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

Design Description

By Kirsten Basinet

November 29, 2015

1. Introduction

This document is a description of the automatic ERT meter reader implemented in GNU Radio. The project was started in winter 2015 by Nicholas Conroy as a MATLAB implementation using an RTL-SDR receiver and continued in summer 2015 by Kirsten Basinet to add support for file reading to the existing code, fix known issues, and create a GNU Radio alternative. Because much of the code is shared between the MATLAB and GNU Radio versions, it is expected that the reader has read Conroy's *Software Defined Radio: Foundational Study* paper (see reference [1]).

This report is not intended to give a comprehensive understanding of the processing algorithms used for the project, because they have already been addressed in Conroy's report. Instead, it is a brief description of the changes that were made and an overview of the GNU Radio implementation in particular.

2. Changes to MATLAB Code

Before work began with GNU Radio, it was necessary to fully understand and improve the existing MATLAB source code (see section 5.1 for the full code). The main changes that were made are:

- Added functionality for reading ERT data from a file
- Removed dependence on Communications Systems Toolbox when not using RTL-SDR
- Enabled program to detect packets that are split between data blocks

In MATLAB, reading data from a .bin file is accomplished by using the `fread` command. Once the data is imported and stored into a double-precision vector, the processing method is identical to the data received by an RTL-SDR.

Although the Communications Systems Toolbox is required to interface with the RTL-SDR, it isn't strictly necessary in order to process information from a file. Therefore, in order to make the code as accessible as possible, the decision was made to remove additional dependencies from the toolbox. The most significant of these was the need for a custom polynomial division function in order to handle the packet's cyclical redundancy check (CRC). The algorithm used for this was based on the XOR method of polynomial division, which is the method one would use when doing the procedure by hand.

Additionally, it was necessary to create a function to handle binary-to-decimal conversions. Although MATLAB's native `bin2dec` command is available, it is extremely inefficient for the requirements of this project because it only accepts binary numbers as character strings with a length less than 53. The custom alternative that was written is very short and accepts a binary number as a vector where each element is a digit. It checks each digit, and for every '1' the square of the number's place relative to the decimal point is added to the total value to give the final decimal number.

Along with adding these features, it was important that any known errors be fixed before proceeding to GNU Radio. The most major bug was the inability for the program to decode packets that fell between data blocks, because there was no mechanism to "remember" values

from the previously consumed block. This was fixed by adding a small amount of redundancy to the processing loop so that a buffer of 2880 samples would be retained between data blocks. Through testing it was determined that this was the smallest practical buffer size to prevent packets from being missed. Occasionally, this method causes the same packet to be read twice. Although this is isn't ideal, the issue was ignored to meet time constraints because it does not take away from the functionality of the program.

3. GNU Radio Implementation

The final (and major) portion of this project was to "translate" the code to a system that could be used by GNU Radio Companion (GRC) using a USRP SDR. Because learning GRC requires a significant learning curve, as much of the original code as possible was used.

To accomplish this, the MATLAB source was first translated to Python as an intermediate step. (See section 5.2 for Python source code.) Then, the necessary code was added in order to interface it with GRC. Mainly, doing this requires knowing the architecture of GNU Radio and defining the input and output vectors of the custom block accordingly.

GNU Radio works by using a "top block" written in Python, which defines all of the processing blocks and parameters used and how each is connected together. The blocks themselves are functions written either in Python or C++ that accomplish a single task and are grouped together in "modules".

There are several major advantages to using GNU Radio over MATLAB or other another tool. One is that it's a completely free, open-source API supported by many community members as well as Ettus Research, arguably the most major general-purpose SDR manufacturer. Because of this, GNU Radio has native support for a large number of software-defined radios [2]. In addition, the graphical style of GRC makes it uniquely easy to see the path a signal takes as it is processed by each block.

For the purposes of this project, GRC is mainly used as a tool to interface the USRP with the ERT decoding code. All of the blocks used in between are used to do simple math operations or convert the data to a different type so that other blocks know how to use it.

4. Suggested Future Additions

If the reader is considering improving on this project or attempting something similar, a few suggestions for additional features include:

- Translate GNU Radio blocks in Python to C++ for increased efficiency
- Take advantage of more native GNU Radio modules to handle signal processing—it's possible that most, if not all, of the functionality of the custom block can already be done with pre-existing ones.
- Fix bug of packets being detected twice

5. Appendix

5.1 MATLAB Source Code

ERT_Decoder.m

```
%% Decoder for ERT standard v1.0
%  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.
%  Dependencies: -Requires custom functions binary2decimal and
%                 polynomialDivision
%  ------------------------------------------------------------------------
%% Revision history:
%  22-feb-2015 v0.1 -initial version
%  22-feb-2015 v0.2 -updated to run on Nick's sample code
%  03-mar-2015 mod2 -added ert_out for message decoding -NC
%  13-mar-2015 mod3 -first attempt at cumsum detector -NC
%  21-mar-2015 mod4 -second attempt at cumsum detector -NC
%  22-mar-2015 mod5 -succesful cumsum detector, integrated ert_out func. -NC
%  23-mar-2015 mod6 -detector optimization test -NC
%  23-mar-2015 mod7 -more detector optimization tests -NC
%  24-mar-2015 mod8 -conversion to fully vector operations
%                   -optimized checksum operation           -NC
%  24-mar-2015 mod9 -first attempt at real time decoding with RTL-SDR -NC
%  10-jul-2015 v1.0 -new version that reads ERT data from .bin file -KB
%  15-jul-2015 mod1 -some packets being dropped, some invalid packets detected
%                   -BCH processing disabled -KB
%  29-jul-2015 mod2 -new function successfully checks BCH -KB
%                   -fixed error in preamble check, all detected packets
%                    are valid -KB
%  03-aug-2015 mod3 -still attempting to fix boundary data block skipping,
%                    some packets still dropped -KB
%  04-aug-2015 mod4 -all packets get decoded successfully, but some multiple
%                    times -KB
%------------------------------------------------------------------------------
clear;
%% 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=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.
fname='rtlamr_log_2-20-2015.bin';  % Raw data file name

%% Load file
fid=fopen(fname);
dat=fread(fid,'uint8=>double');
dat=dat-127;
s=dat(1:2:end)+1j*dat(2:2:end);
fclose(fid);

%% Preallocate buffer space
zbuff = zeros(BLOCKSIZE,1);
softbits = zeros(96,1);
bits = zeros(96,1);
cnt = 0; %Decoded message counter
block_index = 1;
```

```
while block_index < numel(s)-BLOCKSIZE+JMP
    tic %Start timing of one loop
    i = 1; %Counter for sample feeding
    zbuff=s(block_index:block_index+(BLOCKSIZE-1)); %Grab 18688 samples from file, store them in
                                                    %buffer
    buff = int32((real(zbuff)).^2+((imag(zbuff)).^2)); %Cheap absolute value of buffer
    while i < BLOCKSIZE-(96*SP) %Loop feeds samples through decoder
       cu = cumsum(buff(i:i+96*SP)); %Perform cumulative summation
       softbits = (2*cu((SP/2)+1:SP:(95*SP)+(SP/2)+1))- cu(1:SP:(95*SP)+1) -
cu(SP+1:SP:(95*SP)+SP+1);
       bits = (softbits>0); %Column vector with '1' where corresponding index in softbits is
                            %positive

       %% Check if preamble is correct and parse data
       if sum(bits(1:21)==PREAMBLE) == 21
       bin_dec = binary2decimal(bits(22:96)'); %Convert binary bits to decimal
            %% BCH processing
            dc = [zeros(180,1);bits(22:96)];%bin2dec(num2str(bits(22:96)))];
            if polynomialDivision(BCH_POLY,bits(22:96)') == 0
                %%BCH passed
                i = i+(96*SP)-JMP; %Jump past current message on next iteration
                cnt = cnt+1;       %Record successful message detection
                %% Separate BCH Decoded blocks
                dc_id = [dc(181:182);dc(216:239)];
                SCM_ID = [bits(22:23)',bits(56:79)'];
                dc_phy_tmp = dc(184:185);
                dc_ert_type = dc(186:189);
                dc_enc_tmp = dc(190:191);
                dc_consump = dc(192:215);
                %% Convert to decimal
                dc_id = binary2decimal(dc_id);%bin2dec(num2str(dc_id)');
                dc_phy_tmp = binary2decimal(dc_phy_tmp);%bin2dec(num2str(dc_phy_tmp)');
                dc_ert_type = binary2decimal(dc_ert_type);%(bin2dec(num2str(dc_ert_type)');
                dc_enc_tmp = binary2decimal(dc_enc_tmp);%bin2dec(num2str(dc_enc_tmp)');
                dc_consump = binary2decimal(dc_consump);%bin2dec(num2str(dc_consump)');
                %% Print Decoded Output
                fprintf('\nDecoded Meter ID: %d', dc_id);
                fprintf('\nDecoded Meter Type: %d', dc_ert_type);
                fprintf('\nDecoded Physical Tamper: %d', dc_phy_tmp);
                fprintf('\nDecoded Encoder Tamper: %d', dc_enc_tmp);
                fprintf('\nDecoded Consumption: %d', dc_consump);
                fprintf('\n');
            else
                %BCH failed
            end %end: if polynomialDivision(BCH_POLY,bits(22:96)') == 0
           %end %if (nerrs == 0)
       else
           %Preamble not found
       end %end: if sum(bits(1:21)==PREAMBLE) == 21
       i = i+JMP;  %skip ahead
    end %end: i < BLOCKSIZE-(96*SP)
    block_index=block_index+(JMP*96); %feed new data through the loop
    toc %Display end time for one loop
end %end: while block_index < numel(s)-BLOCKSIZE+JMP
```

binary2decimal.m

```
%% Unsigned binary to decimal converter
%  July 23 2015 by Kirsten Basinet
%  Parameters:    -bin_vector: Row vector containing binary number, where
%                  bin_vector(1) is the MSB
%  Returns:      -dec_result: Decimal representation of binary number,
%                  or NaN if an errror occurred
%  Notes:        -This function was created as an alternative to the
%                  vanilla MATLAB function bin2dec, which only accepts char
%                  strings 52 bits or less
%-------------------------------------------------------------------------
function dec_result = binary2decimal(bin_vector)
```

```
    dec_result = 0;
    for count = 0:1:length(bin_vector)-1
        if bin_vector(length(bin_vector)-count) == 1 %Current binary digit is 1
            dec_result = dec_result+2^count;
        elseif bin_vector(length(bin_vector)-count) == 0 %Current binary digit is 0
                %Do nothing
        else
            dec_result = NaN; %Error if bin_vector is not binary
        end %end: if bin_vector(length(bin_vector)-count) == 1
    end %end: for count = 0:1:length(bin_vector)-1
end %end: function binary2decimal
```

polynomialDivision.m

```
%% Unsigned polynomial division function
%  July 27 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 the custom function binary2decimal. MATLAB
%                 native functions bin2dec and num2str can be used if
%                 dividing small polynomials. bi2de can be used if the
%                 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.
%-------------------------------------------------------------------------
function [remainder,quotient] = polynomialDivision(divisor,dividend)
    %Initialize variables
    clear place_count;
    clear remainder;
    quotient=[];

    %Remove leading zeros
    dividend = dividend(find(dividend,1,'first'):numel(dividend));
    divisor = divisor(find(divisor,1,'first'):numel(divisor));
    place_count = numel(divisor);
    temp_dividend = dividend(1:place_count);
    dividing = true;

    %Perform polynomial division
    while dividing
        if temp_dividend(1) == 1 %Use XOR method of polynomial division
            quotient = [quotient,1];
            temp_dividend = bitxor(temp_dividend,divisor);
        elseif temp_dividend(1) == 0
            quotient = [quotient,0];
        else
            %Non-binary number or NaN
            remainder = NaN; %Error
            quotient = NaN; %Error
            dividing = false; %Done
        end %end: if temp_dividend(1) == 1;
        place_count = place_count+1;
           if place_count > numel(dividend)
            %Remove leading zeros and set remainder
            remainder = temp_dividend(find(temp_dividend,1,'first'):numel(temp_dividend));
            if isempty(remainder)
                remainder = 0;
            else
                %Do nothing
            end %end: if isempty(remainder)
            dividing = false; %Done
        else
```

```
            temp_dividend = [temp_dividend(2:numel(temp_dividend)),dividend(place_count)];
        end %end: if place_count > numel(dividend)
    end %end: while dividing
end %end: function polynomialDivision
```

5.2 Python 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
```

6. References

[1] N. Conroy, "Software Defined Radio: Foundational Study," 9 April 2015. [Online]. Available: http://aspect.engr.wwu.edu/reports/conroySDRReport.pdf.

[2] "A Quick Guide to Hardware and GNU Radio," 4 October 2015. [Online]. Available: https://gnuradio.org/redmine/projects/gnuradio/wiki/Hardware. [Accessed 5 November 2015].