Reed-Solomon Encoder and Decoder

Aby Sebastian & Kareem Bonna

Under

Prof.Predrag Spasojevic
Electrical and Computer Engineering Department
Rutgers, The State University of New Jersey

1 Introduction

Burst Error (contiguous errors in the bit stream) is a common occurrence in digital communication systems, broadcasting systems and digital storage devices. Many mechanisms have devised to mitigate this problem. Forward error correction is a technique in which redundant information is added to the original message, so that some errors can be corrected at the receiver, using the added redundant information. Reed Solomon Encoder and Decoder falls in the category of forward error correction encoders and it is optimized for burst errors rather than bit errors. Reed Solomon Encoder and Decoder provide a compromise between efficiency and complexity, so that this can be easily implemented using hardware or FPGA.

Reed Solomon code is based on the Galois Field Arithmetic. Therefore this paper first discusses the Galois Field (GF) arithmetic first, and then goes into the mathematical theory behind Reed Solomon Encoder and Decoder. As part of this project we implemented Reed Solomon Encoder and Decoder on a Lab-view environment. Section 5 discusses the implementation of Reed Solomon Encoder and Decoder in Lab view. This paper ends with discussion of performance statistics generated from Lab view's simulated Digital communication channel as well as National Instruments USRP (Universal Software Radio Peripheral).

2 Galois Field Arithmetic

2.1 Galois field (GF)

Galois field consists of a elements that are generated from a primitive element, which is usually denoted by α. In this project we are using 2 as the primitive element. For a given primitive element α Field Elements takes values: 0, α^0, α^1, α^2, …., α^{N-1}, where $N = 2^m - 1$. One thing we need to understand is the fact that the multiplication we use here to generate α^x, is not the normal arithmetic, but the Galois Field arithmetic which we are going to discuss later in this section. α^x, representation of Galois field elements is very useful to explain GF multiplication and division.

We can also represent GF elements as polynomial expression of form:

$$a_{m-1}x^{m-1} + \dots + a_2x^2 + a_1x^1 + a_0$$

Where $a_{m-1} \dots a_0$ take the values 0 or 1. i.e polynomial representation of a GF element is nothing but the binary number $a_{m-1}a_{m-2}...a_1a_0$. This representation of GF element helps to describe the addition and subtraction operation among GF elements.

Addition and Subtraction operations among GF elements are same and it's defined as follows:

$$a_{m-1}a_{m-2}...a_1a_0 + b_{m-1}b_{m-2}...b_1b_0 = a_{m-1}a_{m-2}...a_1a_0 - b_{m-1}b_{m-2}...b_1b_0 = c_{m-1}c_{m-2}...c_1c_0$$

Where $c_n = a_n \text{ XOR } b_n$

Another important concept we are yet to define is the GF generator polynomial or primitive polynomial p(x), which is a polynomial of degree m which is irreducible, i.e. a polynomial with no factors. In this project we are using following polynomial as p(x):

$$x^8 + x^4 + x^3 + x^2 + 1$$

Galois field is generated on the concept that Primitive Element is a root of above equation, in GF arithmetic. There for we can write :

$$\alpha^8 = \alpha^4 + \alpha^3 + \alpha^2 + 1$$

This property will be used to derive all elements in the Galois field as described in the table below:

index form	polynomial form								decimal
	x^7	x^6	x^5	x^4	x^3	x^2	x^1	x^0	
0	0	0	0	0	0	0	0	0	0
α^0	0	0	0	0	0	0	0	1	1
α^1	0	0	0	0	0	0	1	0	2
α^2	0	0	0	0	0	1	0	0	4
α^3	0	0	0	0	1	0	0	0	8
α^4	0	0	0	1	0	0	0	0	16
α^5	0	0	1	0	0	0	0	0	32
α^6	0	1	0	0	0	0	0	0	64
α^7	1	0	0	0	0	0	0	0	128
α^8	0	0	0	1	1	1	0	1	29
α^9	0	0	1	1	1	0	1	0	58
α^{254}	1	0	0	0	1	1	1	0	142

Table 1 Galois field of 256 elements

Log of GF element is defined as follows:

$$\text{If } \alpha^x = y \text{ then } \log(y) = x$$

Log-inverse of a GF element can be defines as follows:

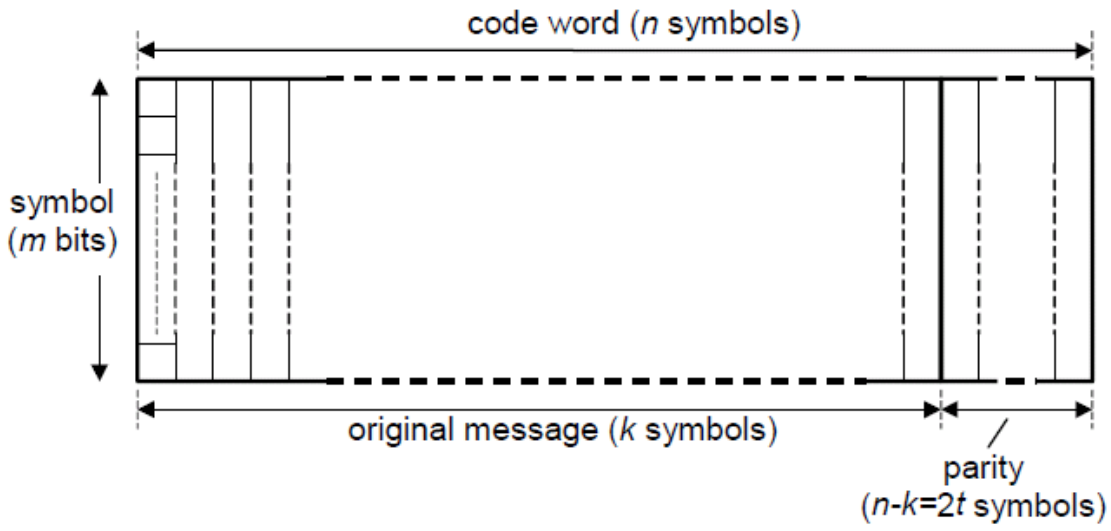
$$\text{If } \alpha^x = y \text{ then } \log^{-1}(x) = y$$

After definition of log and log-inverse it's easy to define multiplication and division among GF elements as follows:

$$0 \cdot a = 0$$
$$a.b = \log^{-1}\left((\log(a) + \log(b)) \bmod 255 \right)$$
$$a/0 \text{ not defined.}$$
$$a/b = \log^{-1}\left((\log(a) - \log(b)) \bmod 255 \right)$$

3 Reed Solomon Encoding

Reed Solomon Coding is a block coding scheme it takes a block of k symbols at a time and append 2t parity symbols. Following figure illustrates the scheme.



Encoder and decoder need to agree on a encoder polynomial g(x) which is defined as :

$$g(x) = \prod_{i=0}^{2t-1}(x + \alpha^i)$$

Encoder considers the k symbol block as a polynomial, M(x); of degree k-1, and first symbol being the coefficient of the most significant term. Encoder multiplies M(x) by x^{2t} and divides it with polynomial g(x) to get a remainder polynomial of maximum degree 2t-1. This polynomial will be added to M(x) .x^{2t} , to form a polynomial which is completely divisible by g(x).

Polynomial division in GF arithmetic is similar to normal arithmetic, only difference is that we will use GF addition/subtraction and GF multiplication in this scheme. Diagram below illustrates a GF polynomial division of a degree 10 polynomial by a degree 4 polynomial. To reduce complexity here we are using a GF(2^4) field, but in our project we are using GF(2^8).

	x^{14}	x^{13}	x^{12}	x^{11}	x^{10}	x^9	x^8	x^7	x^6	x^5	x^4	x^3	x^2	x^1	x^0
	1	2	3	4	5	6	7	8	9	10	11	0	0	0	0
$\times x^{10}$	1	15	3	1	12										
		13	0	5	9	6									
$\times 13x^9$		13	7	4	13	3									
			7	1	4	5	7								
$\times 7x^8$			7	11	9	7	2								
				10	13	2	5	8							
$\times 10x^7$				10	12	13	10	1							
					1	15	15	9	9						
$\times 1x^6$					1	15	3	1	12						
						0	12	8	5	10					
$\times 0x^5$						0	0	0	0	0					
							12	8	5	10	11				
$\times 12x^4$							12	8	7	12	15				
								0	2	6	4	0			
$\times 0x^3$								0	0	0	0	0			
									2	6	4	0	0		
$\times 2x^2$									2	13	6	2	11		
										11	2	2	11	0	
$\times 11x$										11	3	14	11	13	
											1	12	0	13	0
$\times 1$											1	15	3	1	12
												3	3	12	12

4 Reed Solomon Decoding

Reed-Solomon Decoder consider the incoming message as a polynomial R(x), transmitted message as T(x) and Error introduced as polynomial E(x). i.e.

$$R(X) = T(x) + E(x)$$

Now the Decoder problem is to identify the E(x) so that T(x) can be calculated as follows:

$$T(X) = R(x) + E(x)$$

4.1 Syndromes

We know that the T(x) is perfectly divisible by g(x) = $\prod_{i=0}^{2t-1}(x + \alpha^i)$. Therefore in case of no errors; for 'I' in [0, 2t-1], g(α^i) must evaluate to zero. When there are errors, some or all of them will result in a non-zero value and that value is called syndrome$_i$ (S_i) is defined as :

$S_i = R(\alpha^i)$
$= T(\alpha^i) + E(\alpha^i)$
$= E(\alpha^i)$
$= Y_1\alpha^{ie1} + Y_2\alpha^{ie2} + \ldots + Y_v\alpha^{iev}$
$= Y_1X_1^{\,i} + Y_2X_2^{\,i} + \ldots + Y_vX_v^{\,i}$

Where e1, e1 … ev are the locations of error and $Y_1, Y_2 \ldots Y_v$ are the corresponding coefficient/magnitude of the Error polynomial E(x).
And also $X_j = \alpha^{ej}$.

4.2 Error Locator polynomial

Assuming 'v' is the degree of the error polynomial; we can write Error Locator Polynomial as:

$$\Lambda(x) = (1+X_1x)\,(1+X_2x) \ldots (1+X_vx)$$

$$= 1 + \Lambda_1x^1 + \ldots + + \Lambda_{v-1}x^{v-1} ++ \Lambda_vx^v$$

Roots of the above equation Λ(x) are $X_1^{-1}, X_2^{-1}, \ldots X_v^{-1}$

Therefore we can write:

$$1 + \Lambda_1X_j^{-1} + \ldots + + \Lambda_{v-1}X_j^{-v+1} ++ \Lambda_vX_j^{-v} = 0$$

Multiplying both sides by $Y_jX_j^{\,i+v}$:

$$Y_jX_j^{i+v} + \Lambda_1\,Y_jX_j^{i+v-1} + \ldots + \Lambda_v\,Y_jX_j^{1} = 0$$

We can create similar equation errors for different values for X_j and terms collected together gives:

$$\sum_{j=1}^{v} Y_jX_j^{i+v} + \Lambda_1\sum_{j=1}^{v} Y_jX_j^{i+v-1} + \cdots + + \Lambda_v\sum_{j=1}^{v} Y_jX_j^{i} = 0$$

$$S_{i+v} + \Lambda_1\,S_{i+v-1} + \ldots + \Lambda_v\,S_i = 0$$

We can derive 2t-v simultaneous equation by substituting 'l' with different values in [0, 2t-v-1].

Easiest way to solve the above equation is using Berlekamp's algorithm.
Berlekamp's algorithm computes the coefficients of Λ(x) equation using following variables:

a correction polynomial C(x) which is initialized to 'x'.

syndrome values S_0, S_1, … S_{2t-1}.

L, current number of assumed errors; initialized to 0.

N, total number of syndromes, main loop of the algorithm will be executed N times.

B(x), copy of C(x) when L was last updated; initialized to 1.

d, discrepancy in C(x) calculated every iteration.

b, copy of d when L was last updated ; initialized to 1.

m, number of iteration since L was last updated.

Algorithm is as listed below:

- Each iteration , algorithm calculates the discrepancy d; on kth iteration 'd' is calculated using formula:
  $d= S_k +C_1S_{k-1} + …. + C_LS_{k-L}$
- If d is zero, algorithm assumes that C(x) and L are correct for the moment, increments m and continue.
- If d is not zero a new C(x) will be calculated as follows:
  $C(x) = C(x) – (d/b)x^m$. B(x) and L, b, B(x) and m will be updated.
- If the number of errors are less than or equal to N/2, then C(x) will contain Λ(x) on or before Nth iteration.

Once we have the coefficients of Λ(x) , finding $X_1, X_2 …X_v$ is very easy because $X_1^{-1}, X_2^{-1}, …. X_v^{-1}$, are the root of equation Λ(x) and also we know that $X_j = \alpha^{ej}$ and j is in [0,n-1], where n is length of code. So we use Chein search, method on Λ(x) to identify X_j eventually all ej. Chein search method is very simple, it evaluates given polynomial for all possible roots, in our case total n; if equation evaluates to zero then that value assigned to 'x' is considered as a root.

4.3 Error polynomial Coefficients

We use Forney's algorithm to calculate Y_j.

$Y_j = X_j \ [\Omega (X_j^{-1}) / \Lambda'(X_j^{-1})]$

Where $\Omega (x) = S(x) . \ \Lambda(x) \bmod x^{2t}$; i.e $S(x) . \ \Lambda(x)$ with all terms above degree 2t-1 ignored.

And $\Lambda'(x)$ is the first derivative of $\Lambda(x)$.
$\Lambda'(X_j^{-1}) = \Lambda_1 + \Lambda_3 X_j^{-2} + \Lambda_5 X_j^{-4} + \ldots$

5 Implementation

The Reed-Solomon Encoder and Decoder were implemented as part of an existing QPSK transmitter and receiver with key modules replaced by us in the lab. The notable modules that were replaced were the bit generator and BER calculator, the symbol mapper and demapper, the pulse shaping and matched filter, and some parts of the synchronization.

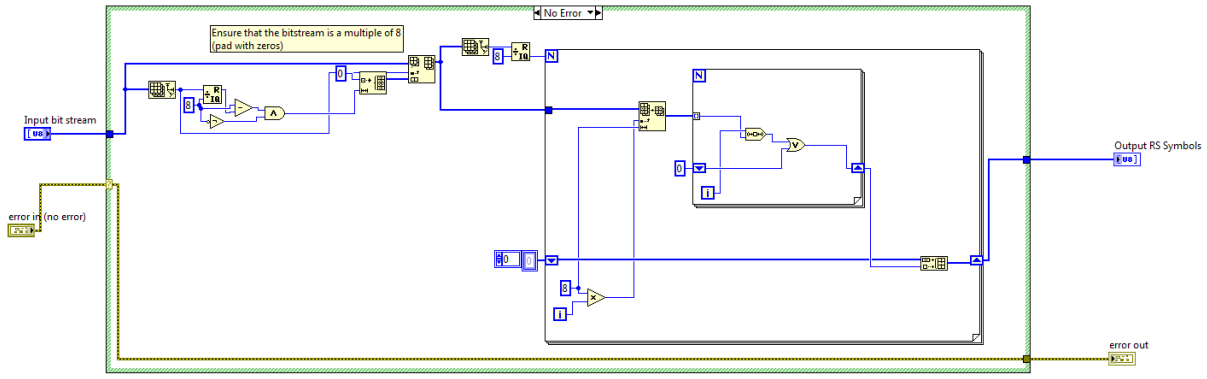
The Reed-Solomon code generator polynomial used was based off of the n=255, k=239 code. In most testing the code was shortened to n=32, k=16 via code shortening (populating the initial 239-16 symbols with zeros).

Software implementation of the Reed-Solomon Encoder and Decoder, and additionally parts of the baseband transmitter and receiver were done in LabVIEW. Testing was performed both in software using a simulator and in hardware using the NI USRP software defined radio.

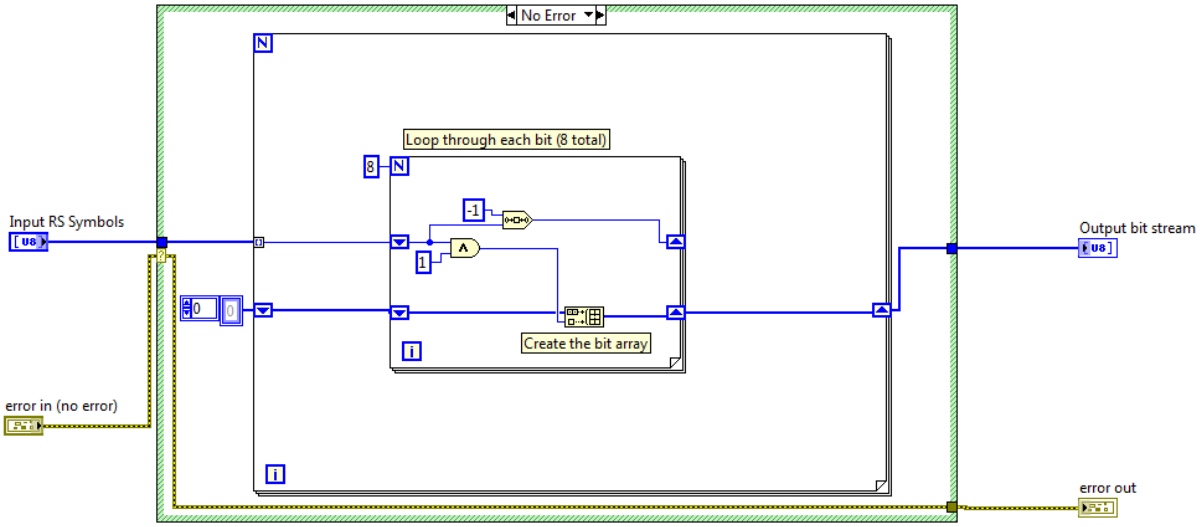
5.1 Generic Vis

Bits -> Symbols, Symbols -> Bits

rs_bits_to_symbols.vi



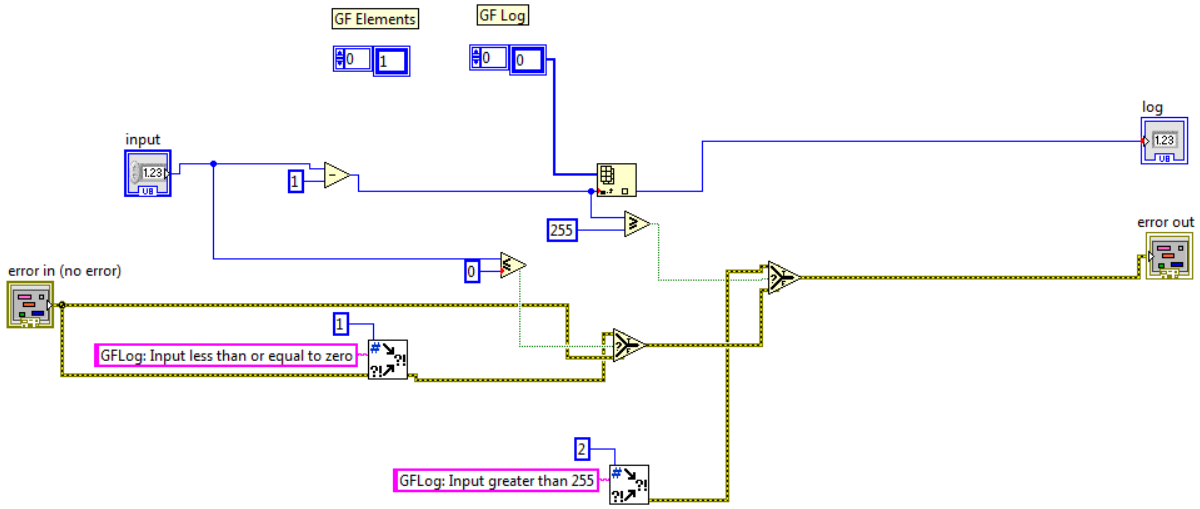
rs_symbols_to_bits.vi



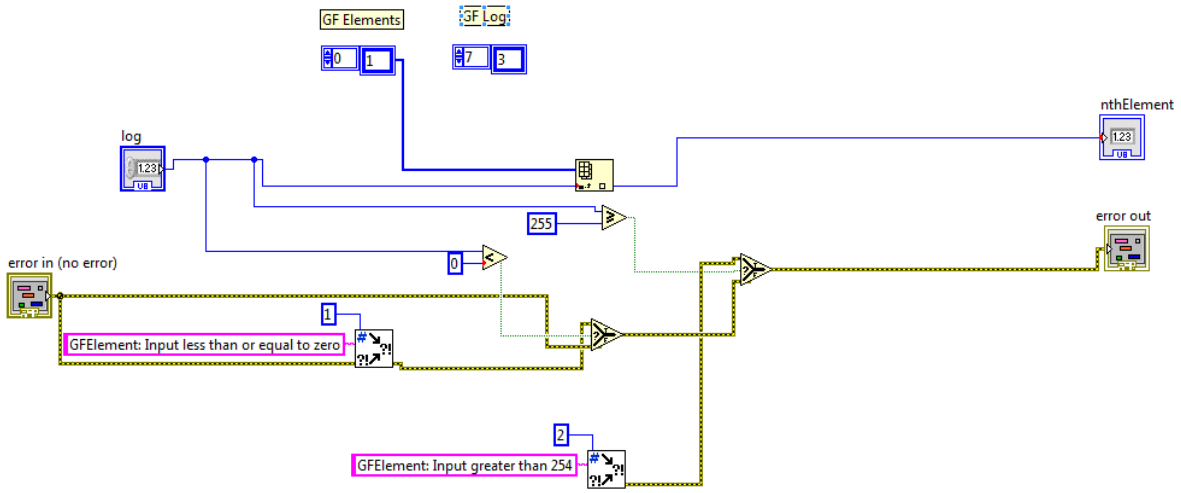
These VIs convert a stream of bits into a stream of symbols in GF(256) and vice-versa. This is done by grouping and ungrouping sets of 8-bits.

GF Log Lookup, Inverse Log Lookup

GFLog.vi



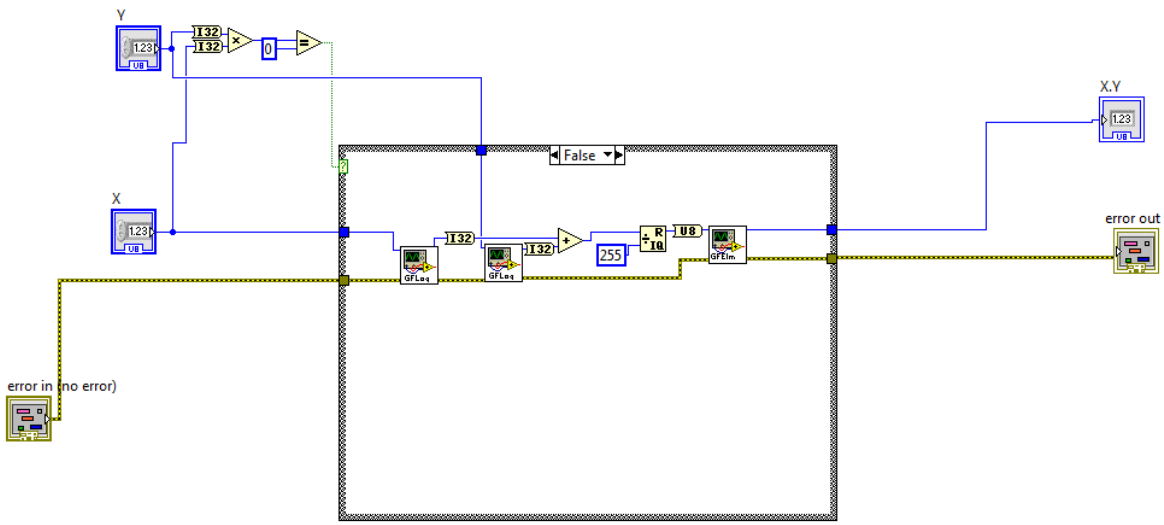
GFElement.vi



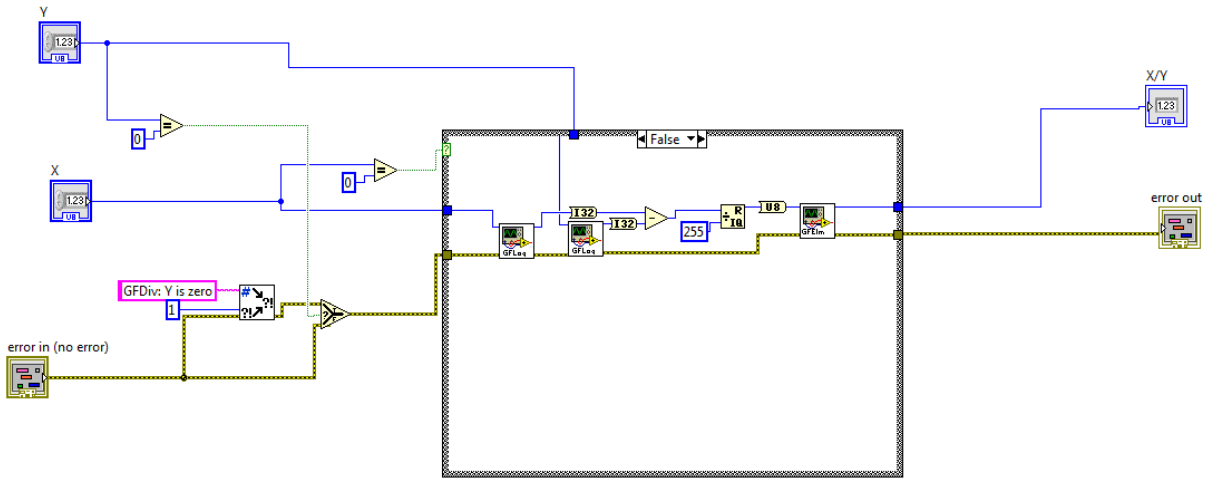
The log lookup VI takes an element in GF(256) and returns the power that the primitive element is raised to using the field generator polynomial. The inverse log lookup VI takes an integer from 0-255, and then returns the element in GF(256) that corresponds to the primitive element in the field raised to that power, using the field generator polynomial.

GF Multiply, Divide (Log Lookup)

GFMult_orig.vi



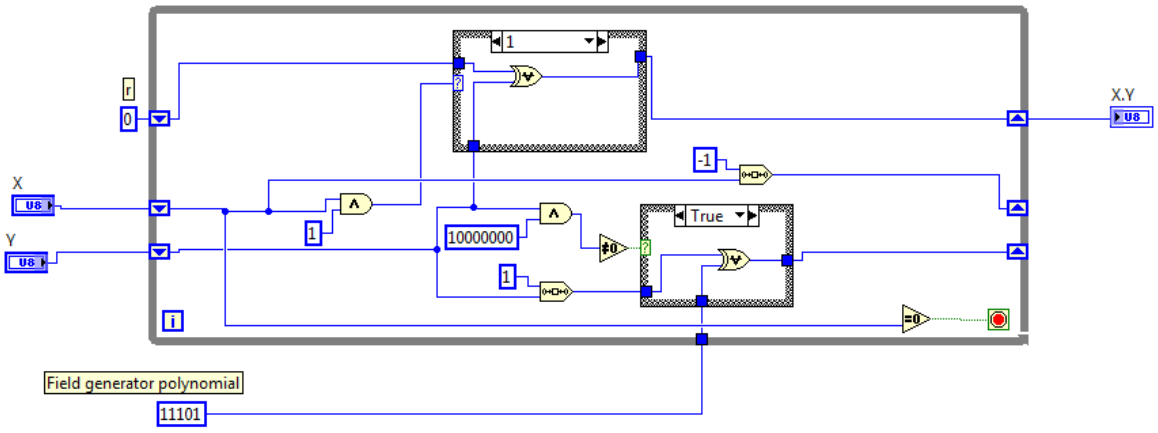
GFDiv.vi



The Log lookup GF multiply and divide VIs take two elements in GF(256), do a log lookup to get their exponents, add or subtract the exponents, and then does an inverse log lookup to get the result

GF Multiply (Iterative)

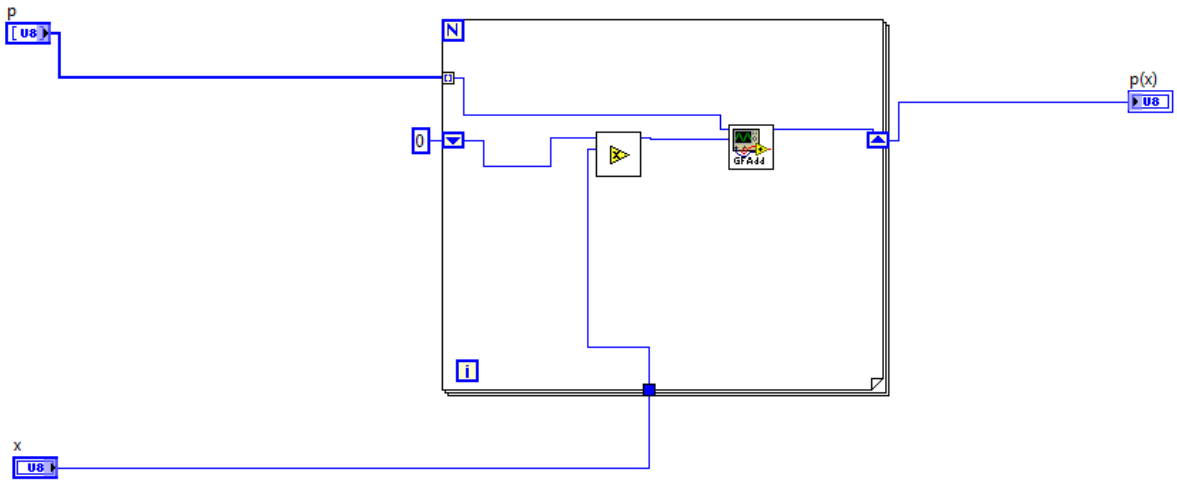
GFMult.vi



The iterative GF multiply VI takes two elements in GF(256) and multiplies directly using the generator polynomial, adding the polynomial back in when there is overflow. In a practical implementation this method would be slower, however in LabVIEW it runs faster than the log lookup method.

GF Polynomial Evaluate

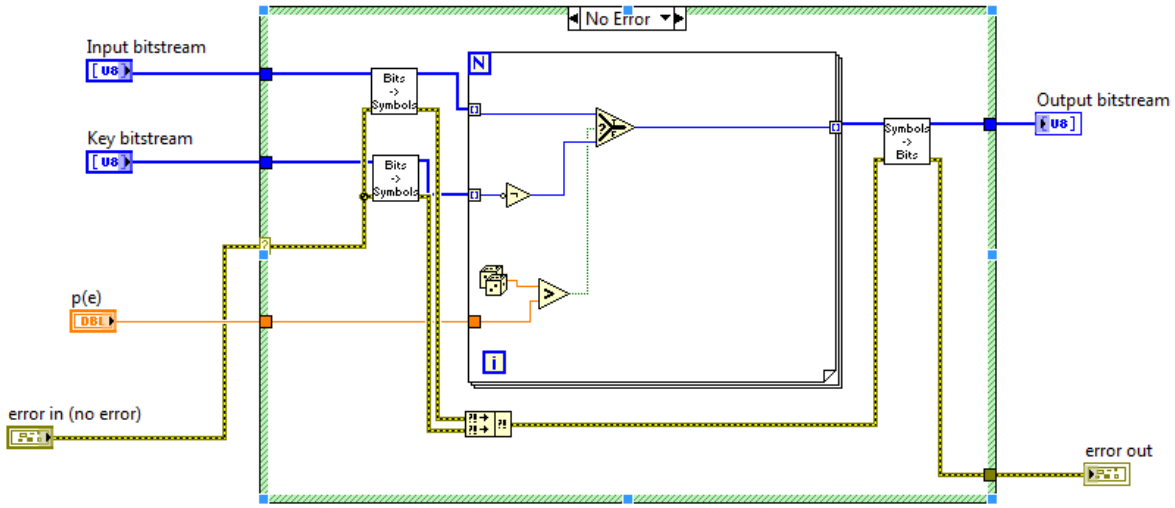
GFPEval.vi



This VI takes an array of coefficients of a GF(256) polynomial and evaluates it at some input value.

Erasure Channel

bec.vi

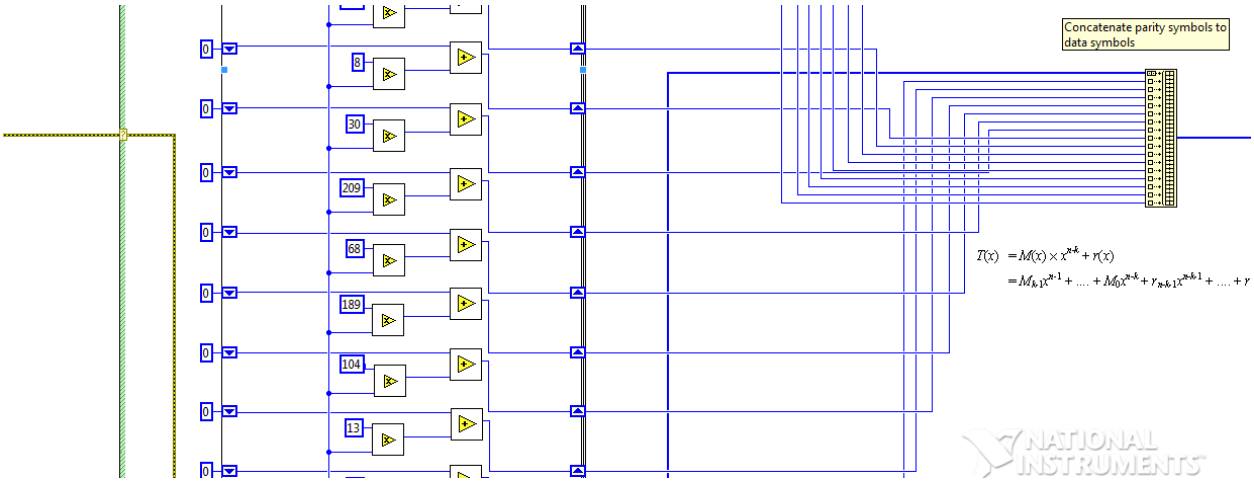


This VI emulates an erasure channel by replacing symbols with their inverted counterparts.

5.2 R-S Specific Vis

RS Encoder

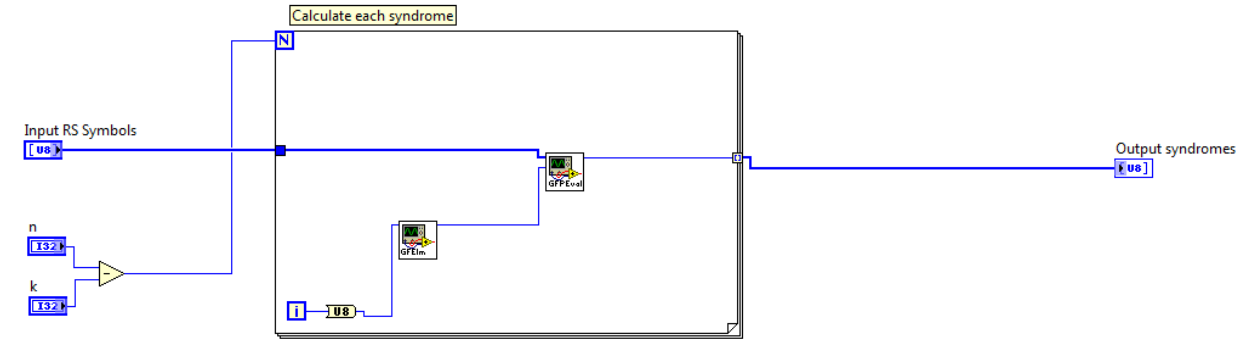
rs_encoder.vi



The RS Encoder is a LFSR that is used to calculate the remainder of polynomial division by the expanded code generator polynomial g(x). After shifting the message polynomial through, the registers hold the remainder which are the parity symbols that are appended to the message.

Syndrome Calculation

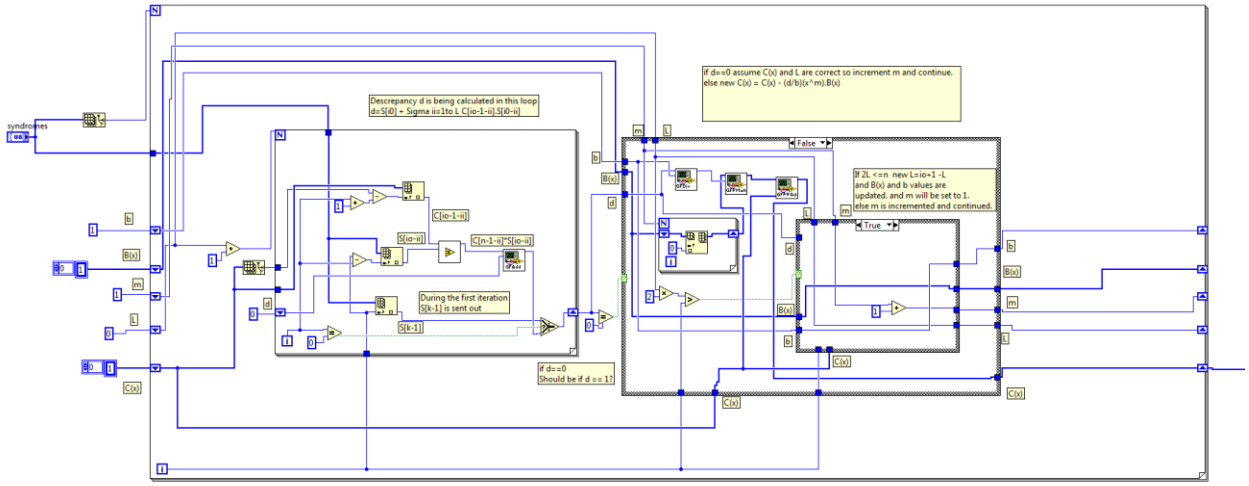
SyndromeGen.vi



The syndrome generator VI is used to evaluate the message polynomial at the roots of the code generator polynomial. All of the syndromes are zero when there is no error since the message polynomial is perfectly divisible by all roots.

Berlekamp's Algorithm

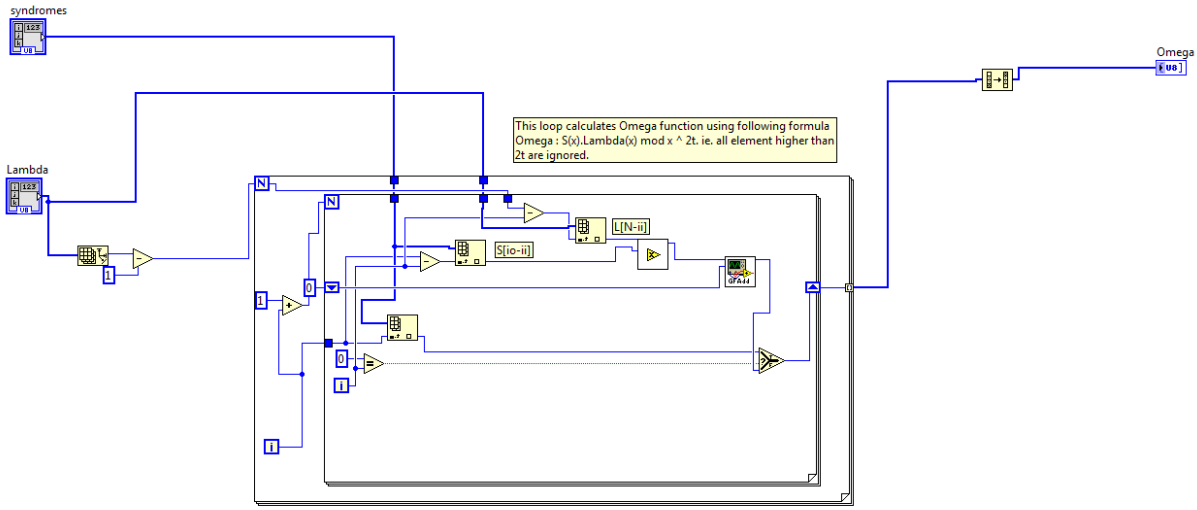
BerlekampsAlg.vi



Berlekamps Algorithm is used to compute the error locator polynomial, used to determine the error locations and error magnitudes.

Omega Calculation

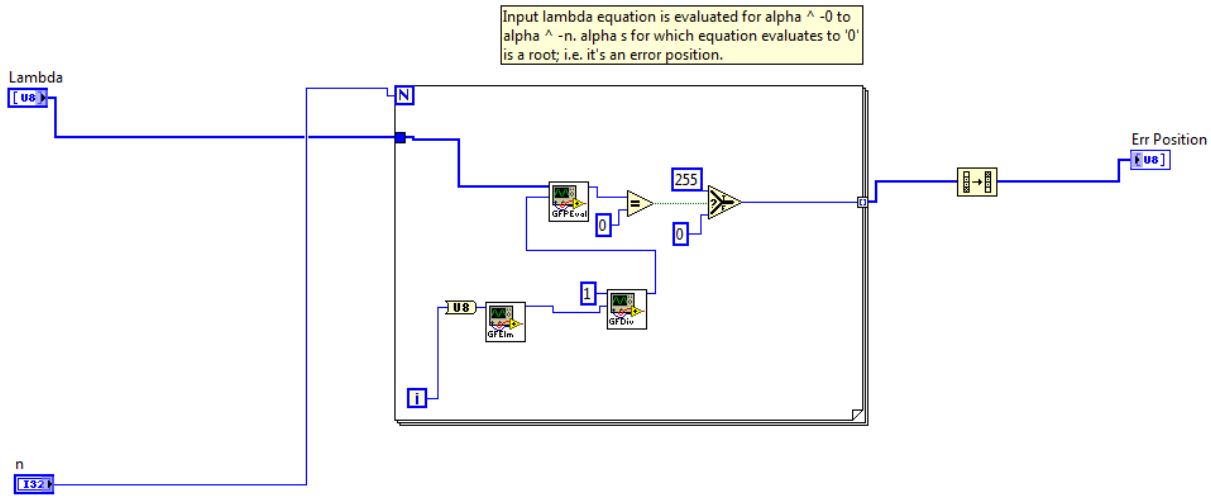
OmegaCalc.vi



The Omega Calculator VI calculates the Omega polynomial, used in Forney's algorithm to compute the error magnitudes.

Chien Search

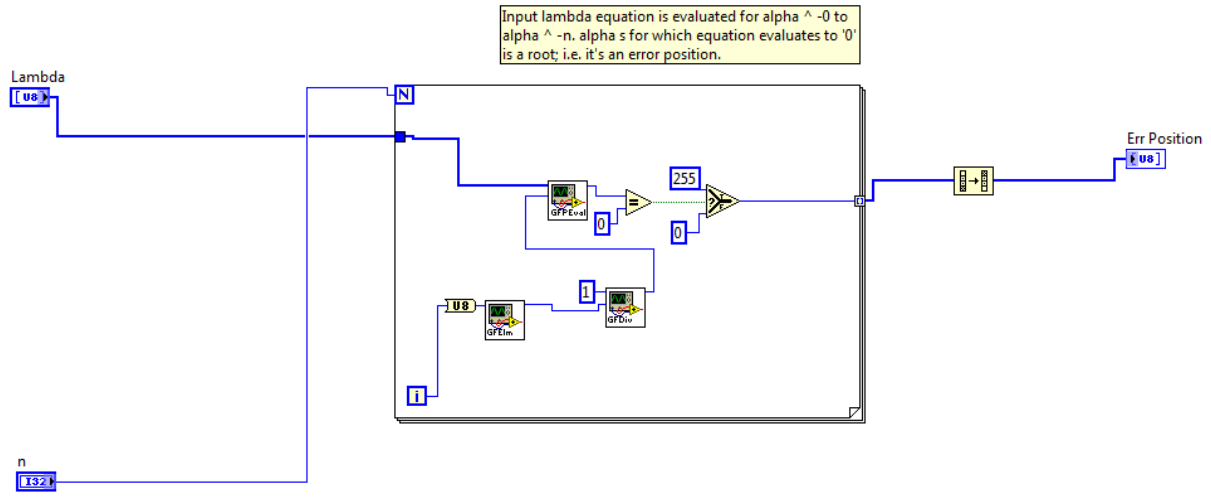
CheinSearch.vi



The Chein Search evaluates the error locator polynomial at the possible roots in GF(256) to determine the error locations in the message.

Forney's Algorithm

forney.vi

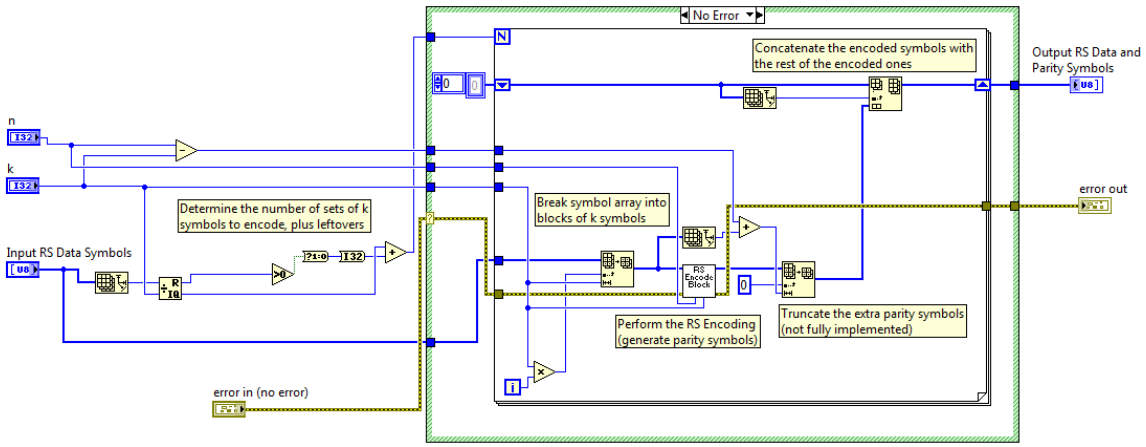


Forney's Algorithm computes the error magnitudes using the Omega and Lambda (error locator) polynomials.

5.3 Encoder Implementation

RS Block Encode

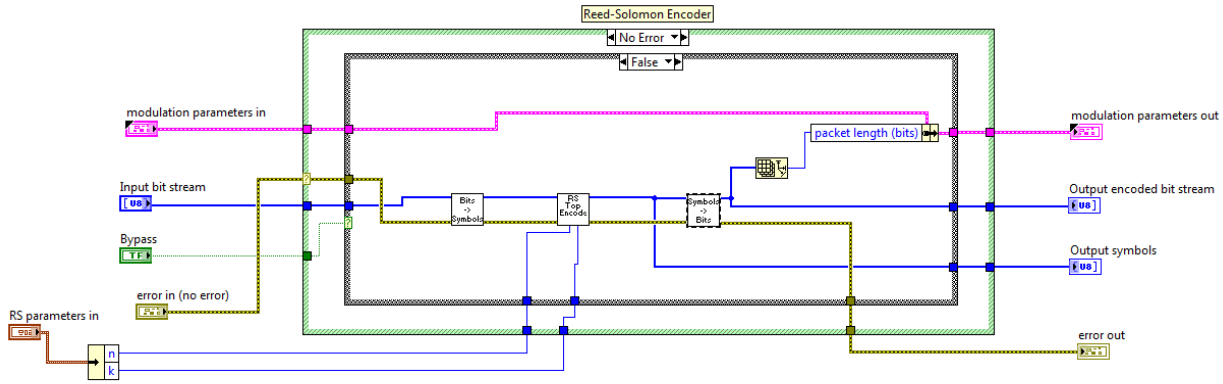
rs_top_encoder.vi



This vi takes an array of elements in GF(256) and splits them into blocks of k to be encoded.

RS Top-level Encoder

top_rs_enc.vi

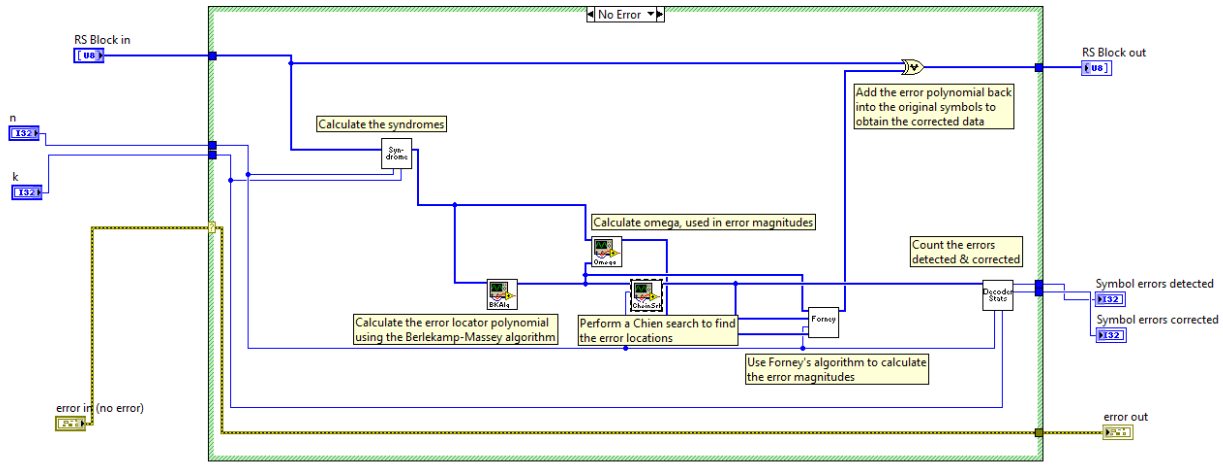


This is the wrapper for the encoder so that the inputs and outputs are bitstreams

5.4 Decoder Implementation

RS Decoder

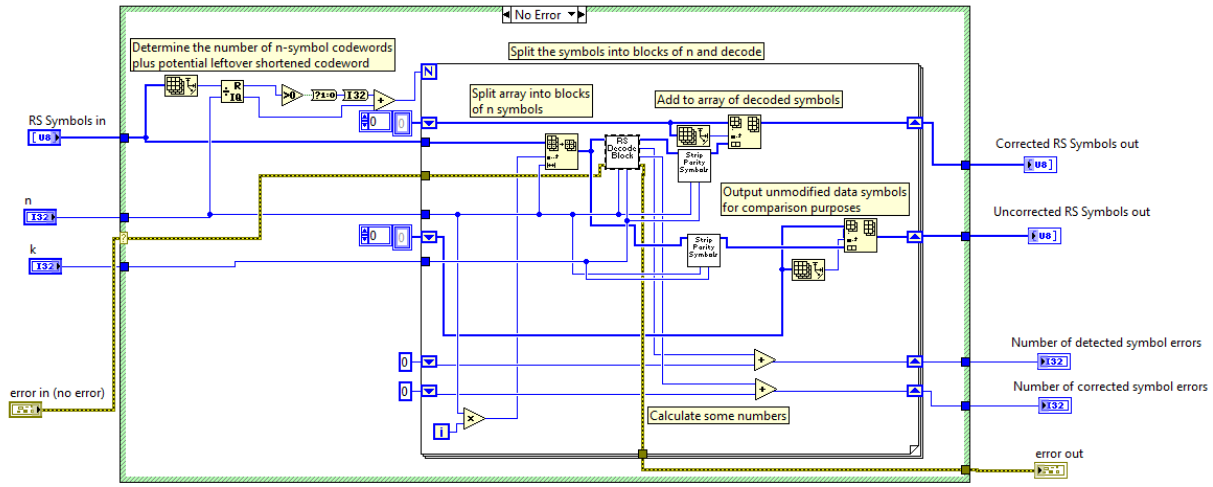
rs_decoder.vi



The decoder VI pulls the syndrome calculation, error locator polynomial calculation (Berlekamps algorithm), error location and error magnitude (Forney's algorithm) calculations together to produce the corrected block of k symbols out.

RS Block Decode

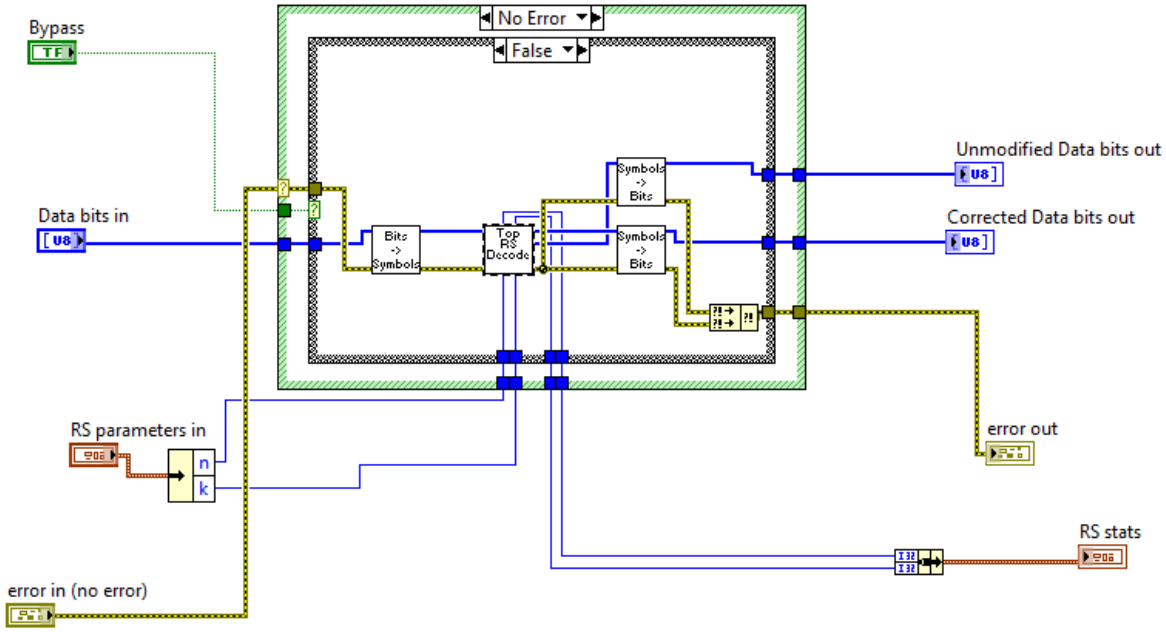
rs_top_decoder.vi



This vi takes an array of elements in GF(256) and splits them into blocks of n to be decoded.

RS Top-level Decoder

top_rs_dec.vi

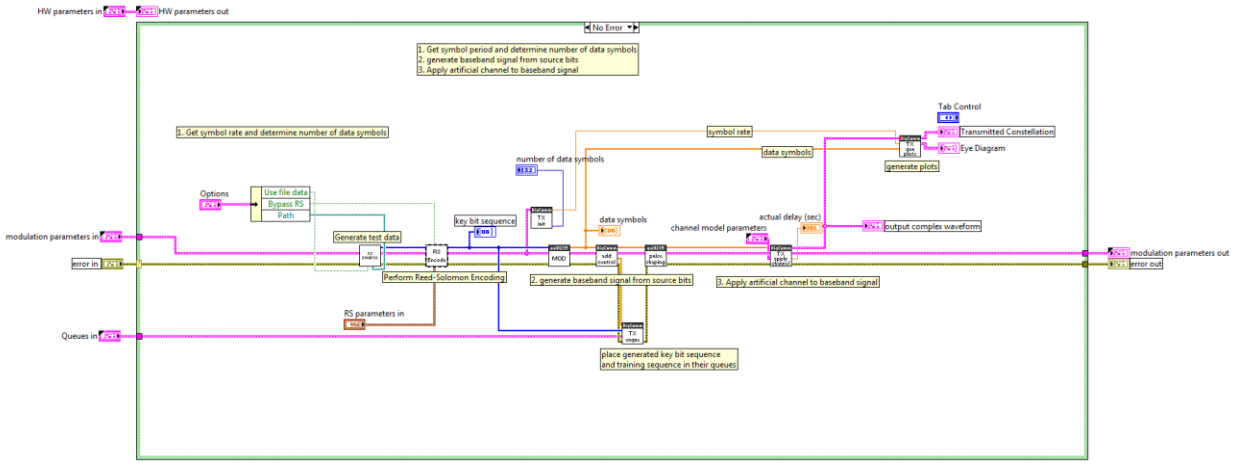


This is the wrapper for the decoder so that the input is the encoded bitstream and the output is the decoded bitstream.

5.5 Transmitter and Receiver

Transmitter

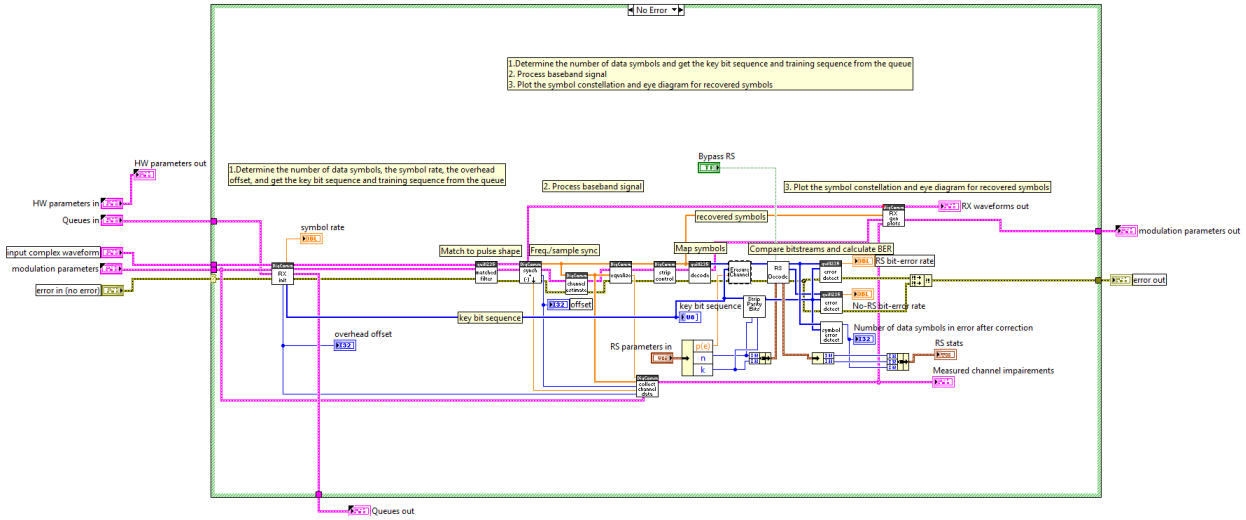
transmitter.vi



This VI has the QPSK transmit chain from the lab, with the RS encoder block inserted prior to symbol mapping.

Receiver

receiver.vi

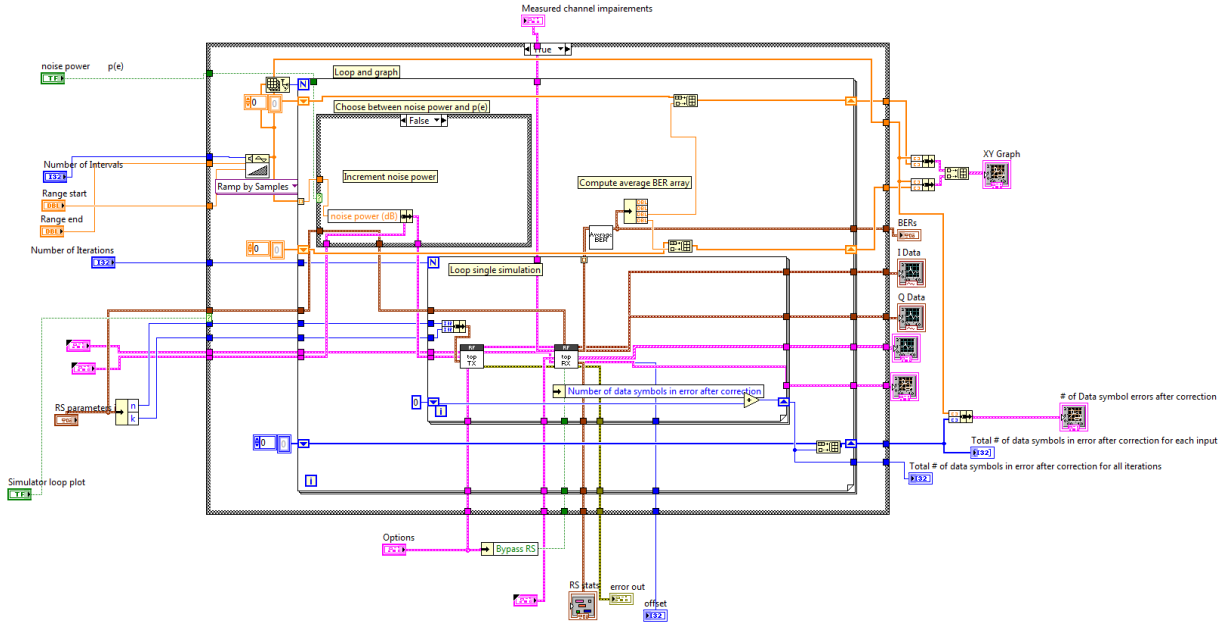


This VI has the QPSK receive chain from the lab, with the erasure channel block inserted after the decoder and also the RS decoder inserted after the symbol demapper.

5.6 Top-level VIs

Simulator

rs_simulator.vi

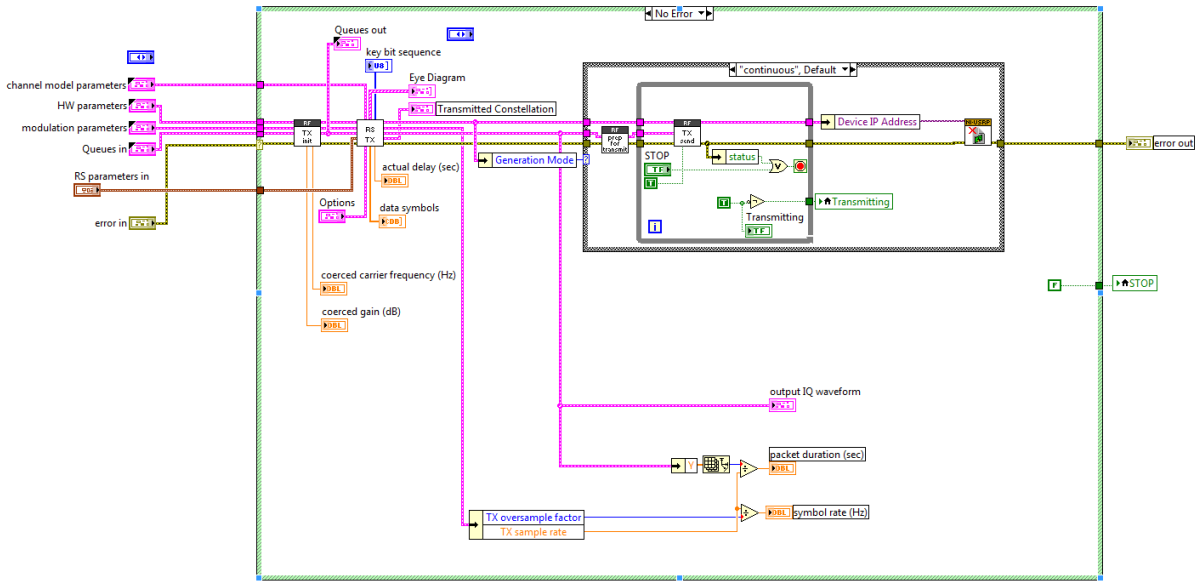


The simulator is used to tie the transmitter and receiver together with an artificial channel and simulate transmitting and receiving packets. It either can run the simulator for static settings or, if loop-plot is

enabled, will vary the input settings for either noise-power for AWGN or p(e) for an erasure channel and plot the corresponding BER curves.

Transmitter

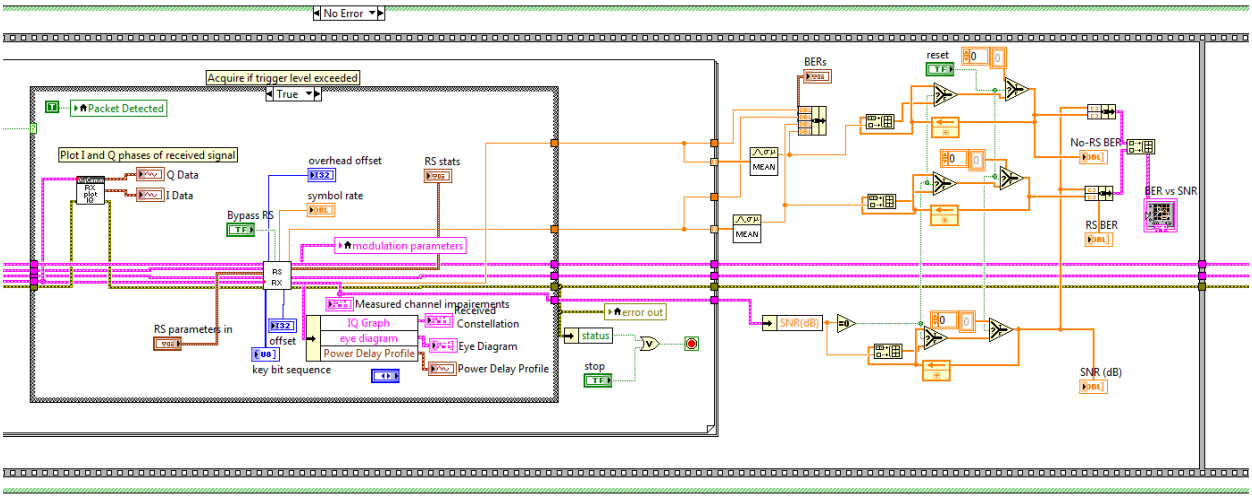
top_tx.vi



This VI is used to transmit data using the physical USRP.

Receiver

top_rx.vi



This VI is used to receive data using the physical USRP. It can also plot points for BER vs SNR.

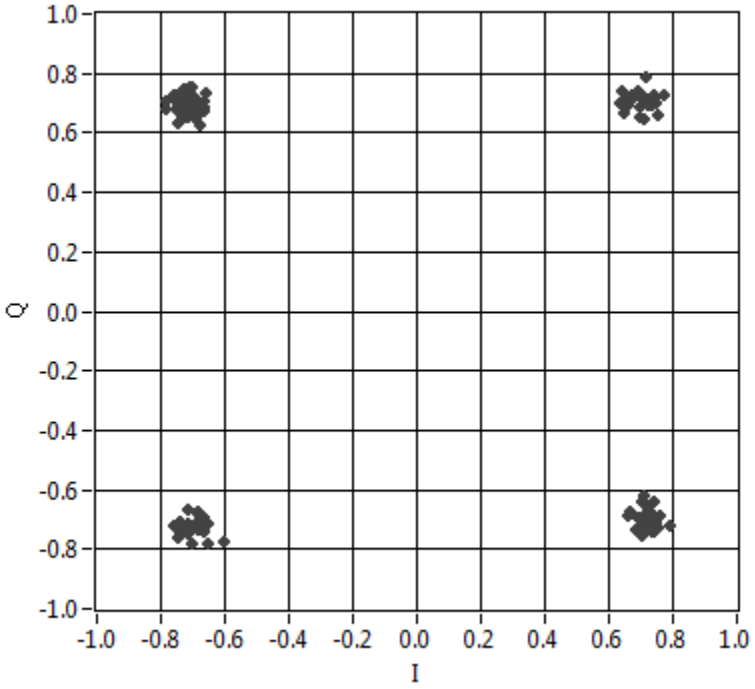
6 Performance Result

It should be noted that all simulations and hardware experiments used the following settings:

Message Length (data)	128 bits	**n**	32
Sample rate	4M Samples/sec.	**k**	16
Oversample factor	20 Samples/symbol	**Pulse shape**	Root-raised Cosine
Symbol rate	200K Symbols/s	**Center frequency**	915.0 MHz

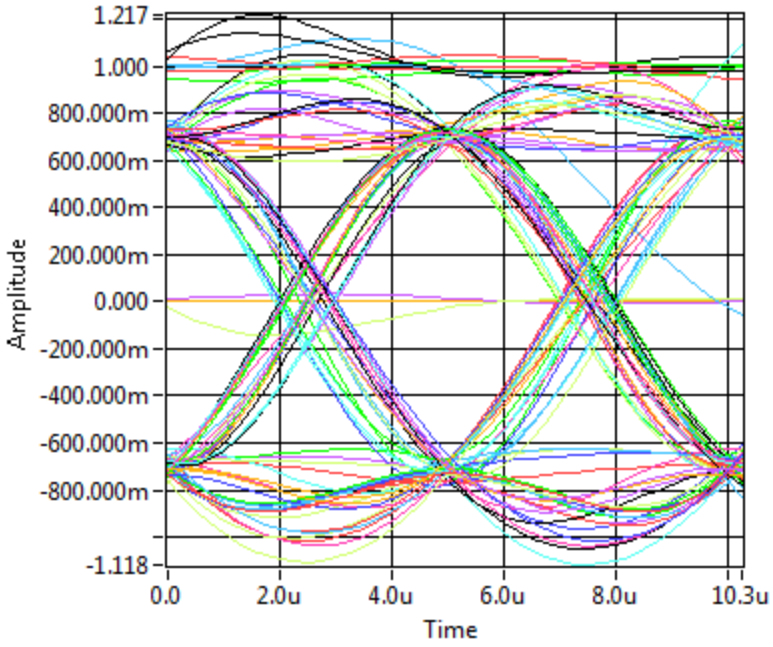
6.1 Single Run

The following is a received signal constellation:



This matches the expected transmitted QPSK signal constellation with the energy normalized to 1.

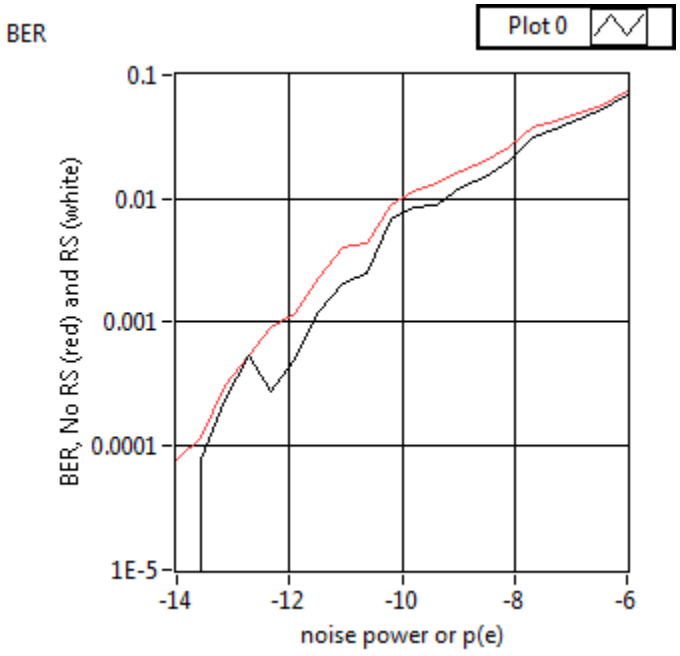
The eye diagram follows:



This also matches the expected eye diagram. The period matches the symbol period.

6.2 AWGN Channel Performance

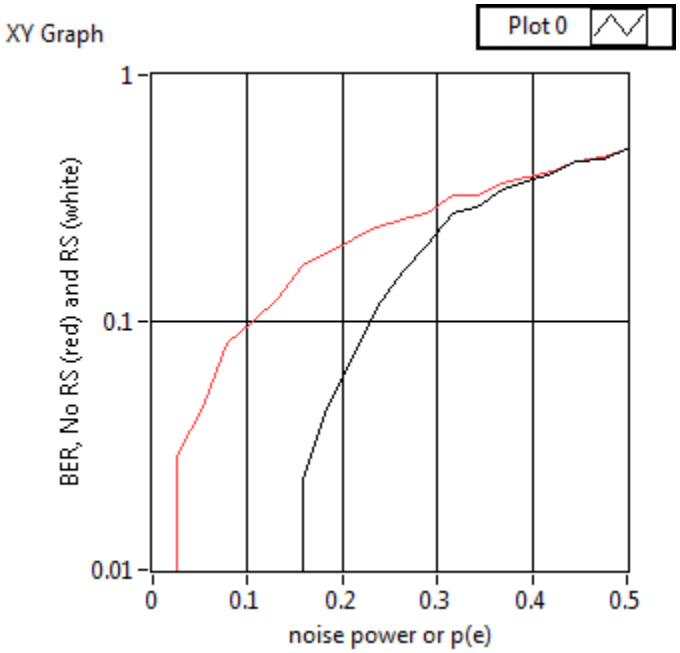
The following figure plots the BER versus noise power (N0) in an AWGN channel using simulation. The Reed-Solomon code used was n=32, k=16.



There is not a significant improvement using Reed-Solomon coding in an AWGN channel. This result is not surprising given that Reed-Solomon codes work well in correcting burst errors, where many bits in sequence are in error. In the AWGN case, bit errors happen randomly, so the burst-error correcting capability is not fully utilized (just 8 bit errors spread across 8 symbols will max out the error correction capability).

6.3 Binary Erasure Channel Performance

The following figure plots the BER versus p(e), the probability of a symbol erasure in a Binary Erasure Channel using simulation. The Reed-Solomon code used was n=32, k=16.



There is a significant reduction in the BER, particularly when p(e) < 0.25. This is not surprising since the code is able to correct up to T=8 symbols, or one quarter of the message symbols in error.

6.4 Power Spectral Density

Power spectral density is given below for two captures on the USRP using antennas, one without any error-correction coding and one utilizing Reed-Solomon Coding. The x axis is in units of Hz away from the center frequency (915 MHz) and the Y axis is in units of dB.

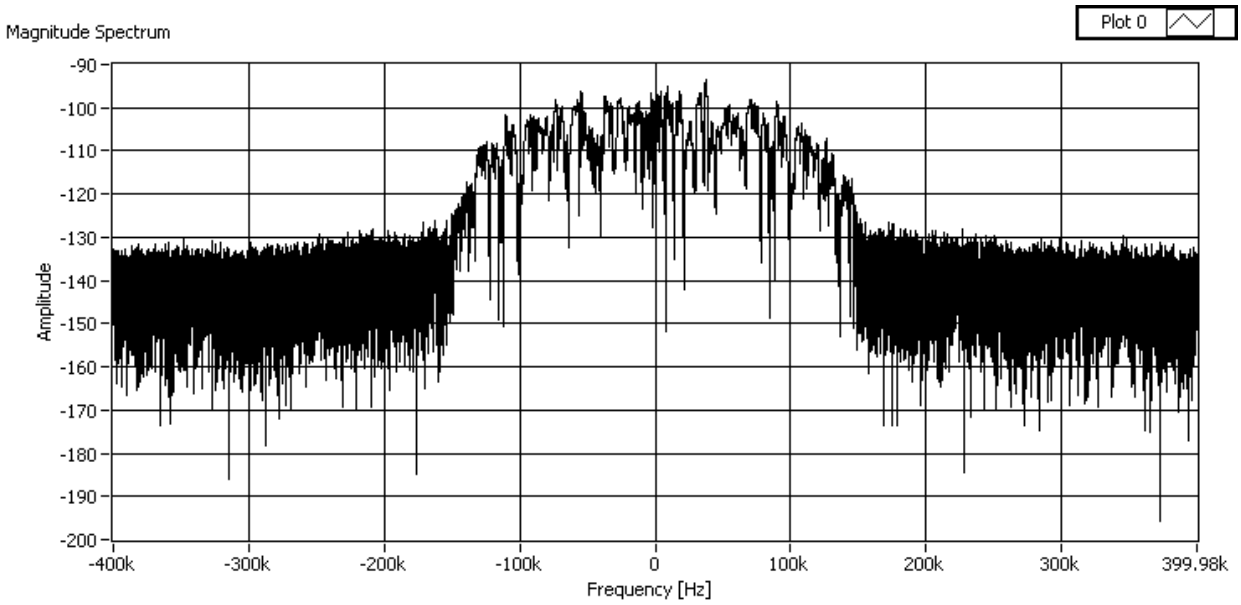


Figure 1: Power Spectral Density without RS-Coding

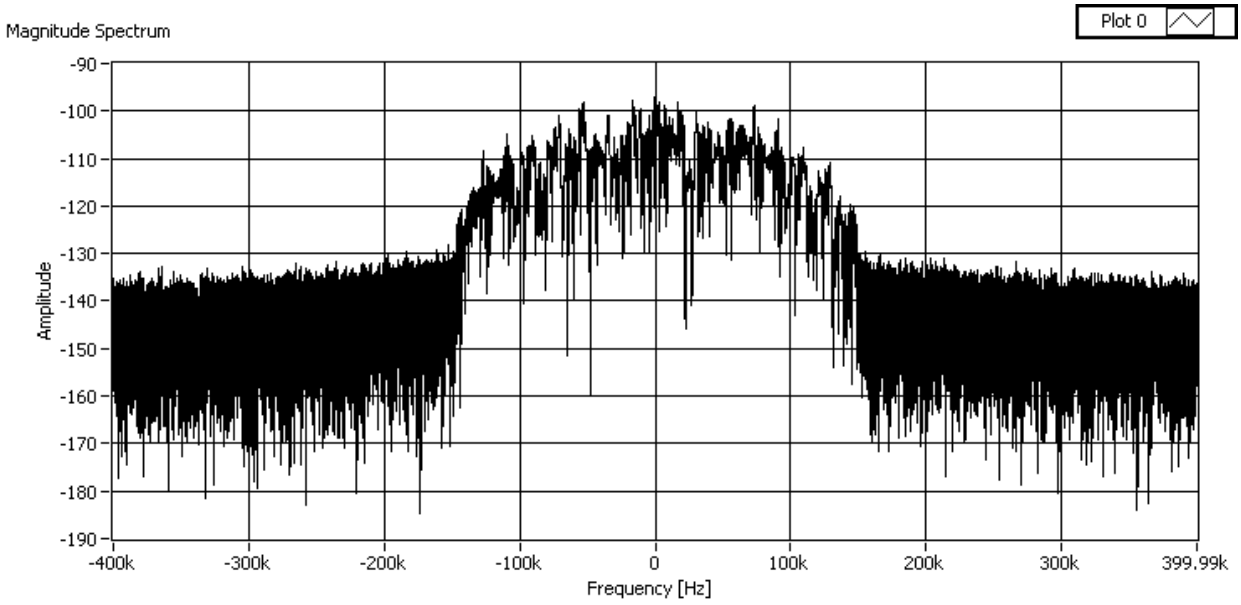


Figure 2: Power Spectral Density with RS-Coding

From the figure, it is observed that the bandwidth of the signal is around 200KHz, which is also the symbol rate. There is no noticeable difference when Reed-Solomon coding is added.

7 References

[1] Polynomial Codes over Certain Finite Fields

Reed, I.S. and Solomon, G.

[2] RS Coding Explanation & Implementation

C.K.P. Clarke

http://downloads.bbc.co.uk/rd/pubs/whp/whp-pdf-files/WHP031.pdf

[3] An introduction to Reed-Solomon codes: principles, architecture and implementation

Martyn Riley and Iain Richardson

http://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html

[4] Reed Solomon Codes

Joel Sylvester

http://www.csupomona.edu/~jskang/files/rs1.pdf

[5] Reed Solomon Codes

Bernard Sklar

http://hscc.cs.nthu.edu.tw/~sheujp/lecture_note/rs.pdf

[6] 802.16-2004 Specification

IEEE

http://standards.ieee.org/getieee802/download/802.16-2004.pdf

[7] The Finite Field GF(2^8)

(GF Multiplier Implementation)

Neal R. Wagner