Reed Solomon Codes Explained

Author : Tony Hill
Date : 8th May 2013
Version : v1-0

INDEX

1 Introduction

Error identification and correction are crucial for reliable and efficient digital communication and data storage. Reed Solomon (RS) codewords are the most widely used method to provide digital error detection and correction capabilities for ADSL/VDSL lines, Digital Video Broadcast (DVB), CD and DVD players, and many other forms of digital communication and data storage.

Most of the articles on RS focus on number theory and many are quite difficult to understand, unless you have a strong background in maths or are studying this topic at college or university. Few provide practical, working examples of how RS codewords are generated, encoded and decoded and how error detection and correction is implemented. This paper does not delve too deeply into number theory (there has to be some) nor does it attempt to prove complex mathematical equations, there is plenty of other material out there that does precisely that. Several working examples are provided throughout the text and in the appendices.

2 Overview

2.1 Example of RS Application

ADSL is an example of the application of RS for error detection and correction. Ignoring for now the ATM encapsulation and SAR function, the sending device extracts user (payload) data from IP packets and places it inside ADSL frames. A polynomial uses the data bits in the frame to generate a Forward Error Correction (FEC) checksum, which is appended to the frame forming a RS codeword. The codewords are converted to symbols and transmitted as tone frequencies over a copper pair to the far-end receiving device.

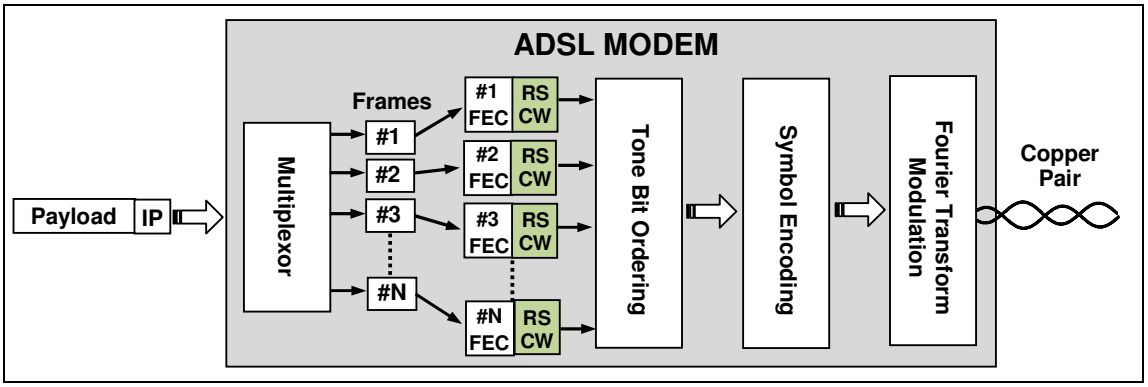

Figure 2-1: RS Codewords in ADSL

Upon receiving the frequency tones, the far-end device converts the symbols back into codewords and divides them by the same polynomial that the sending device used to generate them. No errors have been introduced during transmission if the division does not produce a remainder. If there is a remainder, the receiving device knows there is an error and uses the value of the remainder to identify and correct the specific symbols in which the errors occurred.

The length of the checksum determines the number of errors that can be detected and corrected. Error detection and correction incur a processing overhead so a balance has to be struck between the amount of FEC and user data information that is transmitted. An ADSL1 implementation typically uses 256 x byte codewords that include 224 x data bytes and 32 x FEC bytes.

2.2 Finite Fields

A finite field contains a set of pre-determined elements. The dimensioning and mathematical properties of finite fields make them ideal for symbol encoding and bit error detection applications - the larger the field, the longer the RS codeword, the higher the information transfer rate.

A finite field is constructed using a prime number base. The use of prime numbers is essential to ensure that any single element in the field produces a unique value when it is added to, or multiplied by, any other element in the field. This is not the case if a non-prime base is used.

GF (2), GF (3), GF (5), GF (7) etc. are all examples of valid prime number integer fields. However, so is 2^n, where 2 is the prime base and **n** an exponent that determines the number of elements within the field. For example, GF (2^3) = GF (8) is a base-2 field with 8 elements {0, 1, 2, 3, 4, 5, 6, 7}. GF (2^8) = GF (256) is a base-2 field with 256 elements {0, 1, 2, 3…….255} etc.

It clearly makes sense to use 2 as the prime base for binary computation. The example later in this paper generates a RS codeword based on a GF (2^3) field. However, polynomials are used rather than integers when populating a 2^n finite field, which is explained later.

2.2.1 Finite Field Example

This section shows the construction and properties of a GF (7) integer field to demonstrate finite field concepts. The GF (7) set contains the seven elements {0, 1, 2, 3, 4, 5, 6}. It is a modulus-7 field and, therefore, 6 is the maximum element value. The primitive (root) of the field is 3 i.e. all of the values in the field are derived as powers of 3, as follows:

$3^0 = 1$ [$= 3^6 =$ mod 1]
$3^1 = 3$ [$= 3^7 =$ mod 3]
$3^2 = 2$ [3 x 3 = 9 ÷ 7 = 1 mod 2]
$3^3 = 6$ [3 x 3 x 3 = 27 ÷ 7 = 3 mod 6]
$3^4 = 4$ [3 x 3 x 3 x 3 = 81 ÷ 7 = 11 mod 4]
$3^5 = 5$ [3 x 3 x 3 x 3 x 3 = 243 ÷ 7 = 34 mod 5]
$3^6 = 1$ [3 x 3 x 3 x 3 x 3 x 3 = 729 ÷ 7 = 104 mod 1]
$3^7 = 3$ [3 x 3 x 3 x 3 x 3 x 3 x 3 = 2,187 ÷ 7 = 312 mod 3]
3^n = etc.

Note that $3^6 = 3^0 = 1$ and $3^7 = 3^1 = 3$ and $3^8 = 3^2 = 2$ etc. The values of the field repeat ad infinitum for any powers of 3. Any values obtained as a result of addition, subtraction, multiplication and division are always equal to the values 3^0 to 3^5 no matter what the result of the arithmetic. This cyclic behaviour is well suited to the way that microprocessors use registers to perform bit-shift arithmetic.

The GF (7) addition table below contains the results of all of the possible addition and subtraction operations for the elements in the field. Note the symmetry of the table. There are no negative or duplicate numbers in any of the rows or columns.

GF (7) - {0, 1, 2, 3, 4, 5, 6}

GF (7) - ADDITION TABLE

+	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

Table 2-1: GF (7) Addition Table

Numbers with sums less than 7 are added in the usual way. For example, 1 + 5 = 6, 2 + 4 = 6, 3 + 3 = 6, 4 + 2 = 6, 5 +1 = 6. Addition obeys the commutative law i.e. 1 + 5 = 5 + 1, 2 + 4 = 4 + 2, 3 + 3 = 3 + 3 etc. For numbers with sums greater than or equal to 7 ignore the integer result and take the modulus-7 remainder i.e. 1 + 6 = **0**, 2 + 6 = **1**, 3 + 6 = **2**, 5 + 3 = **1** etc.

Subtraction is also performed as an addition using the additive inverse of elements. The additive inverse of an element is the value which when added to that element results in 0 [**a** + (**-a**) = 0]:

Value		Inverse		Result
1	+	**6**	=	**0**
2	+	**5**	=	**0**
3	+	**4**	=	**0**
4	+	**3**	=	**0**
5	+	**2**	=	**0**
6	+	**1**	=	**0**

Table 2-2: GF (7) Additive Inverse

For example, to subtract 1 - 6 add the first digit 1 to the additive inverse of 6, which is 1 i.e. 1 +1 = **2**. Proof: 6 + 2 = 1 so 2 = 1 - 6.

To subtract 3 - 5 add the first digit 3 to the additive inverse of 5, which is 2 i.e. 3 + 2 = 5. Proof: 5 + 5 = 3 so 5 = 3 - 5.

The GF (7) multiplication table below contains the results of all of the possible multiplication and division operations for the field. As with the addition table, there are no negative or duplicate numbers in any of the rows or columns.

GF (7) - {0, 1, 2, 3, 4, 5, 6}

GF (7) - MULTIPLICATION TABLE

+	0	1	2	3	4	5	6
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**1**	**2**	**3**	**4**	**5**	**6**
2	**0**	**2**	**4**	**6**	**1**	**3**	**5**
3	**0**	**3**	**6**	**2**	**5**	**1**	**4**
4	**0**	**4**	**1**	**5**	**2**	**6**	**3**
5	**0**	**5**	**3**	**1**	**6**	**4**	**2**
6	**0**	**6**	**5**	**4**	**3**	**2**	**1**

Table 2-3: GF (7) Multiplication Table

Multiplication of values within the field follows the same modulus rules for addition. For numbers with products greater than or equal to 7 ignore the integer result and take the modulus-7 remainder i.e. 1 x 5 = **5**, 2 x 5 = **3**, 3 x 5 = **1**, 4 x 5 = **6**, 5 x 5 = **4**, 5 x 6 = **2** etc. Multiplication obeys the commutative law i.e. 1 x 5 = 5 x 1, 2 x 5 = 5 x 2 etc.

It is also possible to multiply numbers by adding the powers of the primitive 3, for example, 2 x 5 = 3^2 x 3^5 = 3^7 = 3^1 = 3. Another example: 6 x 5 = 3^3 x 3^5 = 3^8 = 3^2 = 2. Recall that the powers repeat so 3^6 = 3^0, 3^7 = 3^1, 3^8 = 3^2 etc.

Division is also performed as a multiplication using the multiplicative inverse of elements. The multiplicative inverse of an element is the value which when multiplied by that element results in 1 [**b** x ($\mathbf{b}^{-1}$) = 1]:

Value		Inverse		Result
$\mathbf{3^0}$	x	$\mathbf{3^6}$	=	**1**
$\mathbf{3^1}$	x	$\mathbf{3^5}$	=	**1**
$\mathbf{3^2}$	x	$\mathbf{3^4}$	=	**1**

3^3	x	3^3	=	1
3^4	x	3^2	=	1
3^5	x	3^1	=	1
3^6	x	3^0	=	1

Table 2-4: GF (7) Multiplicative Inverse

From above, it can be seen that the inverse of an element 3^i is 3^{N-i} where **N** = 6, the highest value in the field.

For example, to divide 4 ÷ 3 multiply 4 by the inverse of 3^1, which is 3^5 i.e. 3^4 x 3^5 = 3^9 = 3^3 = 6. Proof: 4 ÷ 3 = 6, 6 x 3 = 4.

To divide 3 ÷ 5, multiply 3 by the inverse of 3^5, which is 3^1 i.e. 3^1 x 3^1 = 3^2 = 2. Proof: 3 ÷ 5 = 2, 2 x 5 = 3.

It is also possible to divide numbers by multiplying and subtracting the powers of the primitive 3, for example, 4 ÷ 3 = 3^4 x 3^{-1} = 3^3 = 6.

This section has confirmed some important properties of finite fields, in particular:

- A finite field must be based on a prime number to ensure that each row and column of its addition and multiplication tables contains unique values;
- The primitive (root) of the field is used to derive all of the values in the field;
- A finite field is indeed finite, but the field repeats an infinite number of times (3^0 = 3^6 = 3^{12} etc.);
- The modulus of the field determines the values of its elements;
- Only addition and multiplication are required - subtraction and division are performed using the additive and multiplicative inverses respectively.

2.2.2 GF (2) Fields

A base-2 finite field, also known as a Galois Field (GF) after the 19th century French mathematician who developed the concept, is denoted using the form GF (2^n), where **n** is the number of elements in the field. For example, a GF (2^3) is a field with 8 elements, also expressed as GF (8). A GF (2^8), or GF (256), contains 256 elements.

The most basic field is a GF (2^1), which contains two elements {0, 1}. This simple field allows us to demonstrate the binary mathematical rules for all GF (2^n) fields.

ADD MODULO-2 [XOR]

+	0	1
0	0	1
1	1	0

MULTIPLY MODULO-2 [AND]

x	0	1
0	0	0
1	0	1

Table 2-5: GF (2) Addition & Multiplication Tables

The following key rules are stipulated for the base-2 field:

- There are only two mathematical operations; addition and multiplication. Subtraction is achieved using an element's additive inverse and division using an element's multiplicative inverse;

- The additive inverse of an element **a** is its inverse **-a** such that **a** + (**-a**) = 0;
- The multiplicative inverse of every non-zero element **a** is its inverse $\mathbf{a}^{-1}$ such that **a** x $\mathbf{a}^{-1}$ = 1.
- There must be a 0 and a 1 element;
- Every mathematical operation results in the value of another element in the field;
- The associative law applies: a + (b + c) = (a + b) + c: a x (b x c) = (a x b) x c;
- The commutative law applies: a + b = b + a: a x b = b x a;
- The distributive law applies: a x (b + c) = (a x b) + (a x c).

The following information is derived from Table 2-5 and the rules above:

- `0+0=0, 0+1=1, 1+0=1, 1+1=0`. Therefore, addition is an XOR operation;
- `0x0=0, 0x1=0, 1x0=0, 1x1=1`. Therefore, multiplication is an AND operation;
- Since `1+1=0` and `1-1=0` then `1 = -1`. Therefore, adding is the same as subtracting;
- Since `1+1=0` and `1+1=2` then `2 = 0`.

2.3 Polynomials

2.3.1 Polynomial Roots

A polynomial is an expression that describes a non-linear set of co-ordinates. For example, the polynomial $y = x^2 - 1$ is used to describe the parabola in Figure 2-2 below. The graph is described by substituting **x** with a series of values and plotting the points **x, y**.

Val X		Val Y
-5	=	24
-4	=	15
-3	=	8
-2	=	3
-1	=	0
0	=	-1
1	=	0
2	=	3
3	=	8
4	=	15
5	=	24

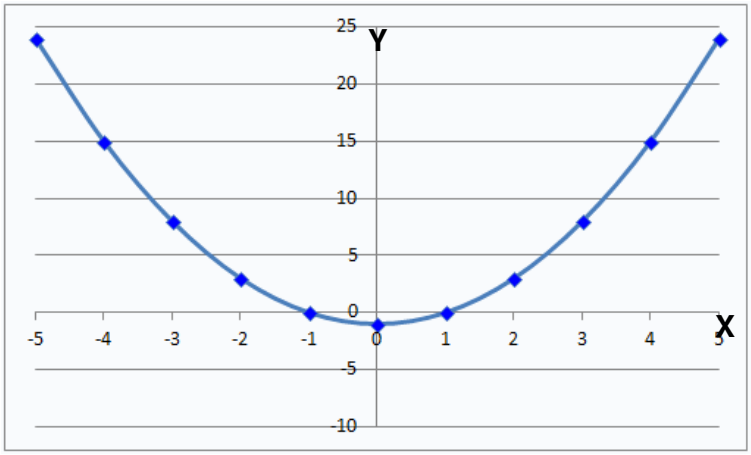

Figure 2-2: Parabola for the Polynomial $x^2 - 1$

The polynomial $x^2 - 1$ is reducible i.e. it can be factored into two lower order polynomials $x^2 - 1 = (x + 1)(x - 1)$. The roots of the polynomial are real integers i.e. x = 1 and x = -1 respectively, which is where the parabola crosses the x axis. The roots are obtained by equating the polynomial to 0, factoring it and determining the values for x:

$x^2 - 1 = (x + 1)(x - 1) = 0$, therefore the roots are **x = -1** and **x = 1** respectively.

Contrast this with the polynomial $x^2 + 1$ described in Figure 2-3 below:

Val X		Val Y
-5	=	26
-4	=	17
-3	=	10
-2	=	5
-1	=	2
0	=	1
1	=	2
2	=	5
3	=	10
4	=	17
5	=	26

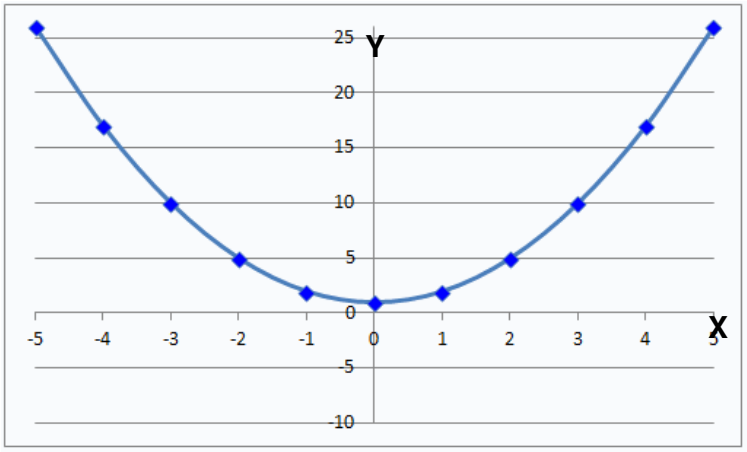

Figure 2-3: Parabola for the Polynomial $x^2 + 1$

The polynomial $x^2 + 1$ has no roots i.e. it is irreducible. In other words, it is the polynomial equivalent of a prime number.

Irreducible (prime) polynomials rather than prime integers are used to generate the values in GF (2^n) finite fields. The irreducible polynomial $x^3 + x + 1$ is used to generate the example GF (2^3) field later in this paper.

2.3.2 Generating a GF (2^n)

GF (2^n) fields are generated using an irreducible polynomial rather than an integer. Every element of a GF (2^n) field is a lower order polynomial of the field generating polynomial. Each low order polynomial element of the field is of the form:

P(x) = $a_0 x^0 + a_1 x^1 + a_2 x^2 + \ldots\ldots a_{m-1} x^{m-1}$ where the coefficients a_0 to a_{m-1} are binary or decimal values.

An irreducible 3rd degree (cubic) polynomial generates an 8 x 3-bit symbol field; an irreducible 4th degree polynomial generates 16 x 4-bit symbol field; an irreducible 8th degree polynomial generates 256 x 8-bit symbol field etc.

Either of the two irreducible 3rd degree polynomials **$P(a) = a^3 + a + 1$** or **$P(a) = a^3 + a^2 + 1$** can be used to generate an 8 x 3-bit symbol field. The first one is used to generate the GF for the examples in this paper.

As discussed in Section 2.3.1, the prime polynomial is irreducible so it does not have a real integer root. However, from Section 2.2.2 we know that for a GF (2^n) field 1 + 1 = 0 and 1 - 1 = 0 so we can assign a value to **a** and make it a root to create a finite field with some interesting properties. If **a** = 2 is the primitive root of **$P(a) = a^3 + a + 1$** and **$a^3 + a + 1 = 0$** then **$a^3 = a + 1$** (because 1 + 1 = 0 and 1 = -1). The first elements of the field are **0** and **a**. Populate the rest of the field by multiplying the previous non-zero element by **a** and substituting **a^3** for **a + 1**. Note that the positions of the bits in the 3-bit symbols match the positions of **a^2**, **a^1** and **a^0** in the table.

Decimal	Binary	Element Polynomial 2^2 2^1 2^0	a	Derivation
0	000		0	First element = 0
1	001	1	a^0	Second element = 1
2	010	a	a^1	a x 1 = a
4	100	a^2	a^2	a x a = a^2
3	011	a + 1	a^3	a x a^2 = a^3 = a + 1
6	110	a^2 + a	a^4	a x (a + 1) = a^2 + a
7	111	a^2 + a + 1	a^5	a x (a^2 + a) = a^3 + a^2 = a + 1 + a^2
5	101	a^2 + 1	a^6	a x (a^2 + a + 1) = a^3 + a^2 + a = a + 1 + a^2 + a = a^2 + ~~2a~~ + 1

Table 2-6: Elements of Polynomial $P(a) = a^3 + a + 1$

Recall from Section 2.2.1 that a finite field repeats an infinite number of times. To verify this, prove that $a^7 = a^0 = 1$:

$a^7 = a \times a^6 = a \times (a^2 + 1) = a^3 + a = a + a + 1 = $ ~~2a~~ $+ 1 = 1$ ✓

Note: 1 + 1 = 0, therefore 2 = 0, so 2a = 0.

Table 2-7 below summarises the field and shows the position of the element polynomials and the symbol values in binary and decimal.

0	a^0	a^1	a^2	a^3	a^4	a^5	a^6
0	$x^0 = x^7$	x	x^2	x + 1	x^2 + x	x^2 + x + 1	x^2 + 1
000	001	010	100	011	110	111	101
0	1	2	4	3	6	7	5

Table 2-7: GF (8) Table for Polynomial $P(x) = x^3 + x + 1$

The irreducible polynomial (**$x^3 + x + 1$**) that generates the field defines the modulus of the field for all arithmetic operations. The value of the modulus is $1011_2 = 11_{10}$:

$P(x) = 1x^3 + 0x^2 + 1x + 1 = 1011_2 = 11_{10}$

2.3.3 GF (2^n) Addition & Multiplication Tables

The addition and multiplication tables for the GF (2^3) field are shown below. Note that bit-wise XOR is used to compute the values of the addition table.

Addition example 1: **3 + 5 = 011 XOR 101 = 110 = 6**.

Addition example 2: **1 + 6 = 001 XOR 110 = 111 = 7**.

Addition and subtraction yield exactly the same result because 1 + 1 = 0 and 1 - 1 = 0.

Subtraction example: **1 - 6 = 001 XOR 110 = 111 = 7**.

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

Table 2-8: GF (8) Addition Table

Multiplication is performed using conventional binary arithmetic combined with XOR. As ever, the result of the multiplication is determined by the modulus of the field. If the result is more than the modulus, it is XORed with the modulus to obtain the remainder.

Multiplication example 1: **3 x 5 = 011 x 101 = 100 = 4**:

		0	1	1	
	x	1	0	1	
		0	1	1	
	0	0	0		Shift left
0	1	1			Shift left
	1	1	1	1	XOR Result
	1	0	1	1	XOR Modulus 1011_2 (11_{10})
		1	0	0	Result = 4_{10}

Verify by adding the powers of the **a** in the field: 3 x 5 = $\mathbf{a^3 \times a^6 = a^9 = a^{2\ (mod\ 7)} = 4}$

Multiplication example 2: **6 x 7 = 110 x 111 = 100 = 4**:

		1	1	0	
	x	1	1	1	
		1	1	0	
	1	1	0		Shift left
1	1	0			Shift left
1	0	0	1	0	XOR Result
1	0	1	1		XOR Modulus 1011_2 (11_{10})
		1	0	0	Result = 4_{10}

Verify by adding the powers of the **a** in the field: 6 x 7 = $\mathbf{a^4 \times a^5 = a^9 = a^{2\ (mod\ 7)} = 4}$

x	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	3	1	7	5
3	0	3	6	5	7	4	1	2
4	0	4	3	7	6	2	5	1
5	0	5	1	4	2	7	3	6
6	0	6	7	1	5	3	2	4
7	0	7	5	2	1	6	4	3

Table 2-9: GF (8) Multiplication Table

Division is performed using conventional binary arithmetic with XOR until the remainder is obtained. Note that for polynomial division the principal objective is to determine the remainder. The reason for this is explained further when generating the codeword in the example later in the paper.

Division example 1: **7 ÷ 3 = 111 ÷ 011 = 2 remainder 1**:

$2^1\ 2^0\ 2^2\ 2^1\ 2^0$

`      1  0` = 2 remainder 1

`1  1 | 1  1  1`

`      1  1` XOR

`      0  0  1` Remainder 1

Division example 2: **5 ÷ 3 = 101 ÷ 011 = 011 = 3 remainder 0**:

$2^1\ 2^0\ 2^2\ 2^1\ 2^0$

`      1  1` = 3 remainder 0

`1  1 | 1  0  1`

`      1  1` XOR

`         1  1` Remainder 0

Verify by subtracting the powers of **a** in the field: 5 ÷ 3 = $\mathbf{a^6 \times a^{-3} = a^3 = 3}$

2.3.4 Polynomial Arithmetic

The use of polynomial arithmetic may seem a bit counter intuitive at first, particularly for microprocessors that manipulate bits rather than algebraic equations. The polynomials are really only representations of bit patterns with the values of the exponents dictating the location of the bits and the coefficients specifying the values at those locations. The polynomials allow us to visualise the arithmetic and to apply mathematical rules to the operations that are performed.

Addition of the two finite filed elements $\mathbf{a^5}$ and $\mathbf{a^6}$ can be achieved by adding the polynomials or by adding the binary values using XOR:

$\mathbf{x^2 + x + 1}$	$\mathbf{a^5}$	**111**	**7**
$\mathbf{+\ x^2 + \quad + 1}$	$\mathbf{a^6}$	**101**	**5**
$\mathbf{\sout{2x^2} + x + 0}$	$\mathbf{a^1}$	**010**	**2**

Note that for a GF (2^n) field since 1 + 1 = 0, then 2 = 0 so $2x^2 = 0$. Therefore, the answer is **010**. Examining the addition table confirms that 7 + 5 = 2.

Ignoring polynomials completely it is possible to add just the coefficients of **x**.

	1	**1**	**1**
XOR	**1**	**0**	**1**
	0	**1**	**0**

Multiplying two polynomials:

	$x^2 + x + 1$	a^5	111	7	
x	$x^2 + x + 0$	a^4	110	6	
	$0 + 0 + 0$		000	0	
	$x^3 + x^2 + x + 0$		1110	14	shift left
	$x^4 + x^3 + x^2 + 0 + 0$		11100	28	shift left
	$x^4 + \cancel{2x^3} + \cancel{2x^2} + x + 0$		10010	18	> modulus

Since 1 + 1 = 0, then 2 = 0 so $2x^3 = 0$ and $2x^2 = 0$. Therefore, the answer is $x^4 + x$. But this is a higher degree polynomial than the field generating polynomial so exceeds the modulus of the field. To calculate the final result is it necessary to divide $x^4 + x$ by $x^3 + x + 1$ (the modulus) to obtain the remainder:

	$x + 0$	= **x** remainder x^2 = **4**
$x^3 + x + 1$	$x^4 + 0x^3 + 0x^2 + x + 0$	
	$x^4 \quad + 1x^2 + x$	XOR
	$x^2 \quad + 0$	
	$0 \quad + 0$	XOR
	x^2	Remainder x^2 = 4

Using coefficient rather than polynomial long division:

	1 0	= **2** remainder **100** = **4**
1 0 1 1	**1 0 0 1 0**	
	1 0 1 1	XOR
	1 0 0	
	0 0 0	XOR
	1 0 0	Remainder 100 = 4

The whole operation could have been performed using binary, as follows:

	1 1 1	a^5	
x	**1 1 0**	a^4	
	0 0 0		
	1 1 1 0		shift left
1 1 1 0 0			shift left
1 0 0 1 0			XOR result
1 0 1 1 0			shift left modulus & XOR
	1 0 0	a^2	$a^5 + a^4 = a^9 = a^2$

Division is also performed as a multiplication using the multiplicative inverse of elements. The multiplicative inverse of an element is the value which when multiplied by that element results in 1 [**a** x ($\mathbf{a}^{-1}$) = 1]. For example,

Dec	a^n	Inv	Dec	Result
0	0	0	0	0 x 0 = 0
1	a^0	a^7	1	1 x 1 = 1
2	a^1	a^6	5	2 x 5 =1
3	a^3	a^4	6	3 x 6 = 1
4	a^2	a^5	7	4 x 7 = 1
5	a^6	a^1	2	5 x 2 = 1
6	a^4	a^3	3	6 x 3 = 1
7	a^5	a^2	4	7 x 4 = 1
1	a^7	a^0	1	1 x 1 = 1

Table 2-10: GF (2^3) Multiplicative Inverse

From above, it can be seen that the inverse of an element $\mathbf{a}^{\mathbf{i}}$ is $\mathbf{a}^{\mathbf{N-i}}$ where **N** = 7, the highest value in the field.

To divide $\mathbf{a}^4 \div \mathbf{a}^3$ multiply $\mathbf{a}^4$ by the inverse of $\mathbf{a}^3$, which is $\mathbf{a}^4$ i.e. $\mathbf{a}^4$ x $\mathbf{a}^4$ = $\mathbf{a}^{8\ (\mathbf{mod}\ 7)}$ = $\mathbf{a}^1$ = 2. Proof: $\mathbf{a}^4$ = 6, $\mathbf{a}^3$ = 3, 6 ÷ 3 = 2.

2.3.5 Exponents and Coefficients

In a Galois Field, the exponents of **x** identify the precise position of symbols in the field and the coefficients determine specific symbol values. The exponents and coefficients are a bit like co-ordinates for the field. A string of 15 x bits split into 5 x 3-bit symbols 7, 7, 4, 3 and 2 would be represented as follows:

$a_5 x^4$	$a_5 x^3$	$a_2 x^2$	$a_3 x^1$	$a_0 x^0$
111	111	100	011	010
7	7	4	3	2

Table 2-11: Exponents & Coefficients

Therefore, it is relatively straightforward to take any string of bits, split them into the 3-bit symbols and generate RS codewords that result in a specific remainder when divided by a certain value.

If the receiving device obtains a different remainder than the sending device it not only knows that an error is present it can use the value of the remainder to identify specific symbols that have been affected.

2.4 RS Codewords

The maximum RS codeword length **n** is given by the following formula where **s** is the symbol size in bytes. In other words, if the symbol size is 8-bits (one byte) the highest codeword byte is 2^8 - 1 = 255.

$$\mathbf{n = 2^s - 1}$$

RS codewords are denoted with the expression RS **[n, k]** where **n** = number of codeword bytes and **k** = number of data bytes. The number of Forward Error Correction (FEC) bytes is **n - k**. A complete RS codeword specification for a 256 symbol codeword that uses 8-bits per symbol with parity symbols would be:

RS [**n** = 255, **k** = 233] **s** = 8, 2**t** = 32

Where **s** = symbol size and 2**t** = the number of parity symbols. 2 x parity symbols are required to correct a single symbol error. Therefore, 2**t** = 32 would correct a single symbol (**t** = 1) errors.

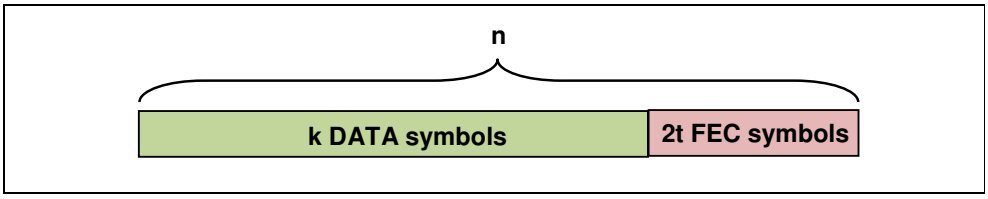

Figure 2-4: RS Codeword

For simplicity, the example later in this paper uses 3-bit symbols so the codeword is **n** = 2^3 - 1 = 7 bytes long. The full specification of the codeword is:

RS [7, 5] **s** = 3, **t** = 2

5 symbols are data and 2 symbols are used for error correction. As there are 2 x FEC symbols it is only possible to identify and correct one errored symbol for each codeword.

2.5 RS Polynomials

Three polynomials are used to generate RS codewords. The first is used to generate the finite field (Galois Field), the second is a generic polynomial that is used to generate the encoding polynomial, and the third is the encoding polynomial itself. The encoding polynomial generates the actual RS codewords.

2.5.1 GF Polynomial

The following polynomial is used to generate the finite field (Galois Field) for the example in this paper. It is an "irreducible" cubic (x^3) polynomial that cannot be factored (reduced) any further. It is effectively the polynomial equivalent of a prime number.

$$P(x) = x^3 + x + 1$$

2.5.2 Generator Polynomial

The following "generator" polynomial is a generic polynomial that is used to generate the encoding polynomial. The values a^1, a^2, a^3 etc. are the values of **a** in the finite field and the value **2t** determines how many FEC symbols are generated.

$$G(x) = (x - a^1)(x - a^2)(x - a^3)....(x - a^{2t})$$

2.5.3 Encoding Polynomial

The following polynomial is the encoding polynomial. It uses factors of the generator polynomial to generate the RS codewords. In our example, the codeword specification is RS [7, 5] s = 3, t = **2**. The t = **2** means that we use 2 x FEC symbols so only the factors $(x - a^1)(x - a^2)$ from the generator polynomial are used. These 2 FEC symbols allow us to identify and correct 1 errored symbol in each codeword.

$$G(x) = (x - a^1)(x - a^2)$$

If the codeword's specification were RS [7, 3] s = 3, t = **4** we would use the 4 values $(x - a^1)(x - a^2)(x - a^3)(x - a^4)$ from the generator polynomial to create a codeword that consists of 3 x data symbols and 4 x FEC symbols allowing us to identify and correct 2 x errored symbols.

As described in the previous sections, addition and subtraction are the same for a GF (2^n) field so **$(x - a^1) = (x + a^1)$** and **$(x - a^2) = (x + a^2)$**, therefore, the encoding polynomial is calculated as follows:

$$G(x) = (x + a^1)(x + a^2)$$

$$G(x) = (x + 2)(x + 4) \qquad a1 = 2,\ a2 = 4 \text{ in the } x^3 + x + 1 \text{ GF } (2^3)$$

$$G(x) = x^2 + 6x + 3 \qquad \text{mod} = 11 \text{ for } x^3 + x + 1 \text{ GF } (2^3)\ [8 \bmod 11 = 3]$$

Note that **$(x + 2)(x + 4) = x^2 + 6x + 8$**. However, the polynomial used to generate the finite field in this example is **$x^3 + x + 1$**, which is modulo $1011_2 = 11_{10}$ so it is necessary to convert the 8 to a 3.

$$\begin{array}{rcl} 1\ 0\ 0\ 0 & = & 8 \\ \text{XOR}\ 1\ 0\ 1\ 1 & = & 11 \\ \hline 1\ 1 & = & 3 \end{array}$$

Therefore, the encoding polynomial used in this paper for generating RS codewords is:

$$G(x) = x^2 + 6x + 3$$

Which can also be expressed as **1 6 3**.

3 RS Codeword Example

3.1 Bit String

The bit string to encode as a RS codeword for transmission in this example is “001010011100101”:

$a_0 x^4$	$a_1 x^3$	$a_3 x^2$	$a_2 x^1$	$a_6 x^0$
001	010	011	100	101
1	2	3	4	5

Table 3-1: Bit String Message to Encode

The bit string message is represented using the following notation:

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

3.2 Codeword Generation

The codeword specification is RS [7, 5] **s** = 3, **t** = 2, which denotes 5 x data symbols and 2 x FEC symbols, total = 7 x symbols.

To accommodate the 2 x FEC symbols at the end of the 5 x symbol message it is necessary to multiply the bit string by x^2 to shift the message left two symbol places:

$$\begin{array}{rl} & 1x^4 + 2x^3 + 3x^2 + 4x + 5 \\ & x \qquad\qquad\qquad\qquad\quad x^2 \\ \hline M(x) = & 1x^6 + 2x^5 + 3x^4 + 4x^3 + 5x^2 + 0x + 0 \end{array}$$

Divide M(x) by G(x) to obtain the FEC symbols, where G(x) is the encoding polynomial $x^2 + 6x + 3$. Note that it is only necessary to use the coefficients because their position is implied by the exponents:

x^2	x^1	x^0	x^6	x^5	x^4	x^3	x^2	x^1	x^0	
					1	4	5	0	1	Remainder **63**
1	6	3	1	2	3	4	5	0	0	
			1	6	3					x $1x^4$
				4	0	4				XOR value
				4	5	7				x $4x^3$
					5	3	5			XOR value
					5	3	4			x $5x^2$
							1	0	0	XOR value
							1	6	3	x $1x^0$
								6	3	Remainder

Note that addition is the same as subtraction for the GF (2^3). When using XOR in the division above, the following method is used. For example, 123 XOR 163:

	1			2			3		
	0	0	1	0	1	0	0	1	1
	1			6			3		
XOR	0	0	1	1	1	0	0	1	1
	0	0	0	1	0	0	0	0	0
	0			4			0		

Add the remainder 63 to G(x) to create the codeword C(x):

$$C(x) = 1x^6 + 2x^5 + 3x^4 + 4x^3 + 5x^2 + 6x + 3$$

C(x) = 001 010 011 100 101 110 011

This method is referred to as “systematic” encoding because the error correction information is embedded in and part of the codeword.

To verify that the codeword is valid, divide C(x) by the encoding polynomial G(x) to check that remainder = 0:

x^2	x^1	x^0	x^6	x^5	x^4	x^3	x^2	x^1	x^0	
					1	4	5	0	1	Remainder **0**
1	6	3	1	2	3	4	5	6	3	
			1	6	3					x $1x^4$
				4	0	4				XOR value
				4	5	7				x $4x^3$
					5	3	5			XOR value
					5	3	4			x $5x^2$
							1	6	3	XOR value

1	6	3	x $1x^0$
	0	0	Remainder

3.3 Error Detection

Upon receiving the RS codeword R(x) from the sender, the receiver divides the generating polynomial G(x) into it. A remainder indicates that an error was introduced during transmission. In this example, there is a double-bit error in the symbol $4x^3$ (100_2) transforming it into $7x^3$ (111_2).

The received RS codeword R(x) is as follows:

$$R(x) = 1x^6 + 2x^5 + 3x^4 + 7x^3 + 5x^2 + 6x + 3$$

The error that causes this transformation is the value 3 (011_2) because 100 XOR 011 = 111. But the receiver doesn't yet know this:

$$E(x) = 3x^3\ (011_2)$$

The receiver divides R(x) by G(x) and obtains a remainder and now knows that there is an error.

x^2	x^1	x^0	x^6	x^5	x^4	x^3	x^2	x^1	x^0	
					1	4	5	3	0	Remainder 3 3
1	6	3	1	2	3	7	5	6	3	
			1	6	3					x $1x^4$
				4	0	7				XOR value
				4	5	7				x $4x^3$
					5	0	5			XOR value
					5	3	4			x $5x^2$
						3	1	6		XOR value
						3	1	5		x $3x^1$
								3	3	XOR value

Dividing the error $0x^6 + 0x^5 + 0x^4 + 3x^3 + 0x^2 + 0x^1 + 0$ by G(X) returns exactly the same remainder proving that the error is in the symbol at position x^3:

x^2	x^1	x^0	x^6	x^5	x^4	x^3	x^2	x^1	x^0	
								3	1	Remainder 3 3
1	6	3	0	0	0	3	0	0	0	
						3	1	5		x $3x^1$
							1	5	0	XOR value
							1	6	3	x $1x^0$
								3	3	Remainder

This remainder is unique and is only generated for this specific bit error at this specific location in the bit string.

3.4 Remainder Lookup Table

Rather than obtaining a remainder, modifying the codeword R(x) and re-dividing until the remainder is 0, the receiver can identify very quickly the precise symbol that contains the error by locating the

remainder in a lookup table. The GF (2^3) field is cyclic and values repeat modulo 11_{10} so there are only 7 x 7 = 49 remainder values for all of the combinations of 7 x 3-bit codeword symbols.

The remainders are obtained by dividing every possible symbol coefficient value {1 to 7} of every exponent value {x^0 to x^6} by G(x) = $1x^2 + 6x + 3$ modulo $x^3 + x + 1$ (1011_2).

Table 3-2 below shows all of the values that are divided to create the table:

÷ 163	x^0	x^1	x^2	x^3	x^4	x^5	x^6
1	1	10	100	1000	10000	100000	1000000
2	2	20	200	2000	20000	200000	2000000
3	3	30	300	3000	30000	300000	3000000
4	4	40	400	4000	40000	400000	4000000
5	5	50	500	5000	50000	500000	5000000
6	6	60	600	6000	60000	600000	6000000
7	7	70	700	7000	70000	700000	7000000

Table 3-2: Creating the Remainder Lookup Table

As an example, taking column $\mathbf{x^3}$ row **7**, divide 7000 ÷ 163:

x^2	x^1	x^0	x^3	x^2	x^1	x^0	
					7	4	Remainder **77**
1	6	3	7	0	0	0	
			7	4	2		x $\mathbf{7x^1}$
				4	2	0	XOR value
				4	5	7	x $\mathbf{4x^0}$
					7	7	Remainder

Table 3-3 below shows the completed remainder lookup table:

÷ 163	x^0	x^1	x^2	x^3	x^4	x^5	x^6
1	01	10	63	11	73	72	62
2	02	20	76	22	56	54	74
3	03	30	15	33	25	26	16
4	04	40	57	44	17	13	53
5	05	50	34	55	64	61	31
6	06	60	21	66	41	47	27
7	07	70	42	77	32	35	45

Table 3-3: Completed Remainder Lookup Table

In Section 3.3, the receiver received the following codeword:

$$\mathbf{R(x) = 1x^6 + 2x^5 + 3x^4 + 7x^3 + 5x^2 + 6x + 3}$$

It divided R(x) by G(x) and obtained the remainder **3 3**. The receiver located this value in the lookup table and determined that the remainder is only possible for coefficient value 3 of exponent x^3. To correct the error, the receiver carries out the following XOR operation on the symbol in position x^3:

	1	**1**	**1**	Received symbol **= 7**
XOR	**0**	**1**	**1**	Coefficient value from table **= 3**
	1	**0**	**0**	Symbol changed to **= 4**

In section 3.3, the sender transmitted the following codeword:

$$R(x) = 1x^6 + 2x^5 + 3x^4 + 4x^3 + 5x^2 + 6x + 3 = C(x)$$

Therefore, the codeword is now valid.

3.5 Advanced Error Detection

Using a lookup table is fine if the codewords are composed of few symbols - but longer codewords require much larger lookup tables, which consumes memory. There are many alternative ways to detect and correct errors but this paper focuses on the use of the following mechanisms:

- Creation of a polynomial syndrome to quantify the error;
- Use of the Euclidian algorithm to define the error detection polynomial;
- Use of the Chien search algorithm to locate the coefficient of the errored symbol;
- Use of the Forney algorithm to correct the errored symbol.

3.5.1 Recap

In Section 3.2 we stipulated a five-symbol message polynomial M(x), as follows:

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

M(x) was multiplied by x^2 to accommodate the 2 x FEC symbols:

$$M(x)\ (x^2) = 1x^6 + 2x^5 + 3x^4 + 4x^3 + 5x^2 + 0x + 0$$

M(x) was divided by the generator polynomial G(x) = $1x^2 + 6x + 3$ to determine the remainder ($6x + 3$), which was appended to M(x) to form the systematic codeword C(x):

$$C(x) = 1x^6 + 2x^5 + 3x^4 + 4x^3 + 5x^2 + 6x + 3$$

Dividing C(x) by G(x) produces a remainder of 0. The codeword was transmitted to a receiving device and an error $E(x) = 3x^3$ was introduced causing the symbol $4x^3$ to change into $7x^3$.

The receiver received the R(x) codeword:

$$R(x) = 1x^6 + 2x^5 + 3x^4 + 7x^3 + 5x^2 + 6x + 3$$

The receiver divided R(x) by G(x) and obtained a remainder of 3x + 3 indicating the presence of an error. The following sections describe an alternative method the receiver uses to locate and correct the error.

3.5.2 Syndromes

Both the sender and receiver use the same polynomial and finite field to encode and decode messages so they know the roots of the generated codewords. When the receiver obtains a remainder, it creates a “syndrome” polynomial to define the error. It does this by substituting values of the finite field roots **a** into the R(x) codeword.

In this example, the roots a^0, a^1 and a^2 are substituted to create three syndromes S_0, S_1 and S_2. The number of values to substitute and the number of syndromes required is dependent on the dimension of the finite field. If the finite field contained 16 x 4-bit values defined by a 4th degree polynomial it would be necessary to create a fourth syndrome, and so on.

The S_0 syndrome is effectively a parity check determined by substituting each occurrence of a^0 in R(x) and XORing every element:

S_0	$a^0 = 1$	$1x^6$	$2x^5$	$3x^4$	$7x^3$	$5x^2$	$6x^1$	3	Parity
		$(a^0)^6$	$2(a^0)^5$	$3(a^0)^4$	$7(a^0)^3$	$5(a^0)^2$	$6(a^0)^1$	$3(a^0)^0$	
		001	010	011	111	101	110	011	**7 (111_2)**

Table 3-4: R(x) S_0 Syndrome

The S_1 syndrome is determined by substituting each occurrence of a^1 in R(x) and XORing every element. The final value is taken modulo 11_{10} which is the modulus of the cubic polynomial:

S_1	$a^1 = 2$	$1x^6$	$2x^5$	$3x^4$	$7x^3$	$5x^2$	$6x^1$	3	XOR
		$(a^1)^6$	$2(a^1)^5$	$3(a^1)^4$	$7(a^1)^3$	$5(a^1)^2$	$6(a^1)^1$	$3(a^1)^0$	
		64	64	48	56	20	12	3	**5**

Table 3-5: R(x) S_1 Syndrome

The S_2 syndrome is determined by substituting each occurrence of a^2 in R(x) and XORing every element. The final value is taken modulo 11_{10} which is the modulus of the cubic polynomial:

S_2	$a^2 = 4$	$1x^6$	$2x^5$	$3x^4$	$7x^3$	$5x^2$	$6x^1$	3	XOR
		$(a^2)^6$	$2(a^2)^5$	$3(a^2)^4$	$7(a^2)^3$	$5(a^2)^2$	$6(a^2)^1$	$3(a^2)^0$	
		4096	2048	768	448	80	24	3	**4**

Table 3-6: R(x) S_2 Syndrome

Note that the equivalent syndromes for the sender's (good) codeword C(x) all equal 0.

The syndrome polynomial S(x) for codeword R(x) is defined using the values of S_0, S_1 and S_2 as follows:

$$S(x) = 4x^2 + 5x + 3$$

This polynomial quantifies the error across the entire R(x) codeword. The accuracy of the S_0 and S_1 syndromes can verified as follows:

- R(x) divided by the (x + 2) root of G(x) returns remainder 5, which is the **x** coefficient of S(x);
- R(x) divided by the (x + 4) root of G(x) returns remainder 4, which is the x^2 coefficient of S(x).

The S(x) polynomial quantifies but does not locate the error within the R(x) codeword.

3.5.3 Euclidean Algorithm

The Euclidean algorithm is used to determine the Greatest Common Divisor (GCD) of two numbers. In the example below, the algorithm is used to find the GCD of the two numbers 1,431 and 864. The GCD of 27 is determined in five steps as follows:

1	1431	÷	864	=	1	remainder **567**
2	864	÷	**567**	=	1	remainder **297**
3	**567**	÷	**297**	=	1	remainder **270**
4	**297**	÷	**270**	=	1	remainder **27**
5	**270**	÷	**27**	=	10	remainder 0

Table 3-7: Example of Euclidean Algorithm

This algorithm is use in the next section to create the error locator polynomial. Please refer to Appendix 3 for an explanation of the extended Euclidean algorithm.

3.5.4 Error Locator Polynomial

The S(x) syndrome polynomial created in Section 3.5.2 is used to create the **Δ(x)** error locator polynomial. The Euclidian algorithm is used to divide S(x) into $\mathbf{x^3}$ (the degree of the finite field polynomial) to find a common divisor of both the finite field and S(x). The quotient of the division is used as a polynomial **Δ(x)** to locate the exponent of R(x) at which the error is located and the remainder is used as a polynomial **Ω(x)** to determine the magnitude of the error. It is essential that we obtain a remainder for this purpose.

Dividend					Divisor				Quotient		Remainder	
x^3	x^2	x^1	x^0		x^2	x^1	x^0		x^1	x^0	x^1	x^0
1	0	0	0	÷	4	5	7	=	**7**	**4**	**1**	**1**
	4	5	7	÷		1	1	=	4	1		6
		1	1	÷			6	=	7	3	0	0

Table 3-8: Determining Δ(x) and Ω(x)

The error locator polynomial is: **Δ(x) = 7x + 4**

The error magnitude polynomial is: **Ω(x) = 1x + 1**

Please note that for this example it isn't actually necessary to continue beyond the first line because the degree of the remainder drops below the degree of the generator polynomial G(x) (i.e. x^2+ 6x + 3) after the first division. However, the remaining operations are carried out to illustrate how the table is completed until the GCD (GCD = 6) is located. For a higher degree finite field with a higher degree generator polynomial G(x) it is necessary to continue dividing until the remainder drops below the degree of G(x); the quotient on that line of the table becomes **Δ(x)** *and the remainder* **Ω(x)**.

To prove that 6 is the GCD of both $\mathbf{1x^3 + 0x^2 + 0x + 0}$ and $\mathbf{4x^2 + 5x + 7}$:

$\mathbf{1x^3 + 0x^2 + 0x + 0 \div 6 = 3x^3 + 0x^2 + 0x + 0}$ **no remainder**

And:

$\mathbf{4x^2 + 5x + 7 \div 6 = 7x^2 + 4x + 2}$ **no remainder**

3.5.5 The Chien Search Algorithm

Having determined the error locator polynomial **Δ(x) = 7x + 4** it is necessary to substitute into it the **inverse** of $\mathbf{a^n x^n}$ [i.e. $\mathbf{a^{-n}x^{-n}}$ or $\mathbf{1/(a^n x^n)}$] for each value of **a** in the R(x) codeword. The exponent location that returns a value of 0 is the location of the error. The reason for this is that S(x) contains the error information of the entire R(x) codeword and **Δ(x)** is derived using S(x). The non-errored coefficients do not contribute to the error component and return a non-zero value when substituted but the coefficient that is responsible for the error cancels out and returns 0.

Recall that the modulus of the finite field GF (2^3) field = 1011_2. The substitution and results are shown in Table 3-9 below:

Δ(x)		a^n		Inverse a^n		Sum		Error = 0
Δx^{-6}	→	$7(a^{-6}) + 4$	=	$7(a^1) + 4$	=	5 + 4	=	1
Δx^{-5}	→	$7(a^{-5}) + 4$	=	$7(a^2) + 4$	=	1 + 4	=	5
Δx^{-4}	→	$7(a^{-4}) + 4$	=	$7(a^3) + 4$	=	2 + 4	=	6
Δx^{-3}	→	$7(a^{-3}) + 4$	=	$7(a^4) + 4$	=	4 + 4	=	0
Δx^{-2}	→	$7(a^{-2}) + 4$	=	$7(a^5) + 4$	=	3 + 4	=	7
Δx^{-1}	→	$7(a^{-1}) + 4$	=	$7(a^6) + 4$	=	6 + 4	=	2
Δx^{-0}	→	$7(a^{-0}) + 4$	=	$7(a^0) + 4$	=	7 + 4	=	3

Table: 3-9: Chien Search Algorithm

The error is located in the R(x) exponent position $\mathbf{x^3}$. We know this to be correct because the error introduced was $\mathbf{E(x) = 3x^3}$, which resulted in the following discrepancy between the C(x) and R(x) codewords:

$$\mathbf{C(x) = 1x^6 + 2x^5 + 3x^4 + 4x^3 + 5x^2 + 6x + 3}$$

$$\mathbf{R(x) = 1x^6 + 2x^5 + 3x^4 + 7x^3 + 5x^2 + 6x + 3}$$

3.5.6 Error Correction

Having located the error it is now possible to fix it using the $\mathbf{\Delta(x) = 7x + 4}$ and $\mathbf{\Omega(x) = 1x + 1}$ polynomials created in Section 3.5.4. This is accomplished using the Forney algorithm, which takes the form:

$$\mathbf{e_j = x_j \frac{\Omega(X^{-1})}{\Delta(X^{-1})}}$$

where **j** is the position of the error in the R(x) codeword.

The $\mathbf{\Delta(x) = 7x + 4}$ polynomial is modified such that the even powers of **x** are deleted and the polynomial is divided by **x** leaving $\mathbf{\Delta(x) = 7}$.

The position of the errored symbol $\mathbf{7x^3}$ is $\mathbf{a^3}$. Therefore, substituting the information for **a** into the formula:

$$\mathbf{e_j = a^3 \frac{(a^{-3} + 1)}{7}}$$

$$\mathbf{e_j = a^3 \frac{(a^4 + 1)}{7}}$$

$$\mathbf{e_j = 3 \frac{(6 + 1)}{7}}$$

$$\mathbf{e_j = 3}$$

The magnitude of the error is 3. Therefore, to correct the error: $\mathbf{7\ (111_2)}$ **XOR** $\mathbf{3\ (011_2) = 4\ (100_2)}$.

R(x) is now correct:

$$\mathbf{R(x) = 1x^6 + 2x^5 + 3x^4 + 4x^3 + 5x^2 + 6x + 3}$$

4 Appendix 1 - Worked Example 1

Codeword to transmit M(x):

$$\mathbf{M(x) = 2x^4 + 2x^3 + 2x^2 + 2x^1 + 2}$$

Generator polynomial G(x):

$$\mathbf{G(x) = 1x^2 + 6x + 3}$$

Multiply M(x) by $\mathbf{x^2}$ to accommodate 2 x FEC symbols:

$$\mathbf{M(x)\ (x^2) = 2x^6 + 2x^5 + 2x^4 + 2x^3 + 2x^2 + 0x + 0}$$

Divide M(x) by the generator polynomial G(x) = $\mathbf{1x^2 + 6x + 3}$ to create the systematic codeword C(x):

$$\mathbf{C(x) = 2x^6 + 2x^5 + 2x^4 + 2x^3 + 2x^2 + 2x + 2}$$

Introduce an error into symbol $\mathbf{2x^5}$ such that it becomes $\mathbf{1x^5}$:

$$\mathbf{E(x) = 3x^5\ (011_2\ XOR\ 010_2 = 001_2)}$$

The receiver R(x) codeword:

$$\mathbf{R(x) = 2x^6 + 1x^5 + 2x^4 + 2x^3 + 2x^2 + 2x + 2}$$

Create the syndrome polynomial S(x):

$\mathbf{S_0}$	$\mathbf{a^0 = 1}$	$\mathbf{2x^6}$	$\mathbf{1x^5}$	$\mathbf{2x^4}$	$\mathbf{2x^3}$	$\mathbf{2x^2}$	$\mathbf{2x^1}$	**2**	**Parity**
		$\mathbf{2(a^0)^6}$	$\mathbf{1(a^0)^5}$	$\mathbf{2(a^0)^4}$	$\mathbf{2(a^0)^3}$	$\mathbf{2(a^0)^2}$	$\mathbf{2(a^0)^1}$	$\mathbf{2(a^0)^0}$	
		010	001	010	010	010	010	010	**1 (001_2)**

Table A1-1: R(x) S_0 Syndrome

$\mathbf{S_1}$	$\mathbf{a^1 = 2}$	$\mathbf{2x^6}$	$\mathbf{1x^5}$	$\mathbf{2x^4}$	$\mathbf{2x^3}$	$\mathbf{2x^2}$	$\mathbf{2x^1}$	**2**	**XOR**
		$\mathbf{2(a^1)^6}$	$\mathbf{1(a^1)^5}$	$\mathbf{2(a^1)^4}$	$\mathbf{2(a^1)^3}$	$\mathbf{2(a^1)^2}$	$\mathbf{2(a^1)^1}$	$\mathbf{2(a^1)^0}$	
		128	32	32	16	8	4	2	**2**

Table A1-2: R(x) S_1 Syndrome

$\mathbf{S_2}$	$\mathbf{a^2 = 4}$	$\mathbf{2x^6}$	$\mathbf{1x^5}$	$\mathbf{2x^4}$	$\mathbf{2x^3}$	$\mathbf{2x^2}$	$\mathbf{2x^1}$	**2**	**XOR**
		$\mathbf{2(a^2)^6}$	$\mathbf{1(a^2)^5}$	$\mathbf{2(a^2)^4}$	$\mathbf{2(a^2)^3}$	$\mathbf{2(a^2)^2}$	$\mathbf{2(a^2)^1}$	$\mathbf{2(a^2)^0}$	
		8192	1024	512	128	32	8	2	**5**

Table A1-3: R(x) S_2 Syndrome

The syndrome polynomial S(x):

$$\mathbf{S(x) = 5x^2 + 2x + 1}$$

Run the Euclidian algorithm to determine the error locator and magnitude polynomials **Δ(x)** and **Ω(x)**:

Dividend					**Divisor**				**Quotient**		**Remainder**	
$\mathbf{x^3}$	$\mathbf{x^2}$	$\mathbf{x^1}$	$\mathbf{x^0}$		$\mathbf{x^2}$	$\mathbf{x^1}$	$\mathbf{x^0}$		$\mathbf{x^1}$	$\mathbf{x^0}$	$\mathbf{x^1}$	$\mathbf{x^0}$
1	0	0	0	÷	5	2	1	=	**2**	**3**	**4**	**3**
	5	2	1	÷		4	3	=	6	2		7
		4	3	÷			7	=	6	7	0	0

Table A1-4: Error Locator and Magnitude Polynomials

$$\mathbf{\Delta(x) = 2x + 3}$$

$$\mathbf{\Omega(x) = 4x + 3}$$

Run the Chien search algorithm using **Δ(x) = 2x + 3**:

Δ(x)		$\mathbf{a^n}$		**Inverse a^n**		**Sum**		**Error = 0**

Δx^{-6}	→	$2(a^{-6}) + 3$	=	$2(a^{1}) + 3$	=	4 + 3	=	7
Δx^{-5}	→	$2(a^{-5}) + 3$	=	$2(a^{2}) + 3$	=	3 + 3	=	0
Δx^{-4}	→	$2(a^{-4}) + 3$	=	$2(a^{3}) + 3$	=	6 + 3	=	5
Δx^{-3}	→	$2(a^{-3}) + 3$	=	$2(a^{4}) + 3$	=	7 + 3	=	4
Δx^{-2}	→	$2(a^{-2}) + 3$	=	$2(a^{5}) + 3$	=	5 + 3	=	6
Δx^{-1}	→	$2(a^{-1}) + 3$	=	$2(a^{6}) + 3$	=	1 + 3	=	2
Δx^{-0}	→	$2(a^{-0}) + 3$	=	$2(a^{0}) + 3$	=	1 + 3	=	4

Table: A1-5: Chien Search Algorithm

Correct the error at $\mathbf{a^5}$ using $\mathbf{\Omega(x) = 4x + 3}$. $\mathbf{\Delta(x) = 2x + 3}$ becomes $\mathbf{\Delta(x) = 2}$:

$$e_j = a^5 \frac{(4a^{-5} + 3)}{2}$$

$$e_j = a^5 \frac{(4a^{2} + 3)}{2}$$

$$e_j = 7 \frac{(6 + 3)}{2}$$

$$e_j = 7 \frac{5}{2}$$

$$e_j = 7 \quad x\ 7$$

$$e_j = 3$$

The magnitude of the error is 3. Therefore, to correct the error: **1 (001_2) XOR 3 (011_2) = 2 (010_2)**.

R(x) is now correct:

$$\mathbf{R(x) = 2x^6 + 2x^5 + 2x^4 + 2x^3 + 2x^2 + 2x + 2}$$

5 Appendix 2 - Worked Example 2

Codeword to transmit M(x):

$$\mathbf{M(x) = 5x^4 + 3x^3 + 6x^2 + 7x^1 + 2}$$

Generator polynomial G(x):

$$\mathbf{G(x) = 1x^2 + 6x + 3}$$

Multiply M(x) by $\mathbf{x^2}$ to accommodate 2 x FEC symbols:

$$\mathbf{M(x)\ (x^2) = 5x^6 + 3x^5 + 6x^4 + 7x^3 + 2x^2 + 0x + 0}$$

Divide M(x) by the generator polynomial G(x) = $\mathbf{1x^2 + 6x + 3}$ to create the systematic codeword C(x):

$$C(x) = 5x^6 + 3x^5 + 6x^4 + 7x^3 + 2x^2 + 5x + 7$$

Introduce an error into symbol $6x^4$ such that it becomes $5x^4$:

$$E(x) = 3x^4\ (110_2\ \text{XOR}\ 011_2 = 101_2)$$

The receiver R(x) codeword:

$$R(x) = 5x^6 + 3x^5 + 5x^4 + 7x^3 + 2x^2 + 5x + 7$$

Create the syndrome polynomial S(x):

		$5x^6$	$3x^5$	$5x^4$	$7x^3$	$2x^2$	$5x^1$	7	Parity
S_0	$a^0 = 1$	$5(a^0)^6$	$3(a^0)^5$	$5(a^0)^4$	$7(a^0)^3$	$2(a^0)^2$	$5(a^0)^1$	$7(a^0)^0$	
		101	011	101	111	010	101	111	4 (100_2)

Table A2-1: R(x) S_0 Syndrome

		$5x^6$	$3x^5$	$5x^4$	$7x^3$	$2x^2$	$5x^1$	7	XOR
S_1	$a^1 = 2$	$5(a^1)^6$	$3(a^1)^5$	$5(a^1)^4$	$7(a^1)^3$	$2(a^1)^2$	$5(a^1)^1$	$7(a^1)^0$	
		111	010	011	010	011	001	111	1

Table A2-2: R(x) S_1 Syndrome

		$5x^6$	$3x^5$	$5x^4$	$7x^3$	$2x^2$	$5x^1$	7	XOR
S_2	$a^2 = 4$	$5(a^2)^6$	$3(a^2)^5$	$5(a^2)^4$	$7(a^2)^3$	$2(a^2)^2$	$5(a^2)^1$	$7(a^2)^0$	
		110	101	001	110	111	010	111	6

Table A2-3: R(x) S_2 Syndrome

The syndrome polynomial S(x):

$$S(x) = 6x^2 + 1x + 4$$

Run the Euclidian algorithm to determine the error locator and magnitude polynomials **Δ(x)** and **Ω(x)**:

Dividend					Divisor				Quotient		Remainder	
x^3	x^2	x^1	x^0		x^2	x^1	x^0		x^1	x^0	x^1	x^0
1	0	0	0	÷	6	1	4	=	3	5	2	2
	6	1	4	÷		2	2	=	3	6		3
		2	2	÷			3	=	7	7	0	0

Table A2-4: Error Locator and Magnitude Polynomials

$$\Delta(x) = 3x + 5$$

$$\Omega(x) = 2x + 2$$

Run the Chien search algorithm using **Δ(x) = 3x + 5**:

Δ(x)		a^n		Inverse a^n		Sum		Error = 0
Δx^{-6}	→	$3(a^{-6}) + 5$	=	$3(a^1) + 5$	=	6 + 5	=	3
Δx^{-5}	→	$3(a^{-5}) + 5$	=	$3(a^2) + 5$	=	7 + 5	=	2
Δx^{-4}	→	$3(a^{-4}) + 5$	=	$3(a^3) + 5$	=	5 + 5	=	0
Δx^{-3}	→	$3(a^{-3}) + 5$	=	$3(a^4) + 5$	=	1 + 5	=	4
Δx^{-2}	→	$3(a^{-2}) + 5$	=	$3(a^5) + 5$	=	2 + 5	=	7

Δx^{-1}	→	$3(a^{-1}) + 5$	=	$3(a^{6}) + 5$	=	4 + 5	=	1
Δx^{-0}	→	$3(a^{-0}) + 5$	=	$3(a^{0}) + 5$	=	3 + 5	=	6

Table: A2-5: Chien Search Algorithm

Correct the error at $\mathbf{a^4}$ using $\Omega(x) = 2x + 2$. $\Delta(x) = 3x + 5$ becomes $\Delta(x) = 3$:

$$e_j = a^4 \frac{(2a^{-4} + 2)}{3}$$

$$e_j = a^4 \frac{(2a^{3} + 2)}{3}$$

$$e_j = a^4 \frac{(6 + 2)}{3}$$

$$e_j = 6 \frac{4}{3}$$

$$e_j = 6 \quad x\ 5$$

$$e_j = 3$$

The magnitude of the error is 3. Therefore, to correct the error: **5 (101_2) XOR 3 (011_2) = 6 (110_2)**.

R(x) is now correct:

$$R(x) = 5x^6 + 3x^5 + 6x^4 + 7x^3 + 2x^2 + 5x + 7$$

6 Appendix 3 - Extended Euclidian Algorithm

When applied to algebra, the Euclidian algorithm can be used to determine the common factors of equations and their coefficients. For example, in the equation:

$$ax + by = c\ gcd(a,b)$$

c is the GCD of the two variables **x** and **y** that have coefficients **a** and **b** i.e. **c** divides into both **a** and **b**. Table A3-1 below illustrates an extension of the Euclidean algorithm that allows values of the variables and their coefficients to be calculated. The approach is to work down steps 1 to 4 on the left hand side for the values of **a** and **b** and to work upwards from steps 4 to 1 to calculate the values of **x** and **y** on the right hand side. Note that **q** is the quotient and **r** the remainder. The starting values of **a** and **b** are **a = 657** and **b = 306**.

Step	a		b		q		r	(gcd) c	x	y	Step
1	657	=	306	x	2	+	45	9	7	-15	4
2	306	=	45	x	6	+	36	9	-1	7	3
3	45	=	36	x	1	+	9	9	1	-1	2
4	36	=	9	x	4	+	0	9	0	1	1

Table A3-1: Extended Euclidean Algorithm

The GCD (or common factor) **c** is found to be 9. On the right-hand side, the value of **y** in step 1 is the value of **x** in step 2; **y** in step 2 is **x** in step 3; **y** in step 3 is **x** in step 4. This facilitates calculation of **y** when moving up the steps from 1 to 4. Substituting the values for **x** and **y** into the equation confirms that the values are correct:

Step	a	x		b	y		c
4	36	0	+	9	1	=	**9**
3	45	1	+	36	-1	=	**9**
2	306	-1	+	45	7	=	**9**
1	657	7	+	306	-15	=	**9**

Table A3-2: Extended Euclidean Algorithm Results