A GNU Radio Implementation of Automatic Meter Reading for the ERT Packet Standard

Implementation Instructions for GNU Radio Companion

By Kirsten Basinet

November 29, 2015

Table of Contents

1. Introduction

1.1. Description

This document provides instructions for setting up and executing automatic ERT utility meter reading using GNU Radio Companion (GRC). The project source code and other helpful resources can be found in the appendix (see section 4).

1.2. Software Requirements

In general, a copy of the project source code along with GNU Radio running on an appropriate operating system with the proper SDR drivers are the only software requirements to perform ERT reading. The implementation described in this document was designed for GNU Radio Companion 3.7.7.1 running on Ubuntu 14.04 LTS, which were the most recent versions of the required software available when this research project was started in July 2015. It is expected—but not guaranteed—that these instructions will be compatible with both GNU Radio and Ubuntu (or other Linux distributions) for the foreseeable future.

Although old versions of GNU Radio and Ubuntu are still available, it is strongly recommended that you download the latest version of each from the respective organizations' websites. Some changes may be required to maintain compatibility, and it is up to you to find out what these are.

Prior to attempting these instructions, it is expected that you have a working copy of GNU Radio Companion installed on their machine and understand its purpose. You will also need the appropriate drivers for your SDR (for the USRP used in this project, this is the UHD available from Ettus). It is also assumed that you are familiar with using the terminal to get around Linux. Additionally, basic proficiency in Python will be needed to modify the source code if necessary.

1.3. Hardware Requirements

This project was completed on an Intel NUC D34010 machine using a USRP B200 software-defined radio (SDR). An unbranded 900 MHz antenna was also used. These instructions have not been tested with other SDRs, but they will likely work for many other models as long as the program parameters (e.g. sampling rate and block size) are updated where appropriate for the chosen radio.

2. Adding the ERT_Decoder Block to GRC

Once GNU Radio Companion is installed on the target machine, we can add the ERT_Decoder block to the list of available modules. The easiest way to do this is using gr-modtool, which is GNC's native tool for adding "out-of-tree" modules.

To get started, open a new terminal window. Navigate to the directory where you wish to save the ERT decoder module. Next, type

```
gr_modtool newmod
```

and press enter. You will be prompted to name a new module. Type

```
ERT_Module
```

Press enter again and wait for the tool to create the new module.

Once it finishes, enter:

```
cd gr-ERT_Module
```

and then

```
gr_modtool add
```

which will add a new block to the module. When prompted for a block type, enter

```
sync
```

followed by

```
python
```

to set the programming language to Python. For the block identifier, type

```
ERT_Decoder
```

and for the argument list type

```
Multiple
```

followed by

```
n
```

to prevent the tool from automatically adding special test code.

If you are using Ubuntu, type

```
sudo ldconfig
```

and then enter your user password. Both Ubuntu and non-Ubuntu users should continue by entering the following command:

```
mkdir build && cd build && cmake ../ && make
```

At this point, the skeleton for the ERT module has been created and all we need to do is add in the right source code. To do this, first open the source code download location and copy the ERT_Decoder.py file. Navigate to the folder containing the module you just created and paste the file into the *python* folder.

Go back to the source code download location and copy ERT_Module_ERT_Decoder.xml. Copy it to the *grc* folder inside the *ERT_Module* directory.

Go back to the terminal, which should now be in the *build* directory of *ERT_Module*. Enter

```
cmake ../ && make
```

into the command line. Verify that there are no errors from this process.

Finally, you need to find the location on your computer where GNU Radio's modules are stored. The exact path will be different for every computer, but on the machine used for this project, the path is

```
usr/local/share/gnuradio/grc/blocks
```

Inside it, there are .xml files for many different GRC blocks. The last step is to move ERT_Module_ERT_Decoder.xml from the *ERT_Module/grc* folder to this directory.

Once you're done, go to the terminal and enter

```
gnuradio-companion
```

to launch GNC. On the right-hand side of the window, verify that ERT_Module is among the list and that it contains the ERT_Decoder block. If it does, you're ready to get started decoding ERT packets.

3. Creating and Running the Flowgraph

3.1. Using ERT Data from a File

With GRC open, click the *open an existing flowgraph* icon on the toolbar (or press CTRL+O). Navigate to the location of the source code package and double-click on the ERT_Flowgraph.grc file to open it. Figure 1 shows what it looks like when opened.

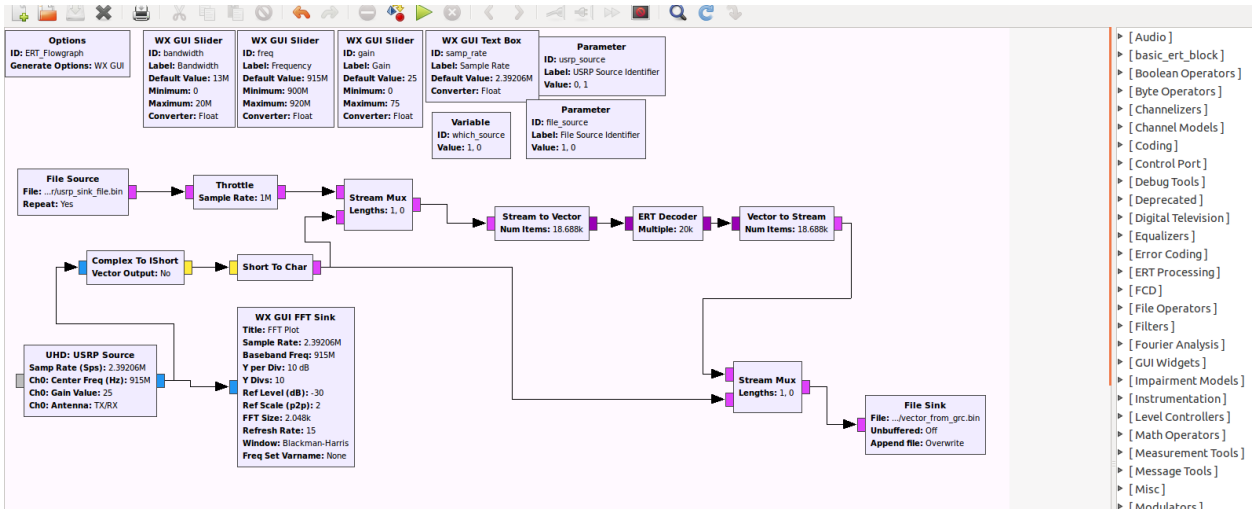

Figure 1 - Overview of Flowgraph

By default, the process will retrieve existing ERT data from a .bin fie instead of using live samples from the SDR. An example .bin file is included in the source code package for testing. You can change the location of the source file by double clicking on the *file source* block, which opens the window shown in figure 2. Next to the *File* text box, click the […] button and select the new file location. (If you would like to use the example data, navigate to the source code directory and choose the .bin file.) If you would like the same file to be processed continuously in a loop, you may change the value of *Repeat* to *Yes*. Click *OK* when finished.

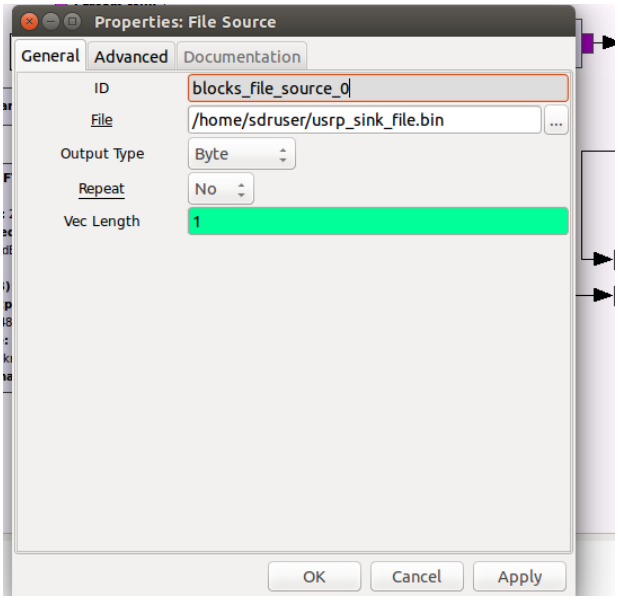

Figure 2 - Selecting File Source Location

To test the flowgraph, run it by clicking the green *execute the flowgraph* icon on the toolbar. Make sure you have an SDR connected to the computer before you do this.[1] GRC will take a few minutes to load drivers and initialize the radio firmware. Once it does, you may see a long message like the one shown in figure 3. You may safely ignore the "RF LO does not support the requested frequency" warning (it has to do with the precision of the frequency variable). The "tried to allocate…" warning may also be ignored.[2]

[1] If you do not have an SDR available, do the following: CTRL+click on the UHD: USRP Source, Complex to IShort, Short To Char, WX GUI FFT Sink, and both Stream Mux blocks. Press CTRL+D to disable them (the blocks will turn grey). Then, connect the output of the Throttle directly to the input of the Stream to Vector block by clicking once on the rightmost pink rectangle on the Throttle, and then on the leftmost pink rectangle on Stream to Vector. Finally, connect the output of the Vector to Stream block to the input of the File Sink block in the same way.

[2] This warning is a little bit different—it indicates that the data block size does not fit perfectly with the block of 4096 bytes that the machine prefers. This is harmless but it results in slightly more memory being used. Users who are extremely memory-conscientious can reduce the block size in the source code but this will result in slower processing.

```
linux; GNU C++ version 4.8.4; Boost_105400; UHD_003.009.git-204-g984575c9

Using Volk machine: avx_64_mmx
- Loading firmware image: /home/sdruser/target/share/uhd/images/usrp_b200_fw.hex... done
- Loading FPGA image: /home/sdruser/target/share/uhd/images/usrp_b200_fpga.bin... done
- Operating over USB 3.
- Detecting internal GPSDO.... No GPSDO found
- Initialize CODEC control...
- Initialize Radio control...
- Performing register loopback test... pass
- Performing CODEC loopback test... pass
- Asking for clock rate 32.000000 MHz...
- Actually got clock rate 32.000000 MHz.
- Performing timer loopback test... pass
- Setting master clock rate selection to 'automatic'.
- Asking for clock rate 38.273024 MHz...
- Actually got clock rate 38.273024 MHz.
- Performing timer loopback test... pass
- Tune Request: 915.000000 MHz
-  The RF LO does not support the requested frequency:
-    Requested LO Frequency: 915.000000 MHz
-    RF LO Result: 914.999999 MHz
-  Attempted to use the DSP to reach the requested frequency:
-    Desired DSP Frequency: -0.000001 MHz
-    DSP Result: -0.000001 MHz
-  Successfully tuned to 915.000000 MHz
-
gr::buffer::allocate_buffer: warning: tried to allocate
  4 items of size 18688. Due to alignment requirements
  16 were allocated.  If this isn't OK, consider padding
  your structure to a power-of-two bytes.
  On this platform, our allocation granularity is 4096 bytes.
gr::buffer::allocate_buffer: warning: tried to allocate
  4 items of size 18688. Due to alignment requirements
  16 were allocated.  If this isn't OK, consider padding
  your structure to a power-of-two bytes.
```

Figure 3 - Terminal Output at Runtime

After GRC has performed its initializations, the processing will start. Using the file, you'll immediately notice ERT data appear on the terminal output (both inside GRC and in the

dedicated terminal window) as shown in figure 4. The decoded meter data is displayed in real-time and includes the meter ID, meter type, tamper flag statuses, and consumption value.

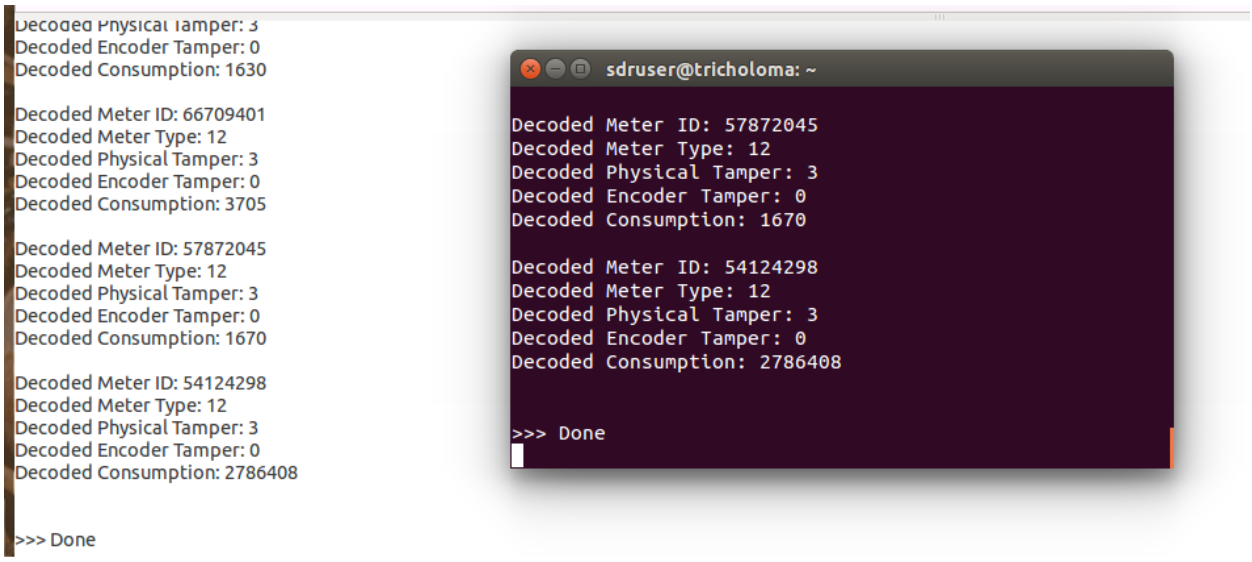

Figure 4 – Packet Detection

If you would like to stop the flowgraph from running, click the red *Kill the flowgraph* icon on the toolbar.

3.2. Using an SDR to Capture Data

Once you've verified that you're able to read ERT data from a file, you can use an SDR to capture real ERT meter data. Obviously, the frequency of packet detection will depend on your location relative to the meters and the type of antenna used.

To set up the flowgraph to use an SDR, double-click on the *which_source* variable. The dialog box that appears is shown in figure 5.

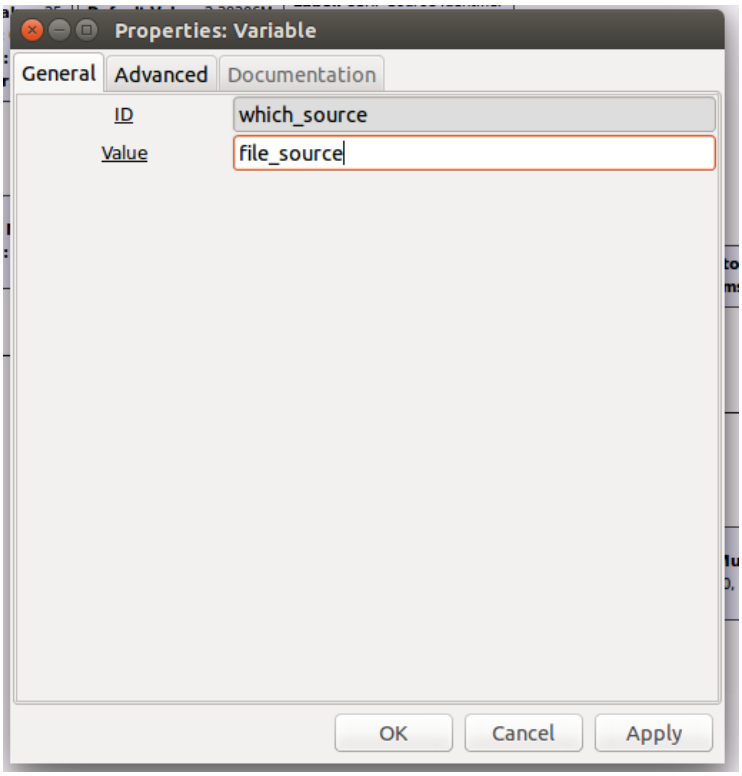

Figure 5 - Selecting the Data Source

By default, the *value* field is set to "file_source", which tells GRC to read the data from the .bin file. Delete this and type "usrp_source" to use the SDR (this will make the *Stream Mu*x blocks direct data to the right places). When you're finished, click *OK*.

All that's left to do is run the flowgraph by clicking on the green *execute the flowgraph* icon in the toolbar. A second GUI window like the one in figure 6 will appear. This window is intended for debugging purposes and lets you change the SDR's sampling rate, gain, center frequency, and bandwidth in real-time. It also displays a real-time FFT plot.

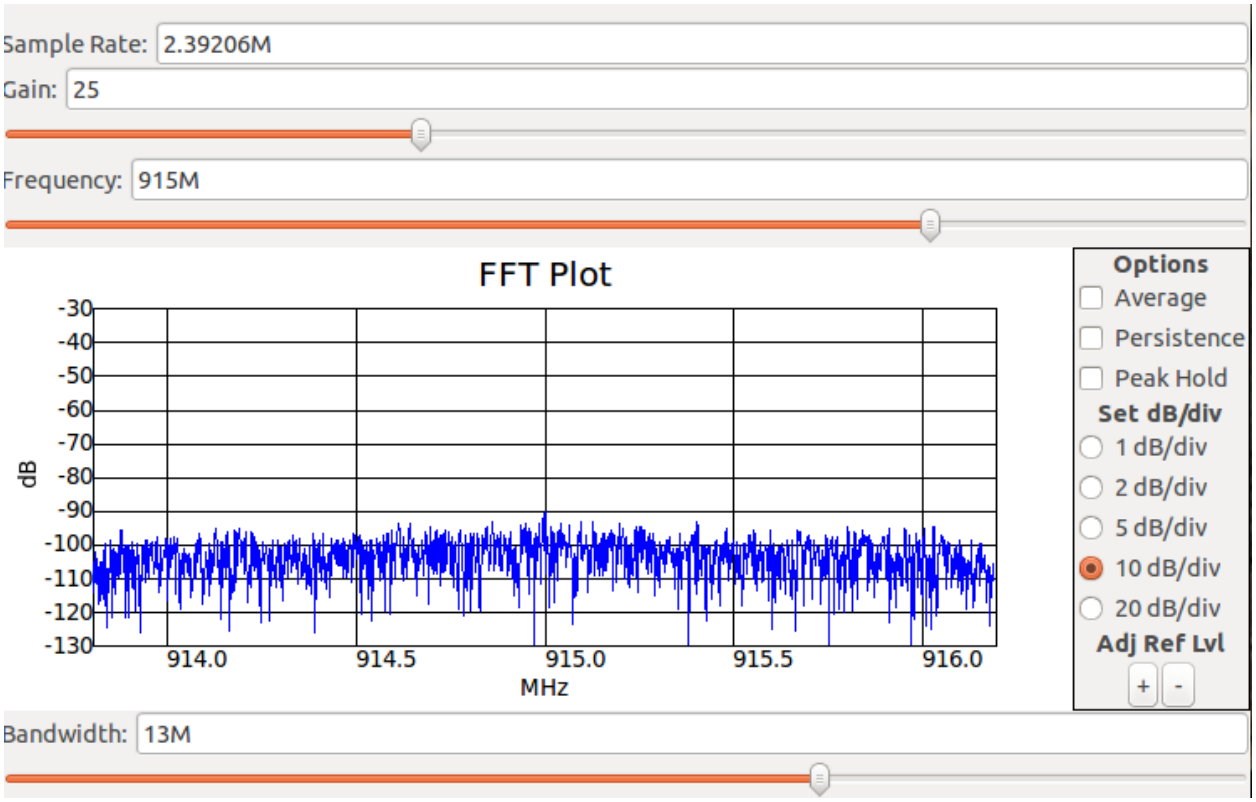

Figure 6 - Variable Selectors and FFT Display

You should start to see ERT data being received in real-time in the terminal window. This will probably happen much less often than it did when using the .bin file to decode data, so don't be alarmed if you're only receiving new data every few minutes even in an area with lots of ERT meters.

The last thing that the flowgraph does after displaying the decoded data is save the samples it gathered to a new .bin file. This is useful for debugging and signal analysis. To select the file save location, double click on the *file sink* block. The window shown in figure 7 will appear.

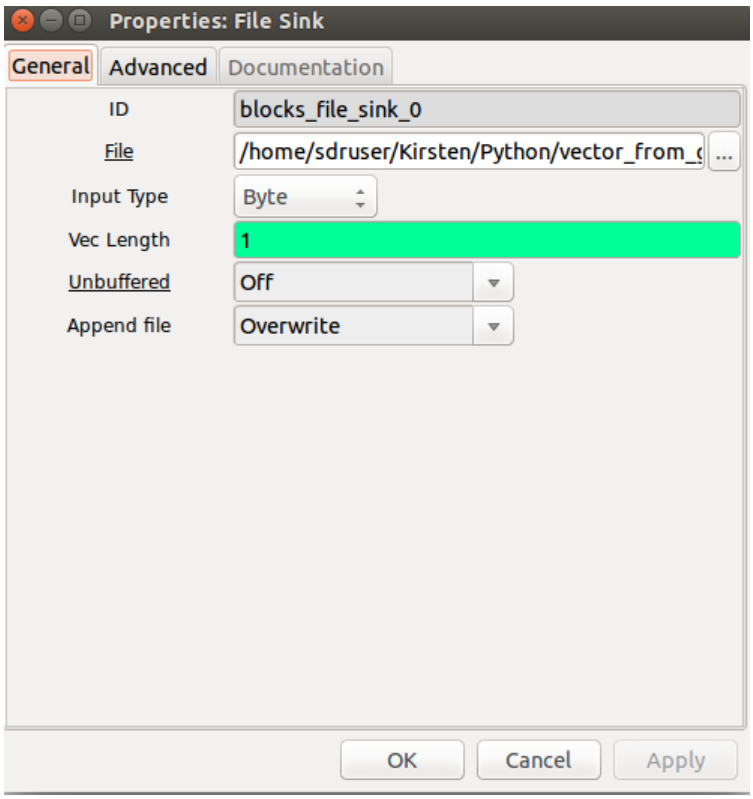

Figure 7 - Setting the File Sink Location

Click the […] icon next to the *File* field. Navigate to the location you'd like to save the file to and enter a name for it, then click *Save*. By default, the *Append file* option is set to *Overwrite*, which will overwrite the existing file data with a brand new file every time the flowgraph is run. If you want to change this behavior, you can change *Append file* to *Append*, which will add new data onto the end of the existing file.

4. Appendix

4.1 Additional Resources

Information about ERT:

http://www.gridinsight.com/community/documentation/itron-ert-technology/ (vendor)

https://en.wikipedia.org/wiki/Encoder_receiver_transmitter (wiki)

Information about USRP:

http://www.ettus.com/product/details/UB200-KIT (product page)

http://files.ettus.com/manual/ (UHD driver)

Information about GNU Radio Companion:

https://gnuradio.org/redmine/projects/gnuradio/wiki/TutorialsCoreConcepts (Basic info)

https://gnuradio.org/redmine/projects/gnuradio/wiki/Tutorials (Tutorials)

https://gnuradio.org/redmine/projects/gnuradio/wiki/MailingLists (support)

4.2 Project Source Code

ERTDecoder.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
## Decoder for ERT standard. v1.0: August 9, 2015
#  A.G. Klein and N. Conroy and K. Basinet
#  Description:  -Decodes ERT gas meter data from .bin file and displays
#                 meter ID, meter type, physical tamper flag, encoder tamper flag,
#                 consumption value, and the elapsed time since the last loop.
#                 Code is based on similary ERT decoder for MATLAB written by
#                 A.G. Klein and N. Conroy and K. Basinet
#  Dependencies: -Requires Numpy and custom function polynomialDivision
#  ----------------------------------------------------------------------
import numpy as np
from polynomialDivision import polynomialDivision

#"Macro" for converting binary number as digits in list to decimal integer
def bin2dec(bin_list):
          bin_list = [int(b) for b in bin_list]
          return int(''.join(str(c) for c in bin_list),2)

#Parameters and constants
JMP = 30                            # Number of samples to jump over each iteration
DataRate = 16384                    # Data rate for determining symbol period
SMPRT = 2392064                     # RTL-SDR Sample Rate
BLOCKSIZE = 18688                   # RTL-SDR Samples per frame
SP = np.int16(SMPRT/DataRate)       # Nominal symbol period (in # samples)
BCH_POLY = [1,0,1,1,0,1,1,1,1,0,1,1,0,0,0,1,1] # BCH generator polynomial coefficients from ERT
standard
PREAMBLE = [1,1,1,1,1,0,0,1,0,1,0,1,0,0,1,1,0,0,0,0,0]  #From ERT standard, includes sync bit

#Load file into list
with open('rtlamr_log2015-12-29.bin', 'rb') as fid:
          dat = np.fromfile(fid,np.int8)
fid.close()
dat = dat-127
s = dat[1:(len(dat)-1):2]+1j*dat[2:(len(dat)-1):2]

#Preallocate buffer space
zbuff = np.zeros(BLOCKSIZE)
softbits = np.zeros(96)
bits = np.zeros(96)
cnt = 0 #Decoded message counter
block_index = 0 #Data block counter
while block_index < len(s)-BLOCKSIZE+JMP:
          i=0 # Counter for sample feeding
          zbuff = s[block_index:block_index+(BLOCKSIZE-1)] #Grab block of samples from file,
                                                           #store them in buffer
          buff = np.int32(np.real(zbuff))**2+np.int32(np.imag(zbuff))**2 #Cheap absolute value of
                                                                         #buffer
          while i < BLOCKSIZE-(96*SP):
                    cu = np.cumsum(buff[i:(i+96*SP)])
                    softbits = (2*cu[(SP/2)+1:(95*SP)+(SP/2)+1:SP])-cu[1:(95*SP)+1:SP]-
cu[SP+1:(95*SP)+SP+1:SP];
                    for n in range(len(softbits)): #List with '1' where corresponding index in
                                                   #softbits is positive
```

```
                        if softbits[n] > 0:
                                bits[n] = 1
                        else:
                                bits[n] = 0
                #Check if preamble is correct and parse data
                if np.array_equal(bits[0:len(PREAMBLE)],PREAMBLE):
                        #BCH processing
                        dc = np.concatenate([np.zeros(180),bits[21:96]])
                        if polynomialDivision(BCH_POLY,bits[21:96])[0] == 0:
                            #BCH passed
                            i = i+(96*SP)-JMP
                            cnt = cnt+1 #Increment detected message counter
                            #Separate BCH decoded blocks
                            dc_id = np.concatenate([dc[180:182],dc[215:239]])
                            SCM_ID = np.concatenate([bits[21:23],bits[55:79]])
                            dc_phy_tmp = dc[183:185]
                            dc_ert_type = dc[185:189]
                            dc_enc_tmp = dc[189:191]
                            dc_consump = dc[191:215]
                            #Convert to decimal
                            dc_id = bin2dec(dc_id)
                            dc_phy_tmp = bin2dec(dc_phy_tmp)
                            dc_ert_type = bin2dec(dc_ert_type)
                            dc_enc_tmp = bin2dec(dc_enc_tmp)
                            dc_consump = bin2dec(dc_consump)
                            #Print decoded output
                            print("Decoded Meter ID: %u" %dc_id)
                            print("Decoded Meter Type: %u" %dc_ert_type)
                            print("Decoded Physical Tamper: %u" %dc_phy_tmp)
                            print("Decoded Encoder Tamper: %u" %dc_enc_tmp)
                            print("Decoded Consumption: %u \n" %dc_consump)
                    else:
                        #Do nothing
                i = i+JMP   #Increment sample feeding counter
        block_index = block_index+(JMP*96) #Increment block counter
print(cnt)
```

polynomialDivision.py

```
#!/usr/bin/env python
## Unsigned polynomial division function
#  August 10 2015 by Kirsten Basinet
#  Parameters:   -divisor: Row vector containing descending polynomial
#                 coefficients
#                -dividend: Row vector containing descending codeword
#                 coefficients
#  Returns:      -quotient: Row vector containing descending quotient
#                 coefficients, or NaN if an error occurred
#                -remainder: Row vector contianing remainder
#  Dependencies: -Requires Numpy
#                 user has the communications systems toolbox.
#  Notes:        -The function may need more debugging for cases where
#                 divisor>dividend, negative numbers are included, and
#                 other possible inputs. Works for CRC applications.
#------------------------------------------------------------------------
import numpy as np

def polynomialDivision(divisor, dividend):
    #Initialize variables
    quotient = []
    remainder = []
    temp_dividend = []
    dividing = True
    #Convert divisor and dividend to integers
    divisor = [int(x) for x in divisor]
    dividend = [int(x) for x in dividend]
```

```
    #Remove leading zeros, return NaN if dividend or divisor are invalid
    try:
        dividend = dividend[np.nonzero(dividend)[0][0]:len(dividend)]
        divisor = divisor[np.nonzero(divisor)[0][0]:len(divisor)]
    except:
        quotient = np.nan
        remainder = np.nan
        dividing = False

    #Return NaN if divisor is bigger than dividend
    if len(divisor)>len(dividend):
        quotient = np.nan
        remainder = np.nan
        dividing = False

    #Perform division
    place_count = len(divisor)
    temp_dividend = dividend[0:place_count]
    while dividing:
        if temp_dividend[0] == 1: #Use XOR method of polynomial division
            quotient.extend([1])
            for i in range(len(divisor)):
                temp_dividend[i] = temp_dividend[i]^divisor[i]
        elif temp_dividend[0] == 0:
            quotient.extend([0])
        else: #Non-binary number or NaN
            remainder = np.nan
            quotient = np.nan
            dividing = False #Done
        place_count = place_count+1
        if place_count > len(dividend):
            #Remove leading zeros and set remainder
            try:
                remainder = temp_dividend[np.nonzero(temp_dividend)[0][0]:len(temp_dividend)]
            except:
                remainder = 0
            dividing = False
        else:
            temp_dividend.pop(0)
            temp_dividend.extend([dividend[place_count-1]])
    return remainder, quotient
#Done
```

Note: need to add project documentation websites/github link, include top block code & XML file