CRC and Reed Solomon ECC

Jeff Reid April 18, 1995

Table of Contents

1. Introduction

This document describes CRC and Reed Solomon ECC processes and implementation. CRC is an acronym for Cyclic-Redundancy Code. ECC is an acronym for Error Correction Code. Reed Solomon ECC is the focus of this document.

CRC and ECC are used in error detection and correction algorithms to improve the reliability of transmitted data. This is accomplished by adding information to data before transmission, and using that extra information to check and correct any errors when that data is received.

2. Terminology and Basics

2.1 Terminology

This document uses expressions as defined in the following:

Raw Data - The original data to be transmitted.
Redundant Data - The additional data appended to original data before transmission.
Parities - Redundant Data created by ECC encode process.
Syndromes - The primary values generated and used by ECC decode process.
Encoded Data - Raw data and redundant data combined.
Encode - The process of generating redundant data and appending it to original data.
Decode - The process of detecting and/or correcting errors in encoded data.
Transmit - The process of sending or writing data.
Receive - The process of receiving or reading data.
Offset - The offset within a string, from 0 through string size -1.
Location - Reverse of offset, (string size -1) - offset.
Locator - Field primitive raised to a multiple of "location" power.

2.2 CRC and ECC Process

CRC and ECC use the same basic process. Raw data is encoded. The encoded data is transmitted. The encoded data is received. The received data is decoded, and the original data, if error free, or corrected, is obtained. An error detection scheme requires the retransmission of data if an error occurs. An error correction scheme can detect and correct errors, avoiding the need for retransmission.

2.3 Field Math

CRC and Reed Solomon ECC are based on field math. A field is a mathematical model where linear algebra works. A mathematical model redefines the operators add and multiply. The operators subtract and divide are defined as the inverse of add and multiply operations, respectively. A mathematical model also defines its domain, the set of numbers that exist in the model. A mathematical model is a field if it meets the requirements to make linear algebra work. One requirement is that all operations (except divide by zero) produce numbers within the

domain of the model. Other requirements include these properties: associative [(a+b)+c=a+(b+c), (ab)c=a(bc)], communative [a+b = b+a, ab = ba], and distributive [(a+b)c=ac+bc].

3. Binary Field Math

3.1 Binary Field Numbers

Binary field math numbers are single bit numbers. The binary field math domain is the set {0,1}.

3.2 Binary Field Operators

The add operator is defined as exclusive or. Since exclusive or is its own inverse, the subtract operator is also exclusive or. The multiply operator is defined to be multiply. The divide operator is defined as divide. The following is all math operations possible with binary field math:

Add:	0+0=0	0+1=1	1+0=1	1+1=0
Subtract:	0-0=0	0-1=1	1-0=1	1-1=0
Multiply:	0*0=0	0*1=0	1*0=0	1*1=1
Divide:		0/1=0		1/1=1

Since subtraction is the same operation as addition, the "+" symbol could be used for both addition and subtraction.

3.3 Binary Field Polynomials

Binary polynomials have one bit coefficients. These polynomials can be represented in several ways: in polynomial notation, as a string of binary (0 or 1) digits, or as a string of hexadecimal digits. Example:

$$x^5+x^3+x+1 = 1x^5 + 0x^4 + 1x^3 + 0x^2 + 1x + 1 = 101011 = 2B$$

3.4 Binary Field Polynomial Operators

Binary polynomials are added by "adding" (exclusive or'ing) like term coefficients. Subtraction is defined as the inverse, which is the same as addition. Polynomials represented in string notation can simply be exclusive or'ed.

$1\ x^3 + 0\ x^2 + 1\ x + 0$	1010
$1\ x^3 + 1\ x^2 + 0\ x + 0$	1100
---	----
$0\ x^3 + 1\ x^2 + 1\ x + 0$	0110

The multiply operator is defined as polynomial multiplication, where coefficients of like terms are combined by exclusive or'ing. A long hand style technique can be used to implement multiply.

$$(x^2+x+1)(x^2+1) = x^4+x^3+ x^2+x^2 +x+1 = x^4+x^3+x+1$$

Note that $x^2 + x^2 = 0$ since exclusive or used to combine terms.

```
        1x^2+1x+1                   111
        1x^2+0x+1                   101
---------------             -----
        1   1  1                   111
    0   0   0                    000
1   1   1                   111
---------------             -----
1x^4+1x^3+0x^2+1x+1         11011
```

The divide operator is defined as the inverse of multiplication. Either polynomial or long hand techniques may be used to implement division. Note that in the long hand example, the divisor "111" goes into the first 3 bits of the dividend "110". In binary math division, only the leading bit of the divisor determines if the divisor "goes into" the leading bits of the dividend.

$(x^4+x^3+x+1) / (x^2+x+1) = x^2+1$

```
1x^4+1x^3+0x^2+1x+1     = 11011 = dividend
1x^2+1x+1               =   111 = divisor

     __101
111 |11011
     111
     011
     000
      111
      111
      000
```

3.5 Binary Field Polynomial Properties

The definition of a binary prime polynomial is a polynomial not exactly divisible by any other polynomial except for itself and 1. Some examples of binary prime polynomials in hexadecimal notation are: 2, 3, 7, B, D, 13, 19, 1F, 11D, 187, F01F, 1002D, 1000000AF.

An *n*-bit polynomial multiplied by an *m*-bit polynomial will always produce an (*n*+*m*-1)-bit product. An *n*-bit divisor divided into an *m*-bit dividend will always produce an (*m*+1-*n)*-bit quotient and (at most) an (*n*-1)-bit remainder.

4. CRC

CRC is a simple means of implementing error detection in transmitted data. The redundant data is the remainder produced by binary math division. A simple description of the process follows:

Define *m* as the number of bits of raw data, and *n* as the number of bits of redundant data. Select an *n*+1 bit divisor (for example, any *n*+1 bit binary prime polynomial) as a basis for the encode

and decode processes. To encode, add *n* zero bits to the raw data string, divide this *m*+*n* bit long dividend by the *n*+1 bit divisor, and replace the *n* zero bits with the remainder of the division. To decode, divide the *m*+*n* bit encoded data string by the same divisor, and check the remainder. If the remainder is not zero, an error exists in the encoded data, otherwise assume that the data is error free.

There is some chance that CRC will fail to detect an error. A simple approximation for this probability would be the odds of an error in the encoded data divided by 2^n.

Most floppy controllers implement CRC using the 17 bit binary polynomial 10001000000100001 (11021 hex) to produce 16 bits of redundant data. 11021 hex is the product of two primes, 3 and F01F. The 3 acts as an "odd" parity check, and the F01F handles general errors. The following example demonstrates this using hex notation on 2 bytes of raw data:

m	16	16 bits of raw data
n	16	16 bits of redundant data
Divisor	11021	17 bit divisor
Encode:		
Raw Data	9d71	
Dividend	9d710000	
Remainder	0001	(9d710000)/(11021) = 9421, rem=0001
Encoded Data	9d710001	
Decode:	(9d710001)/(11021) = 9421, rem = 0000 therefore no error.	

5. Finite Field Math

A finite field is any field (linear algebra works) with a finite (fixed size) domain. This document is only concerned with finite fields based on binary polynomials, since Reed Solomon ECC is based on these type of finite fields

5.1 Finite Field Numbers

Finite field math numbers are fixed size, multiple bit numbers. To compose a field with a domain made up of *n*-bit numbers, an *n*+1 bit binary prime polynomial is chosen. All math operations are implemented as binary polynomial operations, modulo the chosen polynomial. The domain of such a model will have 2^n numbers in its set, {0, 1, ...}. This model will have all the properties required to make it a field (linear algebra works with this model). In addition, a logarithm mapping exists for any finite field domain, which can optimize software implementation. Finite field math numbers are usually represented as hexadecimal strings.

Here is a list of a few binary prime polynomials and the bit size of the field numbers they define: 7 = 2-bit field, 19 = 4 bit field, 187 = 8 bit field, 1002d = 16 bit field, 1000000af = 32 bit field. There are many alternatives to these numbers, for example there are 30 different polynomials that can define an 8-bit field.

5.2 Finite Field Operators

Add and subtract operate the same as binary polynomial addition; exclusive or is used to add or subtract finite field math numbers. Since no size change occurs due to addition or subtraction, no modulo operation is required.

Multiplying can be defined as a two step process. First, the two finite field numbers are multiplied according to the rules of binary polynomial multiplication. This product will contain $2n$-1 bits. Second, this product is divided by the n+1 bit polynomial chosen for the field. The remainder of this division is considered to be the finite field math product; it will contain n bits. Hardware multiplication can be implemented as defined, in a single step using *and*'s and *xor*'s, since sub-product and modulo operations are independent. Software multiplication is usually implemented using log tables.

Division is defined as the inverse of multiplication. Dividing the variable a by b can thought of as trying to solve the equation: $c = a / b$. There is never a remainder, since this is a field. Hardware division is usually implemented by inverting the divisor, and then multiplying the dividend by the inverted divisor. Hardware inversion can be implemented several ways. The simplest is to use a lookup table. Exponentiating b to the 2^{n-2} power through repeated multiplication results in 1/b. Another is a repetitive process based on binary polynomial math (see section 7.3.1). Depending on the field, a single step process using less logic than a lookup table can be used that involves mapping into a special field, doing some math, and mapping back (see section 7.3.2). Software division is usually implemented using log tables.

Using the 3 bit number 7, to create a 2 bit field leads to the following facts and math tables:

```
2*2 mod 7       = 4 mod 7       = 3
2*3 mod 7       = 6 mod 7       = 1
3*3 mod 7       = 5 mod 7       = 2
(3*3 in binary = 11*11 = 101)
```

Add (Subtract)

	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Multiply

	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

Divide

	0	1	2	3
0	x	x	x	x
1	0	1	2	3
2	0	3	1	2
3	0	2	3	1

5.3 Finite Field Primitives

In any n-bit finite field, there exists a primitive. A primitive is a binary prime polynomial, that when raised to any power from 0 to 2^n-2 modulo the chosen n+1 bit binary prime polynomial, will generate all of the numbers in the field except 0. A log table can be created based on this primitive to implement multiply and divide.

The primitive for the 2 bit field defined by the 3 bit number 7, is 2.

$2^0 = 1 \bmod 7 = 1$
$2^1 = 2 \bmod 7 = 2$
$2^2 = 4 \bmod 7 = 3$

There are three 5-bit binary primes, 13, 19, and 1F. These primes can be used to create three different 4-bit finite fields. The primitive for 13 and 19 is 2. The primitive for 1F is 3. The following tables list the field domains in primitive power order.

Poly Primitive Field Domain in primitive power order

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	exponent
13	02	01	02	04	08	03	06	0c	0b	05	0a	07	0e	0f	0d	09	
19	02	01	02	04	08	09	0b	0f	07	0e	05	0a	0d	03	06	0c	
1f	03	01	03	05	0f	0e	0d	08	07	09	04	0c	0b	02	06	0a	

5.4 Finite Field Logarithms

Logarithms are normal decimal numbers, but operated on modulo 2^n-1. Adding and subtracting logarithms (modulo 2^n-1) can be used to multiply and divide. Multiplication of logarithms (modulo 2^n-1) can be used for exponentiation. Logarithm tables can be built from primitive power order tables like the ones listed in the previous section; the exponent above is the logarithm for the number below. For example, in the field based on 19, log 0F = 6, and log 03 = 12.

Using the 4-bit field based on 19, multiply 0F times 03 using defined way and then using logs:

```
03*0F mod 19   = 11 mod 19 = 08

log 0f = 6, log 03 = 12
(log 0F + log 03) mod 15 = (6+12) mod 15 = 18 mod 15 = 3
antilog 3 = 08
```

5.5 Finite Field Polynomials

Finite field polynomials have *n*-bit finite field numbers as coefficients, where *n* is any integer value. These polynomials can also be represented as strings of hexadecimal digits, with spaces separating the coefficients (and zeroes used as place holders).

$c\ x^3 + 2\ x^2 + 8\ x + 5$ = c 2 8 5

5.6 Finite Field Polynomial Operators

Finite field polynomial operators follow the same rules as binary polynomial operators. The only difference is the size of their coefficients. Finite field polynomials are added by "adding" (exclusive or'ing) like term coefficients. Subtraction is defined as the inverse, which is the same

as addition. The multiply operator is defined as polynomial multiplication, where coefficients of like terms are combined by exclusive or'ing. Division is defined as the inverse of multiplication.

Here are some sample math operations using the 4-bit field defined by hex 19.

Add: $(c\ x^3 + 2\ x^2 + 8\ x + 5) + (7\ x^2 + 3\ x + a)$

```
  c 2 8 5
+   7 3 a
  ---------
  c 5 b f   = c x^3 + 5 x^2 + b x + f
```

Multiply: $(c\ x^3 + 2\ x^2 + 8\ x + 5) * (7\ x^2 + 3\ x + a)$

```
      c 2 8 5
    *   7 3 a
    ---------
      5 d 6 9
    d 4 1 f
  f e a 2
  ---------
  f 3 b e 9 9
```

Divide: $(c\ x^3 + 2\ x^2 + 8\ x + 5) / (7\ x^2 + 3\ x + a)$

```
            ---------   = 7x + 6
7 3 a     | c 2 8 5
            c 9 4
            ---------
              b c 5
              b a e
              -------
                6 b     = 6x + b (remainder)
```

5.7 Finite Field Polynomial Properties

Similar to binary polynomials, an *n*-term polynomial multiplied by an *m*-term polynomial will always produce an (*n*+*m*-1)-term product. An *n*-term divisor divided into an *m*-term dividend will always produce an (*m*+1-*n)*-term quotient and (at most) an (*n*-1)-term remainder.

6. Reed Solomon ECC

Reed Solomon ECC is an error detection and correction process based on finite field polynomial math. Similar to CRC, the redundant data is the remainder produced by division. Unlike a simple CRC scheme, the ECC algorithm can detect, locate, and correct errors. Creating an ECC involves choosing a field and a generator polynomial, and implementing encode and decode processes.

6.1 Choosing a Field

A field is chosen so that its numbers are big enough (have enough bits) to contain all the variables needed to perform a correction, mainly error values and location values. If the largest variable fits in *f* bits, then an *f*+1 bit polynomial is chosen for the basis of the field. Many tape

devices use 8-bit fields, since they are correcting 8-bit bytes, and their tape formats designed so that location values never require more than 8 bits. Some disk devices use larger fields, such as 11 bits, to handle a larger range of location values.

Another criteria for choosing a field is its primitive, which should be 2. A field with a primitive of 2 is easier to implement than other fields.

6.2 Choosing a Generator Polynomial

A generator polynomial is an *n*+1 term polynomial with *n* successive powers of its field primitive as roots. A generator polynomial can be calculated by multiplying (x+root) for each of its roots. For example, using the 4-bit field based on hex 19, here are two polynomials that would produce a 4 term remainder, the first using the powers 0 through 3, the second using powers 6 through 9.

$$(x+1)(x+2)(x+2^2)(x+2^3) = 1x^4 + fx^3 + 4x^2 + 5x + f$$

$$(x+2^6)(x+2^7)(x+2^8)(x+2^9) = 1x^4 + 3x^3 + cx^2 + 3x + 1$$

The number of terms required is determined by the correction process. Error correction involves determining error location and value. Two redundancy terms are required for each potential error to be located and corrected. If error locations are determined through an independent process (such as CRC on some tape devices), then only one term per error is required.

The second example polynomial listed above is self-reciprocating; reversing the order of its coefficients results in the same polynomial. Self-reciprocating generator polynomials are used to optimize the encode process. To create a self-reciprocating polynomial with an even number of *n*-bit roots, choose powers around $(2^n-1)/2$. In the example, *n* is 4, $(2^n-1)/2 = 7½$.

To create a self-reciprocating polynomial with an odd number of *n*-bit roots, choose powers around 0 (modulo 2^n-1). Using the 4-bit field based on hex 19, a three term remainder could be produced using powers -1, 0, and 1. Note that -1 mod 15 = 14.

$$(x+2^{-1})(x+2^0)(x+2^1) = (x+c)(x+1)(x+2) = 1x^3 + fx^2 + fx + 1$$

6.3 Encoding

ECC encoding is based on division. Data to be encoded is treated as a large *m*-term finite field polynomial. This *m*-term polynomial is appended with *n* zeroes (equivalent to multiplying by x^n), and divided by the *n*+1 term generator polynomial, producing an *n*-term remainder. The encoded data string is the *m*-term raw data string appended with the *n*-term remainder. The remainder terms are referred to as parities. The encoded string is a multiple of the generator polynomial, and dividing an encoded string by its generator polynomial produces a remainder of zero.

For example, choose the 4-bit field based on hex 19, and the 5-term generator polynomial:

$$(x+1)(x+2)(x+4)(x+8) = x^4 + fx^3 + 4x^2 + 5x + f = 1\ f\ 4\ 5\ f$$

This produces a 4-term remainder. Then encode the 6-term string "f 3 a 7 5 e":

```
                    ? ? ? ? ? ?
1 f 4 5 f  |  f 3 a 7 5 e 0 0 0 0
               . . . . .
             -----------
               c f b 2
```

Append the 4-term remainder to get the 10-term encoded string "f 3 a 7 5 e c f b 2". If this 10-term string is divided by "1 f 4 5 f", the remainder is zero. If an encoded string remainder is not zero, then an error has been detected.

6.4 Decoding

Decoding is the complex part of ECC, since it is used to detect, locate, and correct errors. To correct each erroneous term of an encoded string, ECC decoding needs to determine 2 things: error location and error value. Each error is corrected by adding (exclusive-or) the determined error value to the term at the determined location. If data is encoded with *n* terms of redundancy, then *n*/2 errors can be located and corrected. (e.g. 4 redundancy terms are needed to fix 2 errors.)

Some encoding schemes use additional redundancy to locate errors independently of ECC (such as CRC, or a multi-level ECC). When error locations are pre-determined, ECC decoding only needs to determine an error value for each error, and can fix *n* errors with *n* parities.

Other schemes may involve a combination of known and unknown error locations. This happens when the additional redundancy doesn't adequately detect (or mis-corrects) errors.

Detection and correction are done in six basic steps:

1. Generate syndromes, if all zeroes, then no error is detected.
2. Modify syndromes if some error locations are pre-determined.
3. Generate unknown locator polynomial from (modified) syndromes.
4. Compute unknown locators by solving unknown location polynomial.
5. Compute error values from locator and syndrome values.
6. Correct errors in data.

If all error locations are pre-determined, only steps 1, 5, and 6 are used. If no errors are pre-determined, step 2 is skipped. A detection only scheme can stop after step 1. An modified encode can also be used for error detection. The *m*+*n* term encoded string can be divided by the generator polynomial and the remainder checked to be all zero. Hardware schemes usually append the *m*+*n* term string with an additional *n* zeroes, and then divide the *m*+*n*+*n* term string producing a new set of parities; this is called extended encoding or re-encoding. If the parities are non-zero, an error has been detected.

6.4.1 Decoding Naming Conventions

Symbol	Meaning
O	The original value for a term in the encoded string (original).
D	The actual value for a term in the encoded string (data).
d	Number of terms in the encoded string.
E	Error value (xor of O and D)
e	Number of errors in encoded string
P	Field primitive, usually P=2
R	A root of generator polynomial
ofs	offset = index into string
loc	location = reverse of offset = (d-1)-offset
kloc	known location
uloc	unknown location
k	number of known locations
u	number of unknown locations
S	Syndrome
s	number of syndromes (same as number of roots)
L	Locator = primitive raised to some multiple of location
K	Known locator
U	Uknown locator

6.4.2 Generating Syndromes

Syndrome generation is based on division. The string to be decoded is made up of *m* data terms, and *n* parity terms. This string is treated as an *m*+*n* term polynomial, and is divided by each factor (x+root) of the generator polynomial. The remainder of each division is a syndrome. Since the encoded data string is a multiple of the generator polynomial, the string is also a multiple of each factor of the generator polynomial and all syndromes will be zero if there are no errors. Non-zero syndromes indicate the presence of an error.

Using the encode example, assume an error occurred in the 10 term string at offset 3, changing the value 7 to d, resulting in the string “f 3 a d 5 e c f b 2” instead of "f 3 a 7 5 e c f b 2". To generate the syndromes, divide the 10-term string by each of the factors of the generator polynomial: (x+1), (x+2), (x+4), and (x+8) and keep the remainders (“%” means modulo):

```
S0 = syndrome 0 = f 3 a d 5 e c f b 2 % 1 1 = a
S1 = syndrome 1 = f 3 a d 5 e c f b 2 % 1 2 = 2
S2 = syndrome 2 = f 3 a d 5 e c f b 2 % 1 4 = 7
S3 = syndrome 3 = f 3 a d 5 e c f b 2 % 1 8 = 6
```

6.4.3 Syndrome Equations

For each generator polynomial root, the division steps used to calculate a syndrome can be written in equation form:

$$S = D_1 R^{loc1} + D_2 R^{loc2} + \ldots + D_d R^{locd}$$

If there are no errors, the D's can be replaced with O's and the syndrome is zero. If one error was present in the data at location *z*:

$$\begin{aligned} S &= O_1 R^{loc1} + \ldots + (O^z + E) R^{locz} + \ldots + O_d R^{locd} \\ &= O_1 R^{loc1} + \ldots + O^z R^{locz} + \ldots + O_d R^{locd} + E R^{locz} \\ &= 0 + E R^{locz} \\ &= E R^{locz} \end{aligned}$$

Multiple errors can be separated out in a similar way, and the general form for a syndrome equation based on generator polynomial factor $(x + R_i)$ becomes:

$$S_j = E_1 P^{j\ loc1} + E_2 P^{j\ loc2} + \ldots + E_e P^{j\ loce}$$

Where $j = \log(R_i)$ or $P^j = R_i$.

j ties in the relationship between a generator polynomial factor $(x+R_i)$ and a specific syndrome, S_j. Normally $j = i$ except when a reciprocal generator polynomial is used.

In the example, P = 2, $j = i$, and the error occurs at offset 3, or location 6 ([string size -1] - offset = [10-1] - 3 = 9-3 = 6), and the error value is 7+d = a:

$$\begin{aligned} S_0 &= a * 2^{0 * 6} = a \\ S_1 &= a * 2^{1 * 6} = 2 \\ S_2 &= a * 2^{2 * 6} = 7 \\ S_3 &= a * 2^{3 * 6} = 6 \end{aligned}$$

6.4.4 Syndrome to Locator Conversion

Given a set of *e* error values $\{E_1 , E_2 , \ldots E_e\}$, and *e* locators, $\{ P^{loc1} , P^{loc2} , \ldots P^{loce} \}$ (P is the field primitive), generate a locator polynomial that will be zero whenever X is one of the *e* locators and let A_i represent the coefficients of the locator polynomial:

$$\begin{aligned} &(X+P^{loc1})\ (X+P^{loc2})\ \ldots\ (X+P^{loce}) &&= 0 \\ &X^e + A_1 X^{e-1} + A_2 X^{e-2} + \ldots + A_e &&= 0 \end{aligned}$$

Generate *e* equations by substituting $P^{loc\ i}$ for the X's and multiplying by E_i , for $i = 1$ to *e* :

$$\begin{aligned} &E_1 P^{(e)loc1} + A_1 E_1 P^{(e-1)loc1} + \ldots + A_e E_1 = 0 \\ &E_2 P^{(e)loc2} + A_1 E_2 P^{(e-1)loc2} + \ldots + A_e E_2 = 0 \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \ldots \\ &E_e P^{(e)loce} + A_1 E_e P^{(e-1)loce} + \ldots + A_e E_e = 0 \end{aligned}$$

Noting that the sum of the E and P components of each vertical column is a syndrome equation, sum up the above equations (by column):

$$S_e + A_1 S_{e-1} + \ldots + A_e S_0 = 0$$

Which can be re-written as:

$$A_1\ S_{e-1} \quad + \ldots + A_e\ S_0 = S_e$$

This process converts the locator polynomial equation into a syndrome equation. Do this process *e* times, substituting $P^{loc\ i}$ for the X's, and multiplying by $E_i\ P^{(z)\ loc\ i}$ for $z = j$ to $j+e-1$. This will generate enough equations to calculate the *A*'s given the syndromes:

$$\begin{aligned}
&A_1\ S_{j+e-1} && + \ldots + A_e\ S_j && = S_{j+e}\\
&A_1\ S_{j+e} && + \ldots + A_e\ S_{j+1} && = S_{j+e+1}\\
&A_1\ S_{j+e+1} && + \ldots + A_e\ S_{j+2} && = S_{j+e+2}\\
& && \ldots &&\\
&A_1\ S_{j+2e-2} && + \ldots + A_e\ S_{j+e-1} && = S_{j+2e-1}
\end{aligned}$$

Converting syndromes into locators, requires ($s \geq 2e$) syndromes to determine *e* locators. Given sufficient syndromes, arrange the syndromes into the equation format listed above, and solve the simultaneous equations for the *A*'s. Once the *A*'s are calculated, solve the locator polynomial using the *A*'s as coefficients to calculate the locators.

In the example, there is one error occurs at offset 3, or location 6 and the error value is 7+d = a, and $S_0 = a$, $S_1 = 2$, $S_2 = 7$, $S_3 = 6$. If A_1 represents the one coefficient in this case, then:

$$A_1\ S_0 = S_1$$
$$A_1\ S_1 = S_2$$
$$A_1\ S_2 = S_3$$

$$A_1 = S_1/S_0 = S_2/S_1 = S_3/S_2 = 2/a = 7/2 = 6/7 = f$$

$$X + A_1 = 0$$
$$X + f = 0$$
$$X = f = \text{locator of error.}$$
$$\log(X) = \log(f) = 6 = \text{location of error.}$$

6.4.5 Modifying Syndromes

This process is used when there is a mixture of known and unknown locations. It takes advantage of the relationship between locators and syndromes explained in section 6.4.4. Given a set of *k* known error values $\{E_1, E_2, \ldots E_k\}$, and *u* unknown error values $\{F_1, F_2, \ldots, F_u\}$, a syndrome equation can be written as:

$$S_j = E_1\ P^{j\ kloc1} + \ldots + E_k\ P^{j\ klock} + F_1\ P^{j\ uloc1} + \ldots + F_u\ P^{j\ ulocu}$$

Use T_j to represent the known part of syndrome, and U_j to represent unknown part:

$$T_j = E_1 P^{j\ kloc1} + \ldots + E_k P^{j\ klock}$$
$$U_j = F_1 P^{j\ uloc1} + \ldots + F_u P^{j\ ulocu}$$
$$S_j = T_j + U_j$$

Calculate the coefficients B_i to a locator polynomial based on the known locators, $P^{(kloc\ i)}$'s:

$$(X+P^{kloc1})\ (X+P^{kloc2})\ \ldots\ (X+P^{klock}) = 0$$
$$X^e + B_1 X^{e-1} + B_2 X^{e-2} + \ldots + B_k = 0$$

Based on the relationship explained in section 6.4.4, the following holds for any *j* :

$$T_{j+k} + B_1 T_{j+k-1} + B_2 T_{j+k-2} + \ldots + B_k T_j = 0$$

Define the modified syndrome M_j using the *B*'s:

$$M_j = S_{j+k} + B_1 S_{j+k-1} + B_2 S_{j+k-2} + \ldots + B_k S_j$$

Substitute ($T_j + U_j$) for S_j , note that the T's will drop out due to relationship with the B's:

$$M_j = (T_{j+k} + U_{j+k}) + B_1 \ldots + B_k (T_j + U_j)$$
$$= T_{j+k} + B_1 \ldots + B_k T_j + U_{j+k} + B_1 \ldots + B_k U_j$$
$$= 0 + U_{j+k} + B_1 \ldots + B_k U_j$$
$$M_j = U_{j+k} + B_1 U_{j+k-1} + \ldots + B_k U_j$$

Expand this last equation into a vertical format; and define N_i as the sum of each column:

$$\begin{array}{llll}
B_k\ F_1\ P^{(j\ )\ uloc1} & + B_k\ F_2\ P^{(j\ )\ uloc2} & + \ldots + & B_k\ F_u\ P^{(j\ )\ ulocu} \\
B_{k-1}\ F_1\ P^{(j+1)\ uloc1} & + B_{k-1}\ F_2\ P^{(j+1)\ uloc2} & + \ldots + & B_{k-1}\ F_u\ P^{(j+1)\ ulocu} \\
\quad \ldots & & & \\
F_1\ P^{(j+k)\ uloc1} & + F_2\ P^{(j+k)\ uloc2} & + \ldots + & F_u\ P^{(j+k)\ ulocu} \\
\hline
N_1 & + N_2 & + \ldots + & N_u
\end{array}$$

Define: $G_i = N_i / P^{(j\ )\ uloc\ i}$

Then:
$$M_j = N_1 + N_2 + \ldots + N_u$$
$$M_j = G_1 P^{(j\ )\ uloc1} + G_2 P^{(j\ )\ uloc2} + \ldots + G_u P^{(j\ )\ ulocu}$$

Interpreting the G's as "error" values, this last equation has the same form as the standard syndrome equation in section 6.4.3. This means that unknown locations can be determined by using the conversion process described in section 6.4.4.

The maximum number of unknown locators *u*, that can be determined from a set of *s* syndromes and *k* known locators is $u = (s - k) / 2$. Calculate the B's from the known locators, and then generate 2 *u* modified syndromes:

$$
\begin{aligned}
M_j &= S_{j+k} + B_1 S_{j+k-1} + B_2 S_{j+k-2} + \ldots + B_k S_j \\
M_{j+1} &= S_{j+1+k} + B_1 S_{j+k} + B_2 S_{j+k-1} + \ldots + B_k S_{j+1} \\
M_{j+2} &= S_{j+2+k} + B_1 S_{j+k+1} + B_2 S_{j+k} + \ldots + B_k S_{j+2} \\
\ldots &= \ldots
\end{aligned}
$$

Follow the process described in section 6.4.4 to determine the *u* locators. If this process fails (can't solve the simultaneous equations, or bad locators are calculated), then decrement *u* and repeat the process. If *u* decrements to zero, then quit, there are no unknown errors.

6.4.6 Computing Error Values

Sections 6.4.2, 6.4.4, and 6.4.5 describe the process of generating syndromes, modifying the syndromes if any error locators are pre-determined, using this information to determine the coefficients to an unknown locator polynomial, and solving the polynomial to compute the unknown locators. After these steps, all syndromes and locators are calculated. Error locations can be calculated by taking the log of the locators.

From section 6.4.3, the general form for a syndrome equation can be written as:

$$E_1 P^{j\ loc1} + E_2 P^{j\ loc2} + \ldots + E_e P^{j\ loce} = S_j$$

Where $j = \log(R_i)$ or $P^j = R_i$.

Error values can be calculated by solving *e* simultaneous syndrome equations:

$$
\begin{aligned}
E_1 P^{(j\ )\ loc1} + E_2 P^{(j\ )\ loc2} + \ldots + E_e P^{(j\ )\ loce} &= S_j \\
E_1 P^{(j+1\ )\ loc1} + E_2 P^{(j+1\ )\ loc2} + \ldots + E_e P^{(j+1\ )\ loce} &= S_{j+1} \\
\ldots & \\
E_1 P^{(j+e-1)\ loc1} + E_2 P^{(j+e-1)\ loc2} + \ldots + E_e P^{(j+e-1)\ loce} &= S_{j+e-1}
\end{aligned}
$$

These equations can be re-written using the generator polynomial roots, R_i :

$$
\begin{aligned}
E_1 R_0^{loc1} + E_2 R_0^{loc2} + \ldots + E_e R_0^{\ loce} &= S_j \\
E_1 R_1^{loc1} + E_2 R_1^{loc2} + \ldots + E_e R_1^{\ loce} &= S_{j+1} \\
\ldots & \\
E_1 R_{e-1}^{loc1} + E_2 R_{e-1}^{loc2} + \ldots + E_e R_{e-1}^{\ loce} &= S_{j+e-1}
\end{aligned}
$$

6.4.7 Correcting Errors

After computing the error values as described in section 6.4.6, data is ready to be corrected. The offsets into the data string can be calculated from the formulas:

```
location = log(locator)
offset   = (d-1) - location
```

Where *d* is the number of bytes in the string. Then use xor to correct the data:

$$original[offset_i] = data[offset_i] \text{ xor } error\ value[i]$$

7. Implementation Techniques

7.1 Solving Polynomials

A simple technique for solving finite field polynomials is to repeatedly calculate the polynomial using all possible values and creating a list of the ones that produce zeroes.

7.1.1 Solving a Quadratic Equation

The following algorithm can be used to solve a quadratic locator polynomial. Note that double roots ($a = 0$) or zero root ($b = 0$) indicate that the polynomial does not represent a valid two error case. Given two coefficients, *a* and *b*, find the two roots R and S.

```
     X^2 + a X + b = 0                          (a ≠ 0) (b ≠ 0)
     c = root of X^2 + X + b/a^2                (use table)
     R = ac
     S = R+a
```

7.1.2 Solving a Qubic Equation

The following algorithm can be used to solve a cubic locator polynomial. Note that double root, triple root, or root = 0 indicate that the polynomial does not represent a valid three error case. Given three coefficients, *a*, *b*, and *c*, find the three roots R, S, and T.

```
     X^3 + a X^2 + b X + c = 0
     if(c == 0){                            (root == 0)
          d = a
          goto xx}
     e=a^2 + b
     f = ab+c
     if(f == 0){                            (double or triple root)
          R = a
          S = b^(1/2)                       (use table)
          T = S
          stop}
     g = root of X^2 + X + e^3/f^2          (use table)
     if(g == 0){ g = 1 }                    (fix in table)
     h = (fg)^(1/3)                         (use table)
     d = h + e/h
xx:  R = d + a
     if(R == 0){ e = b }
     else      { e = c/R }
     f = root of X^2 + X + e/d^2            (use table)
     S = df
     T = S + d
     stop
```

7.2 Software Based Math

Software based math usually makes use of logarithm tables. See sections 5.2 and 5.4.

7.3 Hardware Based Math

Multiplication can be implemented in a straightforward method. Division is usually implemented by inverting the divisor (not so easy) and then multiplying.

7.3.1 GCD Finite Field Inversion Algorithm

An algorithm similar to Euclid's GCD (greatest common divisor) algorithm can be used to implement inversion. The goal is to solve for *b* in the equation $a\ b \bmod m = 1$, where *a* and *b* are finite field numbers, and *m* is the polynomial defining the field. This problem can be restated in binary polynomial math instead of finite field math as $a\ b + c\ m = 1 = d$, where *a* and *m* are given, *d* is 1, and *b* and *c* are unknown. Given *a* and *m*, the algorithm determines *b*, *c*, and *d*, where *d* is the GCD. The fact *m* is a prime number, means that the GCD of *b* and *m* is 1, and allows the following algorithm to be used to find *b*:

```
b[-2]   = 0
b[-1]   = 1
f[-2]   = m
f[-1]   = a
i       = -1

i       = i+1
quot    = f[i-2]/f[i-1]                 ;quotient
f[i]    = f[i-2]-quot*f[i-1]            ;remainder
b[i]    = b[i-2]+quot*b[i-1]            ;reciprocal term
repeat above 4 lines until f[i]=0
reciprocal = b[i-1]
```

Using the 4-bit field based on 19, find the inverse of 0B, using the same technique:

```
1 = a b + c m
1 = 0B b + c 19     find GCD of 0B and 19

19/0B=03, 19-03*0B=19-1D=04=f[0]   :    19 = 03 * 0B + 04
0B/04=02, 0B-02*04=0B-08=03=f[1]   :    0B = 02 * 04 + 03
04/03=03, 04-03*03=04-05=01=f[2]   :    04 = 03 * 03 + 01
03/01=03, 03-03*01=03-03=00=f[3]   :    03 = 03 * 01 + 00

b[0] = 00+03*01 = 00+03 = 03
b[1] = 01+02*03 = 01+06 = 07
b[2] = 03+03*07 = 03+09 = 0A

b = 0A
```

7.3.2 Mapping to Sub-Field Inversion Algorithm

This algorithm relies on mapping a number into a field based on double sub-field to find the inverse. A lookup table is still needed, but the table size is the square root of the size of a standard lookup table. The algorithm operations are mapping into the field, calculating the two sub-field values, and mapping back. Although this is complicated, all these operations can be implemented in a single step, with less logic than a full lookup table, but with more propagation delays.