulting signal can be analyzed using spectrum analysis tools such as Baudline.[9] Pulling a file into Baudline has already been explained very well by Dragorn in his Keyfob blog post (see the post's comments for excellent additional information provided by the developer of Baudline). Figure 14 shows how the the captured demodulated signal is displayed in Baudline's primary FFT and the Waveform display.


baudline LOCK FFT=2048 sample_rate=1000000 Hz=1X timebase=1/2X blog_demod_1e6_905.99e6_gfsk_76721...

waveform LOCK amplitude=1X timebase=1X

Figure 14 Demodulated Capture Displayed in Baudline

The window in the background is Baudline's primary FFT. This window is actually a waterfall display that shows the demodulated signal over time. The Waveform display, center and in the foreground, shows the demodulated signal at a particular moment in time. The Baudline tool allows for the analysis of the

[9] Baudline: http://www.baudline.com/

demodulated signal before it is passed other GRC blocks that will determine the ones and zeros from this information. At this point the wave shown in the Waveform display should be the transmission's data. Figure 15 is a close look at this wave.

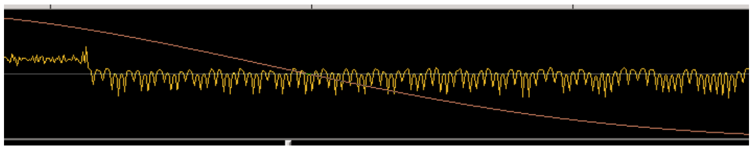


Figure 15 Demodulate Capture Displayed in Waveform Display

Figure 15 provides several pieces of valuable information about the demodulated signal. The first observation is that the signal is not clean enough for additional analysis. It does not appear to be a nicely formed wave with smooth transitions. The next observation is that the waveform is shifted down slightly and does not cross the center line (X-axis) optimally. A clean centered signal is necessary for the GRC blocks designed to convert this information to 0's and 1's to function properly. Hardware radios do this "cleaning" process using the circuitry designed into the radio component. Radios implement a LPF after the demodulation before converting the transmission into data.

Therefore, to convert the data properly, the GRC script needs to include another LPF in the same manner as a radio component. This is done by placing a LPF block between the Quadrature Demodulation and File Sink blocks. As mentioned in the earlier explanation of the LPF, this block needs values for the "Cutoff Freq" and "Transition Width" variables. The issue with implementing this LPF is that there is no good documentation for how to configure these values. The initial values are learned through research and experience and will more than likely be dependent on the frequency, modulation, and other radio configurations. For the "Cutoff Freq" InGuardians usually starts with a value of 100,000 and then selects a "Transition Width" about half of that value. Once configured the transmission is captured again or replayed so that it is processed through the demodulator and the new LPF block. This is done several times. Each time the results are reviewed using Baudline. This process is repeated until the results show a wave pattern with nice transitions that look like data. Figure 16 shows this state which was accomplished by using 80,000 for the "Cutoff Freq" and 50,000 for the "Transition Width."

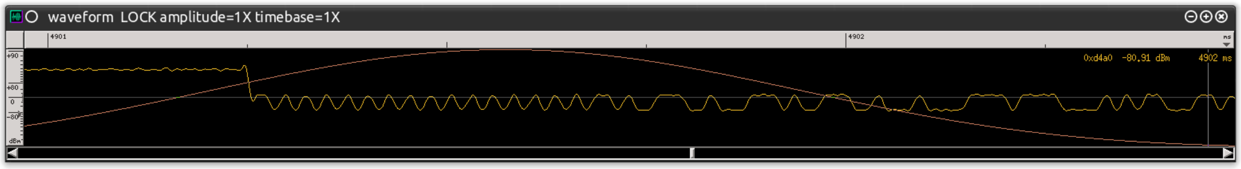


Figure 16 Demodulated Signal Run Through a LPF

The waveform in Figure 16 does appear to have nice smooth transitions that can visibly be converted to 0's and 1's. Taking a closer look at this signal it appears that the waveform is not centered on the X-axis. To translate the data properly GRC needs this signal to be centered, as much as possible, on the X-axis. Modifying this waveform is easy with simple mathematics. One form of modification that can be performed on this waveform is to multiply every point on this wave by a constant number. However, multiplying each point will equally increase the positive or negative value of that point and thus increase the amplitude of the wave pattern. Multiplication will not shift the wave pattern up or down. Adding a constant value to each point will uniformly modify the Y-axis value of the waveform and shift the wave up. In contrast, subtracting a constant value from each point on the wave will shift it down. The actual value needed to shift a waveform properly is determined via experimentation and observation and implemented using an "Add Constant" block. The results of shifting this wave pattern by adding a constant value of 6 is shown in Figure 17.

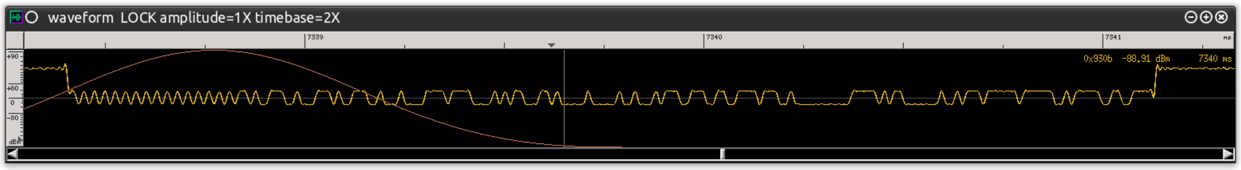

waveform LOCK amplitude=1X timebase=2X
7339
7340
7341
ms
0x930b -88.91 dBm 7340 ms
dBm

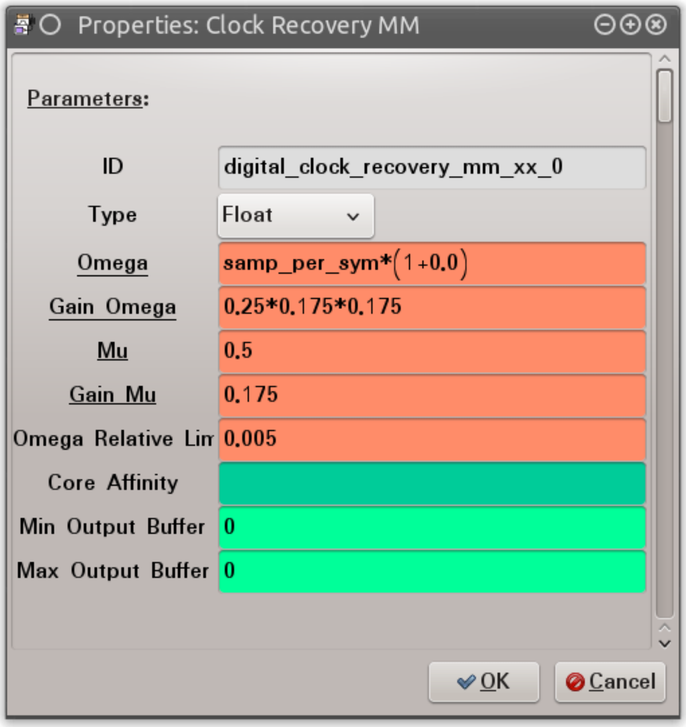
Figure 17 Demodulated Signal Shifted Up to X Axis

Knowing "how" to manipulate the wave pattern without losing any data could be helpful when analyzing other signals. For instance, if the manufacturer documentation documents that the transmitted data is being transmitted inverted (meaning a 0 is a 1 and a 1 is a zero) the signal could be re-inverted by multiplying the signal by a constant value of negative one (-1).

6.0 Counting the Bits

Once a clean demodulated signal has been accomplished, the next step is to analyzed the resulting waveform and detect the highs and lows (0's and 1's). GRC contains two blocks that, in combination, are specifically designed to detect consistent high and low transitions and convert them into 0's and 1's. These two specific blocks are the "Clock Recovery MM" and "Binary Slicer" blocks. The Clock Recovery MM block does the magic of discerning highs and lows. The Binary Slicer marks the highs as a "1" and the lows as a "0". Actually, it marks them as "0x01" or "0x00" which requires translation into actual bytes. The first step is to concentrate on the configuration of the Clock Recovery MM block.

The Clock Recovery MM, shown in Figure 18, has some REALLY scary looking variables. However, the configuration of these variables is actually easy. The "Gain Omega," "Mu," "Gain Mu," or "Omega Relative Limit" variables come preconfigured with very specific values that, generally, do not need to be modified. Leave these settings their default. The parameter that needs to be updates is the "Omega" variable. This variable also comes with a default setting: *samp_per_sym*(1+0.0)*. This means that this parameter needs to be configured with another variable block labeled "samp_per_sym", which equates to "Samples Per Symbol."

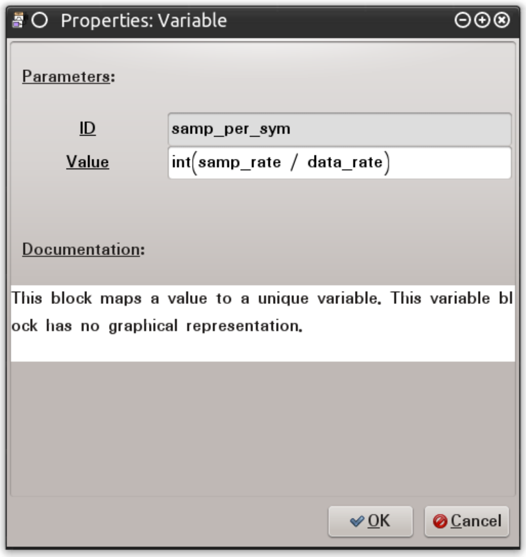

Properties: Clock Recovery MM

Parameters:

ID	digital_clock_recovery_mm_xx_0
Type	Float
Omega	samp_per_sym*(1+0.0)
Gain Omega	0.25*0.175*0.175
Mu	0.5
Gain Mu	0.175
Omega Relative Lim	0.005
Core Affinity	
Min Output Buffer	0
Max Output Buffer	0

Figure 18 GRC Clock Recovery MM Block Configuration

Thus far, for this example, the "Samples Per Symbol" has not been specifically computed. The values necessary to compute this value have been identified. The value for "samp_rate" is actually "Samples per Second". For this capture the "samp_rate" is 1,000,000 samples per second. The "Data Rate" from the radio configuration file represents "Symbols Per Second." A GRC variable block named "data_rate" can be configured with the value 76721.191 from the software example in Figure 12. Simple mathematics modifies the "Samples Per Symbol" reference to "Samples Per Second Divided by Symbols Per Second." The "Second" values cancel out and the resulting equation can be written as "samp_rate/data_rate" which, in this example, equates to "1,000,000 / 76721.191". Creating a "samp_per_sym" variable block with the entry “int(samp_rate / data_rate)” is shown in Figure 19. This will automatically calculate the “Samples Per Symbol” and make it an integer for the “Clock Recover MM” block.



Parameters:

ID: samp_per_sym

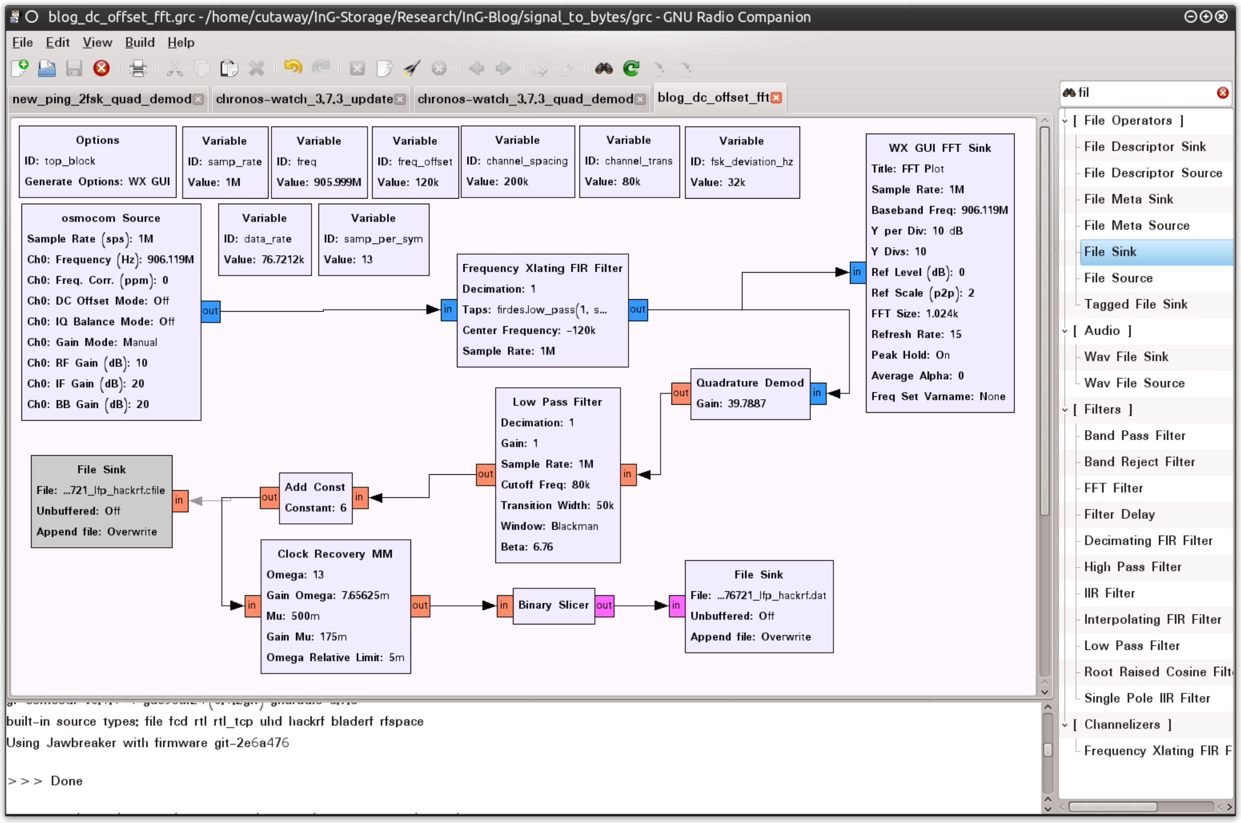
Value: int(samp_rate / data_rate)

Documentation:

This block maps a value to a unique variable. This variable block has no graphical representation.

Figure 19 GRC "samp_per_sym" Variable Block

With the variables for the “Clock Recovery MM” block configured this block can identify the highs and lows. This information is passed to the Binary Slicer block and the resulting zeros and ones (0’s and 1’s) are written to a file. Figure 19 shows the current GRC setup using these two blocks.



Options
ID: top_block
Generate Options: WX GUI

Variable
ID: samp_rate
Value: 1M

Variable
ID: freq
Value: 905.999M

Variable
ID: freq_offset
Value: 120k

Variable
ID: channel_spacing
Value: 200k

Variable
ID: channel_trans
Value: 80k

Variable
ID: fsk_deviation_hz
Value: 32k

osmocom Source
Sample Rate (sps): 1M
Ch0: Frequency (Hz): 906.119M
Ch0: Freq. Corr. (ppm): 0
Ch0: DC Offset Mode: Off
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Manual
Ch0: RF Gain (dB): 10
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20

Variable
ID: data_rate
Value: 76.7212k

Variable
ID: samp_per_sym
Value: 13

Frequency Xlating FIR Filter
Decimation: 1
Taps: firdes.low_pass(1, s...
Center Frequency: -120k
Sample Rate: 1M

WX GUI FFT Sink
Title: FFT Plot
Sample Rate: 1M
Baseband Freq: 906.119M
Y per Div: 10 dB
Y Divs: 10
Ref Level (dB): 0
Ref Scale (p2p): 2
FFT Size: 1.024k
Refresh Rate: 15
Peak Hold: On
Average Alpha: 0
Freq Set Varname: None

Quadrature Demod
Gain: 39.7887

Low Pass Filter
Decimation: 1
Gain: 1
Sample Rate: 1M
Cutoff Freq: 80k
Transition Width: 50k
Window: Blackman
Beta: 6.76

File Sink
File: ...721_lfp_hackrf.cfile
Unbuffered: Off
Append file: Overwrite

Add Const
Constant: 6

Clock Recovery MM
Omega: 13
Gain Omega: 7.65625m
Mu: 500m
Gain Mu: 175m
Omega Relative Limit: 5m

Binary Slicer

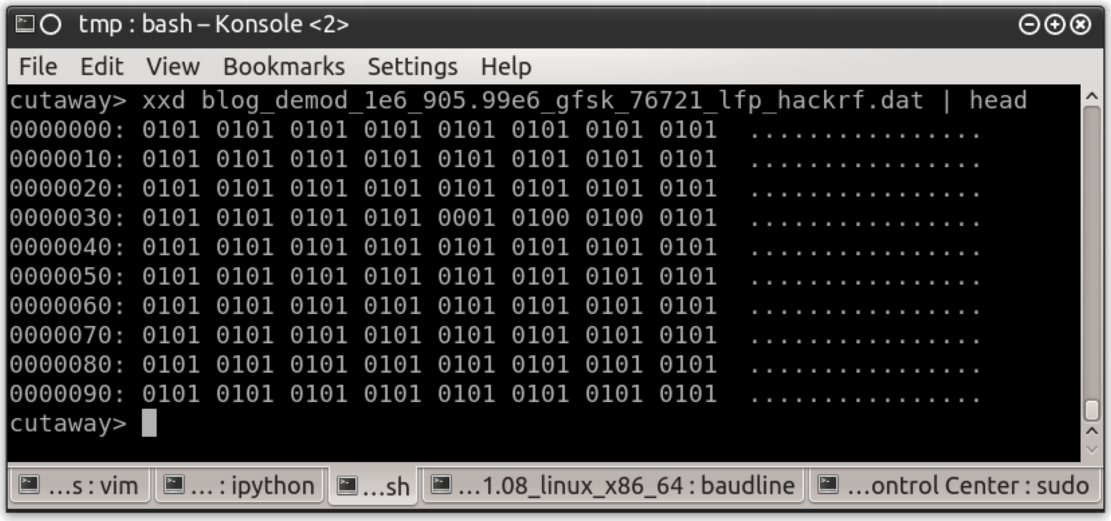
File Sink
File: ...76721_lfp_hackrf.dat
Unbuffered: Off
Append file: Overwrite

```
built-in source types: file fcd rtl rtl_tcp uhd hackrf bladerf rfspace
Using Jawbreaker with firmware git-2e6a476

>>> Done
```

Figure 20 GRC with Clock Recovery MM To Extract Data

7.0 ANALYZING DEMODULATED DATA

Running the GRC script with the Clock Recovery MM and Binary Slicer blocks, shown in Figure 20, will write the transmitted data to a file. Figure 21 shows how this data looks when the contents of the file are reviewed using the Linux "xxd" command or a Hex Editor on any operating system.



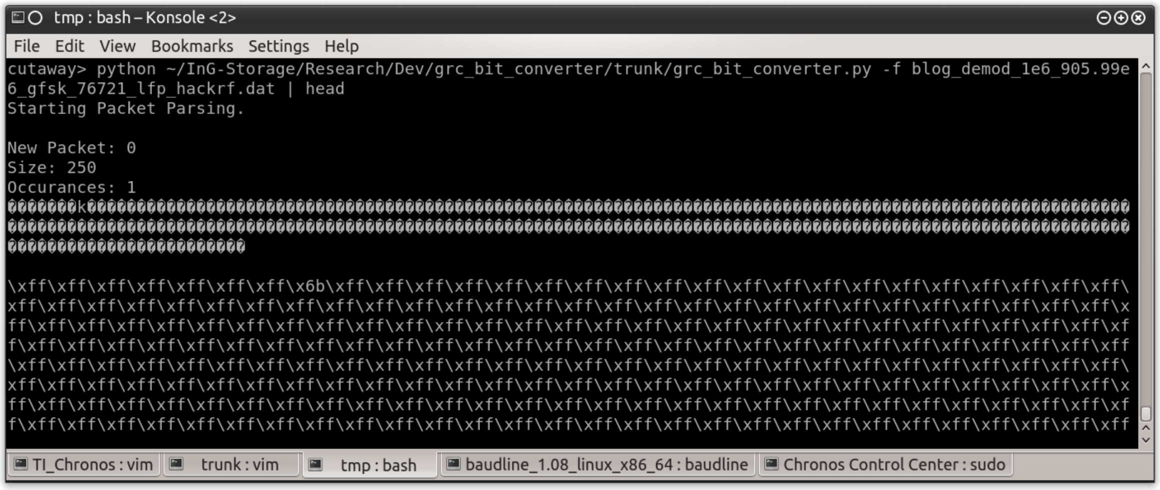
```
cutaway> xxd blog_demod_1e6_905.99e6_gfsk_76721_lfp_hackrf.dat | head
0000000: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0000010: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0000020: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0000030: 0101 0101 0101 0101 0001 0100 0100 0101  ................
0000040: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0000050: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0000060: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0000070: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0000080: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0000090: 0101 0101 0101 0101 0101 0101 0101 0101  ................
cutaway> 
```

Figure 21 Demodulated Data Viewed Using "xxd"

As mentioned, the Binary Slicer block outputs a single byte for each low and high value detected by the Clock Recovery MM block. Thus the output file is a file full of 0's and 1's, literally, as eight (8) bits are used to represent one bit of actual data. The next step is to squash each these bits into a byte. The first line "0101 0101 0101 0101 0101 0101 0101 0101" actually relates to "0b1111111111111111" or "0xff". The line starting at offset 0x30 is "0101 0101 0101 0101 0001 0100 0100 0101" or "0b1111111101101011" or "0xff6b".

This output is very promising and could be the data transmitted by the radio, but anything can spit out 0's and 1's. Detecting actual data is the challenge. From experience, InGuardians can predict that the information represented in Figure 21 is very like just noise floating through the air. This occurs because the "Clock Recovery MM" block functioned properly at this moment in time of the transmission. It looked at the incoming signal, made an estimation as to whether that signal was high or low at that moment in time and passed that information to the "Binary Slicer" which converted "High" and "Low" into "1" and "0". In this case there is a lot more noise out there than there is actual signal. The GRC script has not been configured so that it can tell the difference between the transmission of the TI Chronos Watch, the signal from another radio, or just noise in the air. Another issue to consider is whether or not any GRC block will understand the starting bit of the data. In other words, which bit is the detected low or high in a byte? Is it the Most Significant Bit? Could it be the third bit from the Least Significant Bit. The data represented in the output data file, and Figure 21, is just raw data.

Parsing the data in this output file will be necessary to combine each of the bits in the file into a byte. InGuardians is not aware of any GRC blocks that will automatically perform this action. Even if there were a block it would be difficult to manage the block to account for a wide variety of conditions and data parsing tricks. To assist with combining the bits output by the Clock Recovery MM and Binary Slicer block combination, InGuardians has developed GRC Bit Converter (grc_bit_converter.py)[10]. This python script provides a variety of functions to manipulate the data file including outputting data packets from the combined bits, as seen in Figure 22.



```
cutaway> python ~/InG-Storage/Research/Dev/grc_bit_converter/trunk/grc_bit_converter.py -f blog_demod_1e6_905.99e
6_gfsk_76721_lfp_hackrf.dat | head
Starting Packet Parsing.

New Packet: 0
Size: 250
Occurances: 1
�������k�������������������������������������������������������������������������������������������������������������
��������������������������������������������������������������������������������������������������������������������
�������������������������

\xff\xff\xff\xff\xff\xff\xff\x6b\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\x
ff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xf
f\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\x
ff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xf
f\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
```

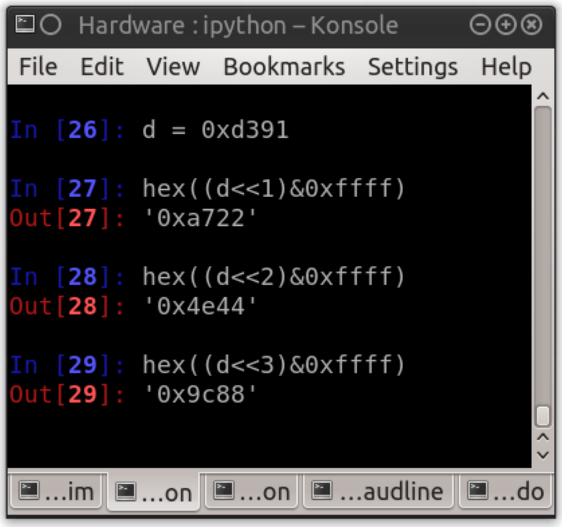
tmp : bash – Konsole <2>
File Edit View Bookmarks Settings Help
TI_Chronos : vim
trunk : vim
tmp : bash
baudline_1.08_linux_x86_64 : baudline
Chronos Control Center : sudo

Figure 22 Processing Radio Data using "bit_convert.py"

The "grc_bit_converter.py" script run with the default settings just starts at the beginning of the file, pulls out the first 2000 (250 * 8) bits, combines them into bytes, and prints the results in ASCII representation and byte format. The script considers these 250 bytes to be a packet and continues to process all packets until the end of the file. Each of these packets is marked with a packet number for easy identification. The "Occurrences" value shows how many packets contained the exact same data. This value will be very useful when data packets are identified in the file.

Once the transmitted data is translated into bytes the next thing to consider is how the transmitted data is formatted to help search for it. Most radios (not all but most) will begin a transmission with some type of preamble followed by a SyncWord. Preambles are generally a series of high to low transmissions. When viewed in binary it will appear as "0b1010101010101010" or "0xaaaa". The number of preamble bytes depends on the radio and how it is configured. As it cannot be predetermined exactly which bit is the first bit (or if the implementer has inverted the data) the preamble may appear as "0101010101010101" or "0x5555" in the parsed file. The job of a preamble is to let the radio know that there is incoming data that should be processed. The job of a SyncWord is to tell the radio where the data packet begins. It can also be used as a designator between two different networks. Any value can be picked for the SyncWord with the obvious exceptions being those that closely resemble a preamble. The most common SyncWord that is used by default by many radios is "0xd391". Like the preamble the SyncWord might appear as a different value in the parsed data file. One way to determine how the SyncWords might appear in this file is to shift the bits of "0xd391" using iPython. Figure 23 shows that the SyncWord it may actually appear as 0xa722, 0x4e44, or 0x9c88 when reviewing the information in this data file.

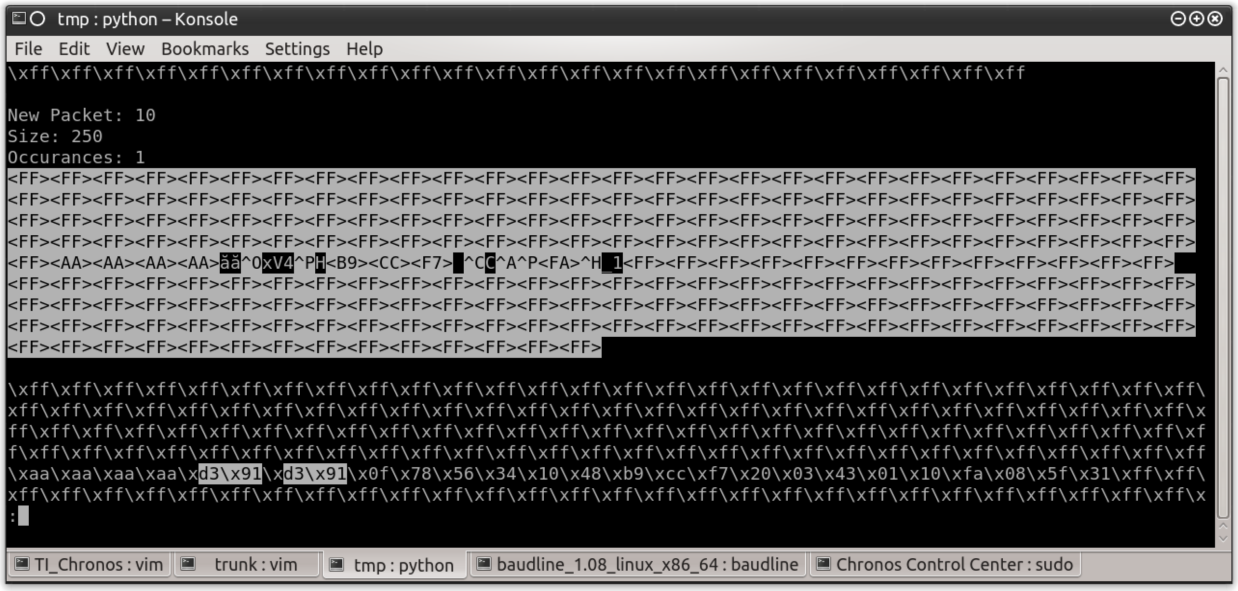
[10] GRC Bit Converter: https://github.com/cutaway/grc_bit_converter



```
In [26]: d = 0xd391

In [27]: hex((d<<1)&0xffff)
Out[27]: '0xa722'

In [28]: hex((d<<2)&0xffff)
Out[28]: '0x4e44'

In [29]: hex((d<<3)&0xffff)
Out[29]: '0x9c88'
```

Figure 23 SyncWord Shifted By One, Two, and Three Bits

Locating transmitted data packets is easy using these SyncWord values.. A quick analysis technique is to just output the converted data to "less" and then search for the known SyncWord, 0xd391, using "d3\\x91". Figure 24 shows the results of this search and one of the transmitted packets.



```
\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff

New Packet: 10
Size: 250
Occurances: 1
<FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF>
<FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF>
<FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF>
<FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF>
<FF><AA><AA><AA><AA>ăă^OxV4^PH<B9><CC><F7> ^CC^A^P<FA>^H_1<FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF>
<FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF>
<FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF>
<FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF>
<FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF><FF>

\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\x
ff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xf
f\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
\xaa\xaa\xaa\xaa\xd3\x91\xd3\x91\x0f\x78\x56\x34\x10\x48\xb9\xcc\xf7\x20\x03\x43\x01\x10\xfa\x08\x5f\x31\xff\xff\
xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\x
:
```

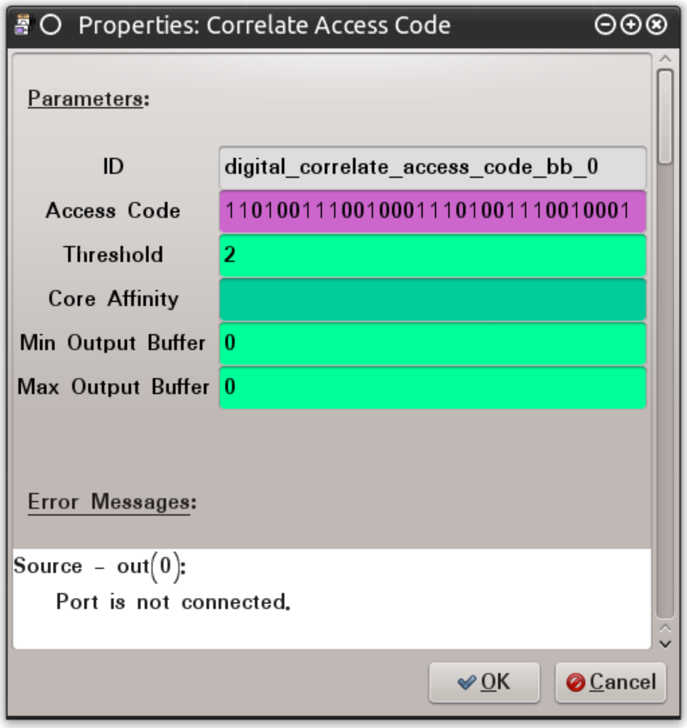
Figure 24 SyncWord Search In Data Output by "grc_bit_conveter.py"

This information definitely looks like a data packet. This can be confirmed with other information by looking back at the source code for the TI Chronos Dongle shown in Figure 12. The source code specifically states that the Preamble count is two (2) which is equal to four (4) bytes. The Preamble of this packet is "\xaa\xaa\xaa\xaa" which is also four (4) bytes. The source code states that the Sync mode is configured to detect 30 bits of a 32 bit SyncWord. The SyncWord of this packet is "\xd3\x91\xd3\x91" which is 32 bits. The source code states that the packet length is variable and defined by the first byte after the SyncWord. The first byte after the SyncWord of the displayed packet is "\x0f" which equates to fifteen (15). From this byte counting out fifteen bytes and leaves two remaining bytes before the bit stream reverts to all 1's or 0xff. These last two bytes are the Cyclic Redundancy Check (CRC) for the packet which, according to the source code, has been enabled.

8.0 MARKING THE PACKET

Manual or scripted parsing of the data file will eventually locate transmitted data packets When information is limited, this may be the only way to handle the data. But in this example there is enough information to perform additional steps using GRC. Even in cases where there is a lack of information logical assumptions can be made from data review, to perform the following actions. These actions are to mark each of the data packets as they is output from the Binary Slicer before they are placed into the output data file GRC.

Marking packets in GRC is accomplished using the "Correlate Access Code" block. This block will take two values: "Access Code" and "Threshold". The Access Code is a sequence of bits that indicates the beginning of a packet. This block will use this value to monitor the data passing out of the "Binary Slicer" block. If the Correlate Access Code block sees the Access Code sequence of bits in the byte stream it will mark the next byte it processes by flipping the second bit of the byte. Remember, the information coming out of the "Binary Slicer" is a byte of data for each 0 and 1. By flipping the second bit of this byte the byte is converted to a 2 or 3, respectively. Since the Least Significant Bit does not change the data is unmodified and remains intact for analysis and conversion. The "Threshold" variable merely indicates the number of bits that can be "different" from the actual value provided in the "Access Code". This is done in case an incoming packet is slightly corrupted but can still be processed thus allowing other means to manage transmission errors. Figure 25 shows the "Correlate Access Code" configured with the 0xd391d391 bit sequence and a "Threshold" of two (2) as indicated in the source code.


Properties: Correlate Access Code

Parameters:

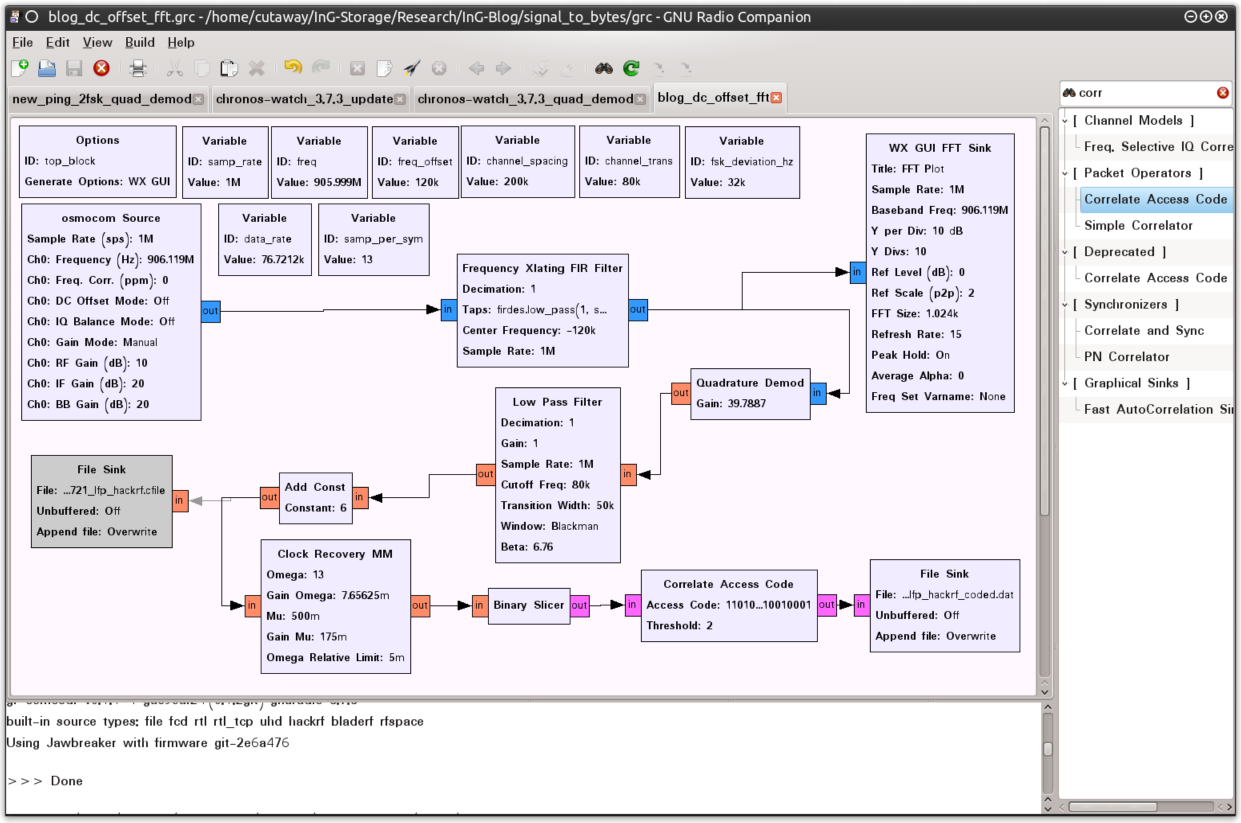
ID	digital_correlate_access_code_bb_0
Access Code	11010011100100011101001110010001
Threshold	2
Core Affinity	
Min Output Buffer	0
Max Output Buffer	0

Error Messages:

```
Source - out(0):
    Port is not connected.
```

Figure 25 Correlate Access Code Configured with 0xd391d391

As mentioned, the "Correlate Access Code" block processed the data exiting the "Binary Slicer" block. Figure 26 shows the placement of these blocks so that the output it directed into a data file.



Options
ID: top_block
Generate Options: WX GUI

Variable
ID: samp_rate
Value: 1M

Variable
ID: freq
Value: 905.999M

Variable
ID: freq_offset
Value: 120k

Variable
ID: channel_spacing
Value: 200k

Variable
ID: channel_trans
Value: 80k

Variable
ID: fsk_deviation_hz
Value: 32k

osmocom Source
Sample Rate (sps): 1M
Ch0: Frequency (Hz): 906.119M
Ch0: Freq. Corr. (ppm): 0
Ch0: DC Offset Mode: Off
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Manual
Ch0: RF Gain (dB): 10
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20

Variable
ID: data_rate
Value: 76.7212k

Variable
ID: samp_per_sym
Value: 13

Frequency Xlating FIR Filter
Decimation: 1
Taps: firdes.low_pass(1, s...
Center Frequency: -120k
Sample Rate: 1M

WX GUI FFT Sink
Title: FFT Plot
Sample Rate: 1M
Baseband Freq: 906.119M
Y per Div: 10 dB
Y Divs: 10
Ref Level (dB): 0
Ref Scale (p2p): 2
FFT Size: 1.024k
Refresh Rate: 15
Peak Hold: On
Average Alpha: 0
Freq Set Varname: None

Quadrature Demod
Gain: 39.7887

Low Pass Filter
Decimation: 1
Gain: 1
Sample Rate: 1M
Cutoff Freq: 80k
Transition Width: 50k
Window: Blackman
Beta: 6.76

File Sink
File: ...721_lfp_hackrf.cfile
Unbuffered: Off
Append file: Overwrite

Add Const
Constant: 6

Clock Recovery MM
Omega: 13
Gain Omega: 7.65625m
Mu: 500m
Gain Mu: 175m
Omega Relative Limit: 5m

Binary Slicer

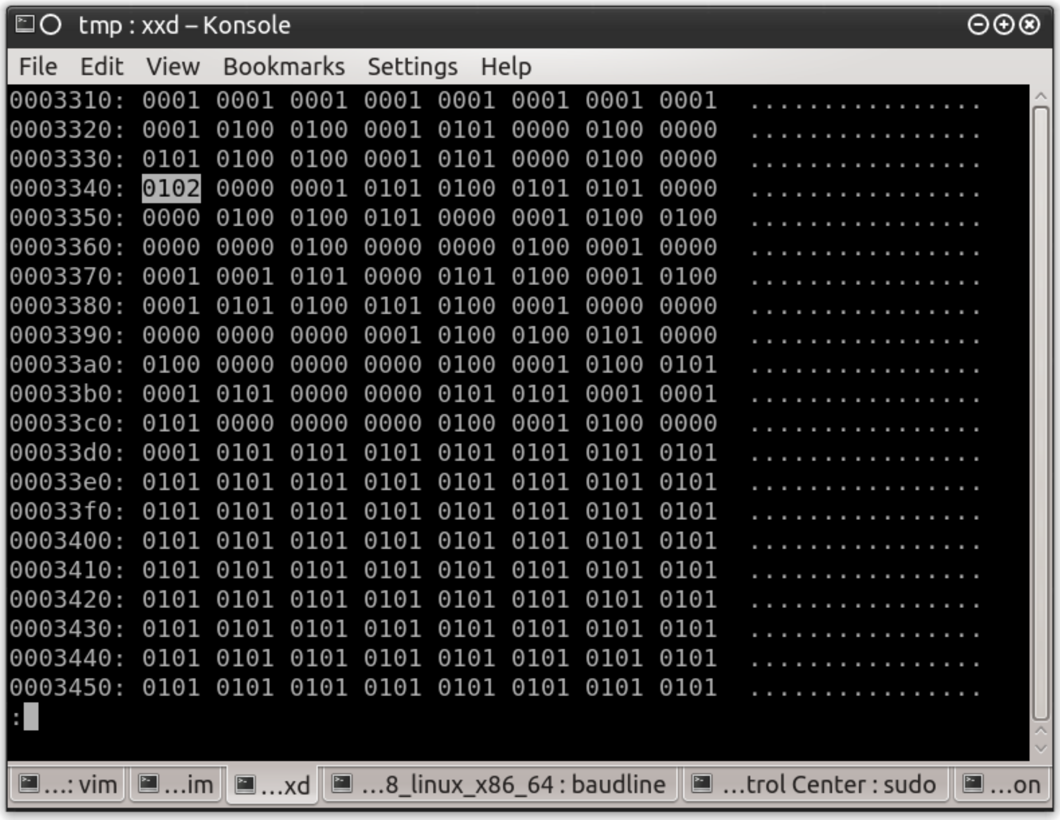
Correlate Access Code
Access Code: 11010...10010001
Threshold: 2

File Sink
File: ...lfp_hackrf_coded.dat
Unbuffered: Off
Append file: Overwrite

```
built-in source types: file fcd rtl rtl_tcp uhd hackrf bladerf rfspace
Using Jawbreaker with firmware git-2e6a476

>>> Done
```

Figure 26 GRC Configured with the Correlate Access Code Block

After capturing the transmissions with this GRC configuration the data can be analyzed in the same manner as the previous output data file. Starting with a quick review using “xxd” a search for the value “02” or “03” will show any marked packets. Figure 27 shows one of these packets.

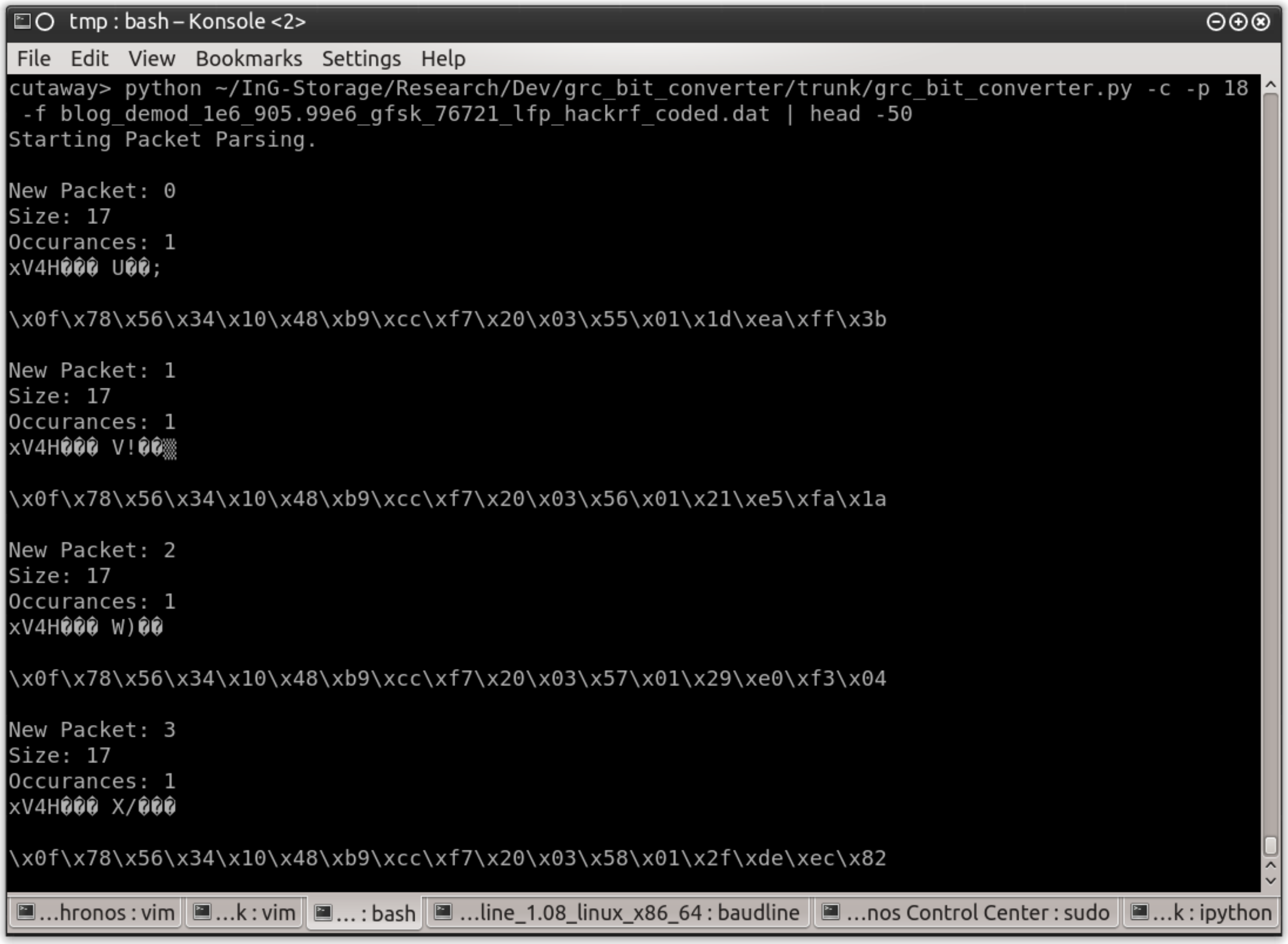


tmp : xxd – Konsole
File Edit View Bookmarks Settings Help
...: vim
...im
...xd
...8_linux_x86_64 : baudline
...trol Center : sudo
...on

```
0003310: 0001 0001 0001 0001 0001 0001 0001 0001  ................
0003320: 0001 0100 0100 0001 0101 0000 0100 0000  ................
0003330: 0101 0100 0100 0001 0101 0000 0100 0000  ................
0003340: 0102 0000 0001 0101 0100 0101 0101 0000  ................
0003350: 0000 0100 0100 0101 0000 0001 0100 0100  ................
0003360: 0000 0000 0100 0000 0000 0100 0001 0000  ................
0003370: 0001 0001 0101 0000 0101 0100 0001 0100  ................
0003380: 0001 0101 0100 0101 0100 0001 0000 0000  ................
0003390: 0000 0000 0000 0001 0100 0100 0101 0000  ................
00033a0: 0100 0000 0000 0000 0100 0001 0100 0101  ................
00033b0: 0001 0101 0000 0000 0101 0101 0001 0001  ................
00033c0: 0101 0000 0000 0000 0100 0001 0100 0000  ................
00033d0: 0001 0101 0101 0101 0101 0101 0101 0101  ................
00033e0: 0101 0101 0101 0101 0101 0101 0101 0101  ................
00033f0: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0003400: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0003410: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0003420: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0003430: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0003440: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0003450: 0101 0101 0101 0101 0101 0101 0101 0101  ................
:
```

Figure 27 Marked Packet Viewed Using "xxd"

As before the "grc_bit_converter.py" can be used to analyze the output data file. This script has an option to detect the packet markers provided by the "Correlate Access Code" block. The script also has the ability to modify the size of the packet printed to the screen. Figure 28 shows the first packet detected using these markers and printing a packet size of 18 which includes the length byte and the CRC. Reviewing this output quickly shows that this data packet matches the criteria in the previous analysis of this data.


tmp : bash – Konsole <2>
File Edit View Bookmarks Settings Help
...hronos : vim
...k : vim
... : bash
...line_1.08_linux_x86_64 : baudline
...nos Control Center : sudo
...k : ipython

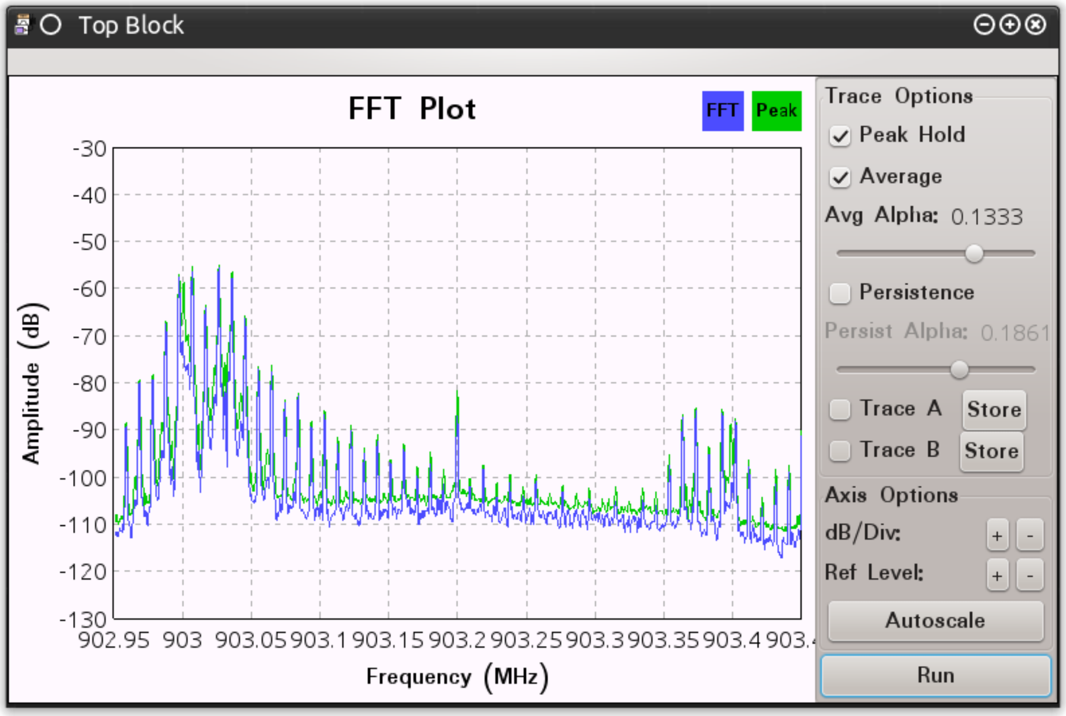
```
cutaway> python ~/InG-Storage/Research/Dev/grc_bit_converter/trunk/grc_bit_converter.py -c -p 18
 -f blog_demod_1e6_905.99e6_gfsk_76721_lfp_hackrf_coded.dat | head -50
Starting Packet Parsing.

New Packet: 0
Size: 17
Occurances: 1
xV4H��� U��;

\x0f\x78\x56\x34\x10\x48\xb9\xcc\xf7\x20\x03\x55\x01\x1d\xea\xff\x3b

New Packet: 1
Size: 17
Occurances: 1
xV4H��� V!��▒

\x0f\x78\x56\x34\x10\x48\xb9\xcc\xf7\x20\x03\x56\x01\x21\xe5\xfa\x1a

New Packet: 2
Size: 17
Occurances: 1
xV4H��� W)��

\x0f\x78\x56\x34\x10\x48\xb9\xcc\xf7\x20\x03\x57\x01\x29\xe0\xf3\x04

New Packet: 3
Size: 17
Occurances: 1
xV4H��� X/���

\x0f\x78\x56\x34\x10\x48\xb9\xcc\xf7\x20\x03\x58\x01\x2f\xde\xec\x82
```

Figure 28 Marked Data Output by "grc_bit_conveter.py"

9.0 TESTING THE CONTROL

The GRC configuration in this example works well for the TI Chronos Watch. But will the techniques leveraged be useful for other, similar, radio transmissions? To test, the TI Chronos Watch will be considered the control case. A good test subject is the radio transmission provided by Jay Radcliffe in his question to the GNU-Radio discussion list.[11] For this test Jay provided me a slightly different capture using the DC Offset method.

To analyze this transmission capture file the Source Block was replaced by a "File Source" and "Throttle" block. These blocks read the data from a file and throttle the data as it is passed to the rest of the GRC blocks. Throttling is performed so that the following blocks see the data at the captured rate, rather than just a straight data dump that would over load the program. In this case, the capture rate is 500,000 samples per second and the DC Offset is 200,000 Hz. The signal is being transmitted on the frequency 903 MHz using GFSK modulation. Figure 29 shows this capture from which the analysis will begin by trying to determine the Channel Width.



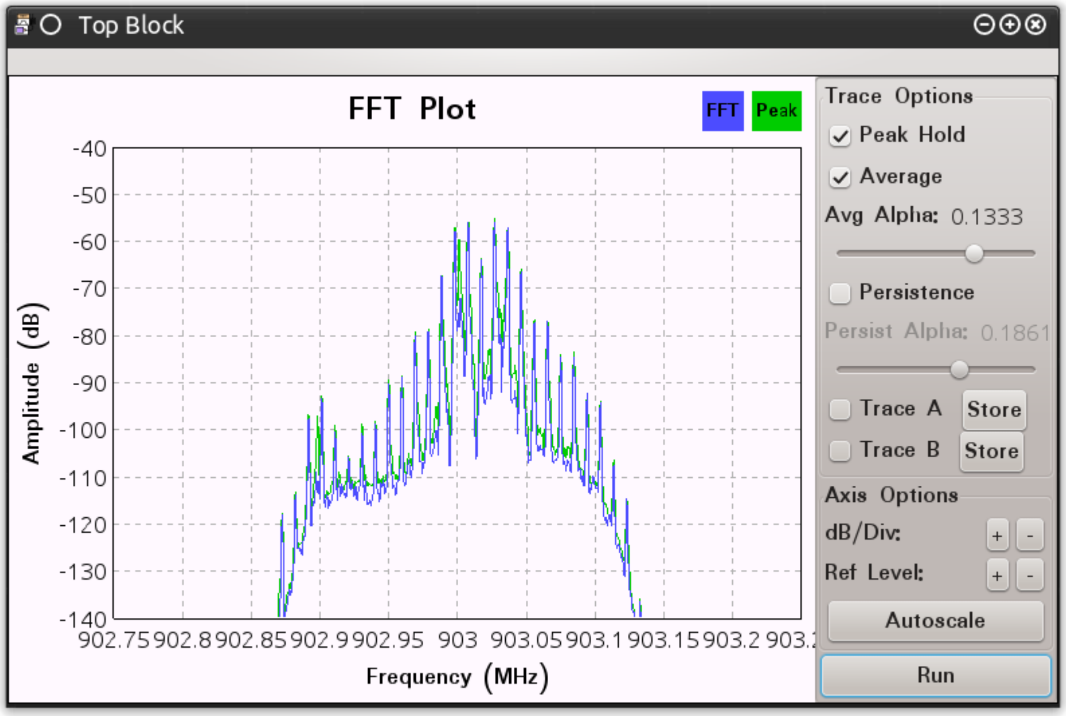
Trace Options
✓ Peak Hold
✓ Average
Avg Alpha: 0.1333
Persistence
Persist Alpha: 0.1861
Trace A
Trace B
Axis Options

Figure 29 Transmission on 903 MHz Captured with .2 MHz Offset

The captured transmission is close to the left-edge of the maximum capturing bandwidth as defined by the sampling rate. Although close, it does look like Jay captured the full transmission width. Because the signal is right on the edge of the captured bandwidth, care must be taken when configuring the settings for "Channel Spacing" and "Channel Transition." This is necessary to ensure that these values do not fall outside of the captured bandwidth and negatively impact the rest of the demodulation. Using the Frequency scale it looks like the "Channel Spacing" goes from 902,950,000 to 903,100,000 giving a 150,000 Hz channel width. To stay inside this range the "Channel Spacing" will be set to 100,000 and the

[11] [Discuss-gnuradio] FSK Demodulation trouble : http://lists.gnu.org/archive/html/discuss-gnuradio/2014-04/msg00097.html

"Channel Transition" to 50,000. These values are used to update the FXFF parameters. The results of this update can be seen in Figure 30.



✓ Peak Hold
✓ Average
Avg Alpha: 0.1333
Persistence
Persist Alpha: 0.1861
Trace A
Trace B

Figure 30 Transmission Filtered Using FXFF Block

From Jay's GNU Radio posting much of the radio's configuration is known. The Deviation for this transmission is 16,500 and the data rate is 19,200 Symbols Per Second. The Deviation value is used to update the "fsk_deviation_hz" block. Once this is configured, the signal can be processed through the "Quadrature Demod" block and analysis of the demodulated signal using Baudline can begin. From this analysis the "CutOff Freq" and "Transition Width" values of 25,000 and 10,000, respectively, were chosen for the "Low Pass Filter" block. Figure 31 shows the resulting wave pattern as displayed in Baudline.

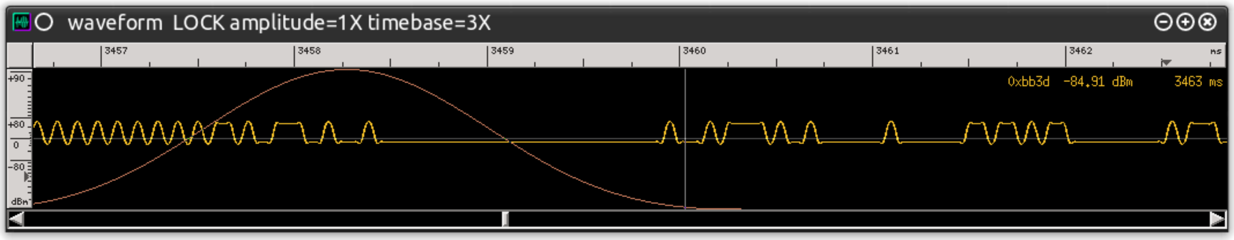
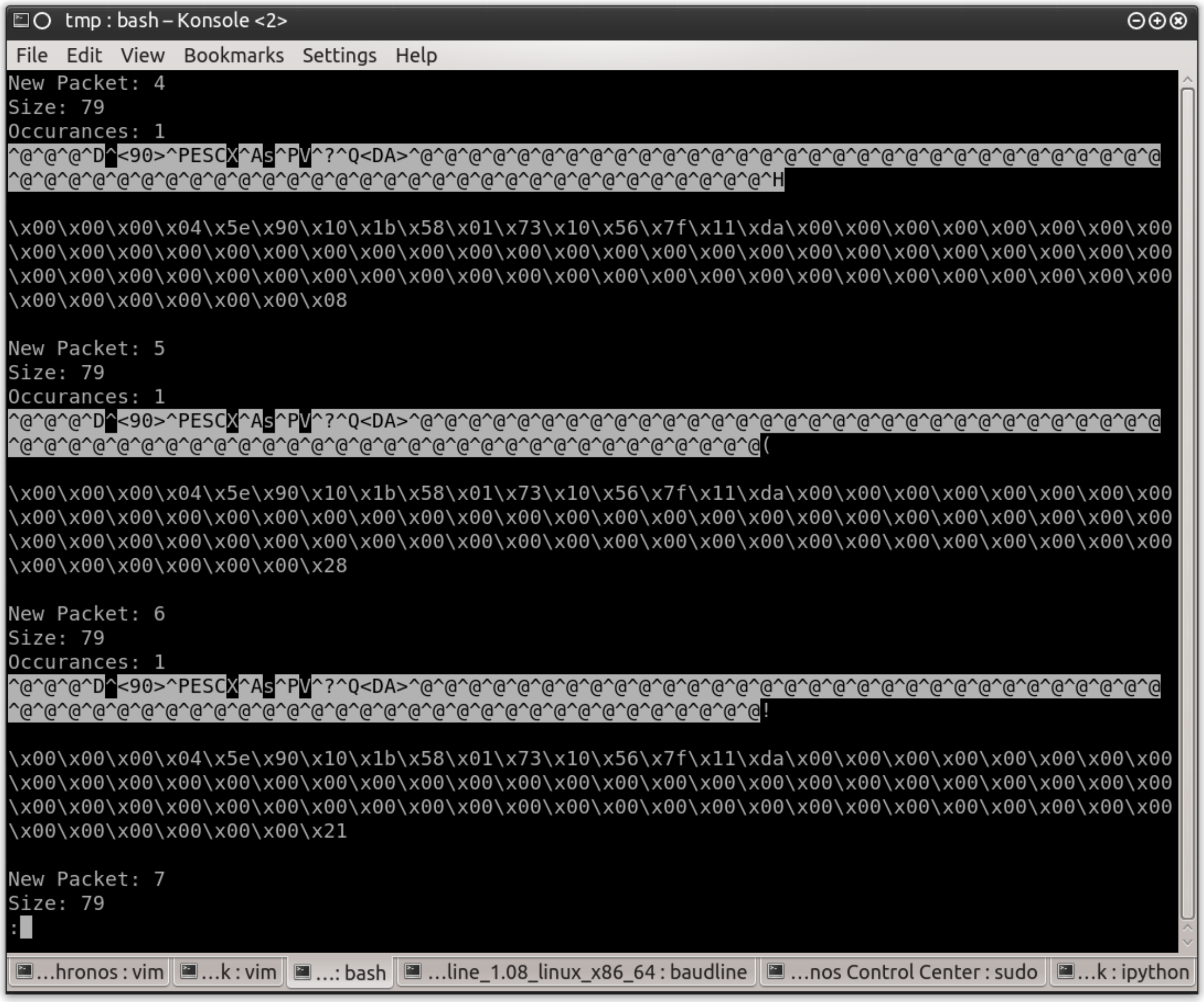


Figure 31 Demodulated Signal From Captured Transmission

From this image the wave form needs to be shifted down slightly. Through trial-and-error it was determined that configuring the "Add Constant" block with a constant value of negative six (-6) properly shifted the wave pattern down. Review of the data provided in Figure 31 determined that this radio only sends one SyncWord with the value of "0xd391". Using this information the "Correlate Access Code" block was also updated so that it marked a packet every time it detected this SyncWord. The data was output to a file and the "grc_bit_converter.py" script was used to check for marked packets. Initial analysis determined that the packet size for the transmitted data appears to be consistently output eighty (80) bytes per packet. Figure 32 shows some of the resulting packets output by the script.


```
New Packet: 4
Size: 79
Occurances: 1
^@^@^@^D^<90>^PESCX^As^PV^?^Q<DA>^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^H

\x00\x00\x00\x04\x5e\x90\x10\x1b\x58\x01\x73\x10\x56\x7f\x11\xda\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x08

New Packet: 5
Size: 79
Occurances: 1
^@^@^@^D^<90>^PESCX^As^PV^?^Q<DA>^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@(

\x00\x00\x00\x04\x5e\x90\x10\x1b\x58\x01\x73\x10\x56\x7f\x11\xda\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x28

New Packet: 6
Size: 79
Occurances: 1
^@^@^@^D^<90>^PESCX^As^PV^?^Q<DA>^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@!

\x00\x00\x00\x04\x5e\x90\x10\x1b\x58\x01\x73\x10\x56\x7f\x11\xda\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x21

New Packet: 7
Size: 79
:
```

tmp : bash – Konsole <2>
File Edit View Bookmarks Settings Help
...hronos : vim
...k : vim
...: bash
...line_1.08_linux_x86_64 : baudline
...nos Control Center : sudo
...k : ipython

Figure 32 Marked Packets Displayed using "grc_bit_converter.py"

The next steps in this process would be to determine the protocol used in these packets. With this data analysis could begin reviewing the data packets for length bytes, CRC bytes, and processing to determine if this data has been transformed using data whitening techniques (which it has not). Additional captures of the radios transmissions should be taken during different states: Power up, Power Down, Initial Pairing, etc. The more information collected the better understand how these transmissions are used to communicate, which endpoints actually transmitted them, and how GRC or other radios can be configured to interact with the end point devices.

10.0 CONCLUSION

Capturing transmissions with GRC is fairly easy to accomplish with the proper equipment and software. Understanding how to manipulate the transmissions captured using GRC is a bit more difficult. This makes demodulating the transmissions into data packets even more difficult. Fortunately, there is a growing amount of information about radio analysis, provided by people active in the GRC and radio analysis communities, and on the Internet. These communities make conducting signal analysis a bit easier for beginner and intermediate radio hackers and security professionals even if some of the information is initially difficult to comprehend. InGuardians hopes that the information provided in this document helps shed some light on this topic and the readers will also contribute back to the community.

For those individuals looking for training on radio analysis we highly recommend monitoring Mike Ossmann and his HackRF project. Mike usually gives one or two radio analysis classes a year and they contain a more in-depth understanding of radio transmissions. Much of the information provided in this paper is a result of experiences resulting from this training and the assessments it has allowed us to conduct.

Additionally, the techniques outline in this paper are being worked into the "Assessing and Exploiting Control Systems with SamuraiSTFU"[12] courseware. Check the SamuraiSTFU website for training dates. Students attending this class will get to do everything outlined in this paper using the TI Chronos Watch provided by the class' hardware kit.

[12] Assessing and Exploiting Control Systems with SamuraiSTFU: http://www.samuraistfu.org/training-syllabus