Reed-Solomon codes: basic concepts and implementation

Vladimir Gerasik

February 8, 2016

1 Introduction

Reed-Solomon codes are block-based error correcting codes with a wide range of applications in digital communications and storage. Reed-Solomon codes are used to correct errors in many systems including:

- Storage devices (including tape, Compact Disk, DVD, barcodes, etc)
- Wireless or mobile communications (including cellular telephones, microwave links, etc)
- Satellite Communications
- Digital Television / Digital Video Broadcasting
- High-speed modems such as ADSL, xDSL, etc.

A typical data transmission system through the noisy channel is shown below.

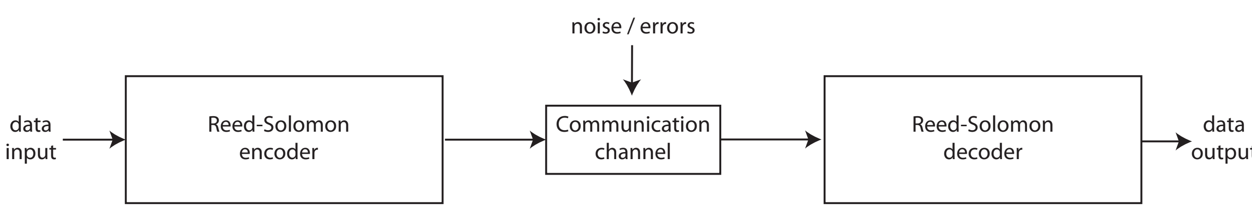

Figure 1: Encoded data transmission through the noisy channel

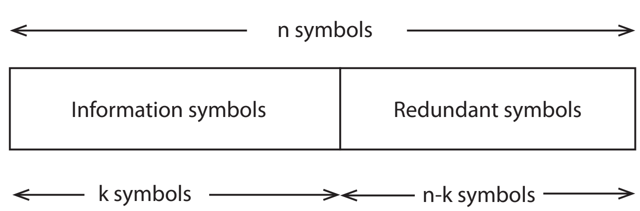

Figure 2: Block encoding for RS error correction

Error correcting codes use the same principle: redundancy is added to the information in order to correct errors which occur in the transmission process. Reed-Solomon codes are

block codes which treat the blocks of information symbols independently in a consecutive manner. A **cyclic code** is a block code, where the circular shifts of each codeword gives another word that belongs to the code.

Reed-Solomon codes are non-binary cyclic block codes with symbols made up of m-bit sequences, where m is any positive integer having a value greater than 2. RS(n, k) codes on m-bit symbols exist for all n and k for which

$$0 < k < n < 2^m + 2, \tag{1}$$

where k is the number of data symbols being encoded, and n is the total number of code symbols in the encoded block.

For the most conventional RS(n, k) code,

$$(n, k) = (2^m - 1, 2^m - 1 - 2t), \tag{2}$$

where t is the symbol-error correcting capability of the code, and $n - k = 2t$ is the number of parity check symbols. The RS(n,k) code is capable of correcting any combination of t or fewer errors, where t can be expressed as

$$t = \left[\frac{n-k}{2} \right], \tag{3}$$

where square brackets $[x]$ denote the largest integer not exceeding x.

For example, RS(7,3) code operates on 3-bit symbols, the information block consists of $k = 3$ symbols, the codeword is $n = 7$ symbols long, and it is capable of correcting up to 2 symbol errors. Let the information is represented as a sequence of $k = 3$ symbols [3 0 2], or [011 000 010] in binary representation, then the encoded word is [3 0 2 7 1 5 4] = [011 000 010 111 001 101 100] with $n - k = 4$ parity check symbols. Any 2 symbol errors in the codeword can be corrected by the RS decoder. Note, that this code is capable of correcting of up to 4-bit contiguous **burst errors**, since when a symbol is wrong, it might as well be wrong in all of its bit positions. This gives the RS code a tremendous burst-noise advantage over binary codes. Thus, for example, RS(255,247) code is capable of correcting of up to 4 errors, or any 25-bit long burst noise. In this example, if the 25-bit noise disturbance had occurred in a random fashion rather than as a contiguous burst, it should be clear that many more than four symbols would be affected (as many as 25 symbols might be disturbed). Of course, that would be beyond the capability of the (255, 247) code.

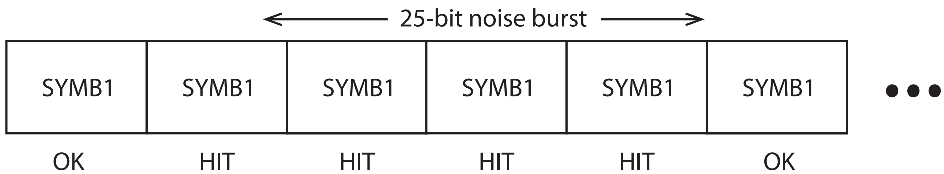

Figure 3: 25-bit noise burst for RS(255,247) code

A brief summary of Reed-Solomon terminology:
- Symbol length m is the number of bits per symbol

- Code word is the block of n symbols
- RS (n, k) code: n is the total number of symbols per code word,
k is the number of information symbols per code word
- Code rate is equal to k/n
- Number of parity check symbols $r = n - k$
- Maximum number of symbols with errors that can be corrected is $t = [(n-k)/2]$

2 Finite fields GF(2^m)

Finite fields, or Galois fields (GF) are used in most of the known construction of codes, and for decoding. The encoding and decoding algorithms for Reed-Solomon codes are developed using the concepts of finite fields. A **finite field** is a field that contains a finite number of elements. As with any field, a finite field is a set on which the operations of multiplication, addition, subtraction and division are defined and satisfy certain rules. For any prime number, p, there exists a finite field denoted GF(p) that contains p elements. Elements from the field GF(2^m) are used in the construction of Reed-Solomon RS($2^m - 1, k$) codes.

Primitive element and primitive polynomial of GF(2^m). An element $\alpha \in GF(2^m)$ is called primitive if any nonzero element $\beta \in GF(2^m)$ can be represented as $\beta = \alpha^j$, $0 \leq j \leq 2^m - 1$. This element is the root of irreducible polynomial, called primitive polynomial, $p(x)$ over $\{0, 1\}$, that is, $p(\alpha) = 0$. A primitive element α of the field GF(2^m) satisfies the equation $\alpha^{2^m-1} = 1$.

There are 4 different equivalent ways to represent the elements of the finite fields, that is 1) powers of primitive element, 2) in terms of polynomials with binary coefficients, 3) binary and 4) decimal representations.

Example 1. Construction of GF(2^3). Let α be a primitive element of GF(2^3), such that $p(\alpha) = \alpha^3 + \alpha + 1 = 0$ and $\alpha^7 = 1$. Then the elements of the GF(2^3) can be represented as follows,

Power	Polynomial	Binary	Decimal
-	0	000	0
α^0	1	001	1
α^1	α	010	2
α^2	α^2	100	4
α^3	$1+\alpha$	011	3
α^4	$\alpha+\alpha^2$	110	6
α^5	$1+\alpha+\alpha^2$	111	7
α^6	$1+\alpha^2$	101	5

Table 1: Various representations of GF(2^3) elements

Addition and Multiplication over GF. Addition of the elements of GF is defined as bitwise XOR (exclusive or) binary operation, also denoted as $\oplus$. Consider the addition of

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Table 2: Various representations of GF(2^3) elements

the two elements from the field in Example 1. Thus, for instance

$$\begin{aligned} 3+6 &= 011 \oplus 110 = 101 = 5, && (4)\\ 4+0 &= 100 \oplus 000 = 100 = 4, && (5)\\ 2+2 &= 010 \oplus 010 = 000 = 0. && (6) \end{aligned}$$

Obviously, in general we have $a + a = 0$, and $a + 0 = a$. To illustrate isomorphism of various representations we add the same elements using polynomial representation.

$$\begin{aligned} 3+6 &= 1+\alpha+\alpha+\alpha^2 = 1+\alpha^2 = 5, && (7)\\ 4+0 &= \alpha+\alpha^2+0 = \alpha+\alpha^2 = 4, && (8)\\ 2+2 &= \alpha^2+\alpha^2 = 0. && (9) \end{aligned}$$

Multiplication of the elements is better explained using power representation. It is important to keep in mind that by definition $\alpha^{2^m-1} = 1$, so that in our example $\alpha^7 = 1$. Thus, for example,

$$\begin{aligned} 3\cdot 6 &= \alpha^3\cdot\alpha^4 = \alpha^7 = 1, && (10)\\ 4\cdot 0 &= \alpha^2\cdot 0 = 0, && (11)\\ 7\cdot 5 &= \alpha^5\alpha^6 = \alpha^{11} = \alpha^7\alpha^4 = \alpha^4 = 4. && (12) \end{aligned}$$

In general, when using power representation, the multiplication is simply addition of powers modulo $2^m - 1$, so that

$$\alpha^a\alpha^b = \alpha^{(a+b) \bmod 2^m-1}. \tag{13}$$

The binary **modulo operation** mod finds the remainder after division of one number by another, e.g. 7 mod 2 = 1, 12 mod 3 = 0, or we say that two numbers are congruent modulo n, e.g. $38 = 14 \mod 12$, $25 = 13 \mod 12$.

Using polynomial representations, we arrive at the same result,

$$\begin{aligned} 3\cdot 6 = \quad & (\alpha+1)(\alpha^2+\alpha) \bmod (\alpha^3+\alpha+1) = && (14)\\ & \alpha^3+\alpha+\alpha+1 \bmod (\alpha^3+\alpha+1) = && (15)\\ & \alpha^3+1 \bmod (\alpha^3+\alpha+1) = 1. && (16) \end{aligned}$$

$$
\begin{array}{r|l}
 & \alpha^2 + 1 \\
\hline
\alpha^3 + \alpha + 1 & \alpha^5 \\
 & \underline{\alpha^5 + \alpha^3 + \alpha^2} \\
 & \quad \alpha^3 + \alpha^2 \\
 & \quad \underline{\alpha^3 + \alpha + 1} \\
 & \qquad \alpha^2 + \alpha + 1
\end{array}
$$

Figure 4: Long division of the two polynomials over GF

Here the modulo operation mod is applied to polynomials. To find the result of binary mod operation long division is often used as illustrated above. In this example we prove that $\alpha^5 = \alpha^2 + \alpha + 1 \bmod \alpha^3 + \alpha + 1$ (see Table 1, line 7) by dividing α^5 by primitive polynomial, and evaluating the reminder of the division. We note in passing that polynomial division is a core procedure for the systematic encoding algorithm.

2.1 Log and Antilog tables.

A convenient way to perform both multiplications and additions in GF is to use the look-up tables which allow for changing between the different representations. Consider the table for the previous GF(2^3) example.

index, i	Antilog table, Alog(i)	Log table, Log(i)
0	1	-1
1	2	0
2	4	1
3	3	3
4	6	2
5	7	6
6	5	4
7	0	5

Table 3: Log / antilog table for the GF(2^3)

The following equalities hold:

$$\alpha^i = \text{Alog}(i), \tag{17}$$

$$\log(\alpha^i) = i, \tag{18}$$

$$\alpha^{\log(\alpha^i)} = \text{Alog}(i), \tag{19}$$

where Alog points out the decimal representation of the element.

To illustrate the application of the look-up table we first verify the fact that the primitive element α is the root of the primitive polynomial $\alpha^3 + \alpha + 1$, indeed

$$\alpha^3 + \alpha + 1 = \text{Alog}(3) \oplus \text{Alog}(1) \oplus \text{Alog}(0) = 011 \oplus 010 \oplus 001 = 0. \tag{20}$$

Now consider the polynomial $f(x)$ over the GF(8),

$$f(x) = \alpha^4 x^2 + \alpha^5 x + \alpha^3 = 6x^2 + 7x + 3. \tag{21}$$

Assume, we need to evaluate this polynomial at the element $x = \alpha^2 = 4$. Using the look-up tables we proceed as follows:

$$f(\alpha^2) = \text{Alog}\{\log[\text{Alog}(4)] + 2\log[\text{Alog}(2)]\} \oplus \tag{22}$$
$$\text{Alog}\{\log[\text{Alog}(5)] + \log[\text{Alog}(2)]\} \oplus \text{Alog}\{\log[\text{Alog}(3)]\} = \tag{23}$$
$$\text{Alog}(8\%7) \oplus \text{Alog}(7\%7) \oplus \text{Alog}(3) = \tag{24}$$
$$\text{Alog}(1) \oplus \text{Alog}(0) \oplus \text{Alog}(3) = 2 \oplus 1 \oplus 3 = 0. \tag{25}$$

It is important to stress that that all power summations are implied modulo 7, i.e. $8\%7 = 1$, $7\%7 = 0$ (note that % operator is used in C). In practice, it is more convenient to use decimal representations of the elements, so that $f(x) = 6x^2 + 7x + 3$, $x = 4$. In this case we proceed similarly,

$$f(4) = \text{Alog}[\log(6) + 2\log(4)] \oplus \text{Alog}[\log(7) + \log(4)] \oplus \text{Alog}[3] = 0. \tag{26}$$

Decimal representation of the polynomial coefficients is thus used in the C code. Note that the polynomial evaluation problem can be easily solved in MatLab. The corresponding MatLab code for self-check reads (I used short codes like this for testing purposes)

```
clear all;
m = 3;
apoly = gf([6 7 3],m,11);    % Create polynomial over GF(2^3)
x0 = gf([ 4 ],m,11);         % Points at which to evaluate the polynomial
y = polyval(apoly,x0)        % Result
```

2.2 Linear feedback shift register

First practical problem to solve is generation of the Galois field GF(2^m) and corresponding look-up table for a given m, primitive polynomial, and primitive element. In practice, we do not have to find primitive polynomial ourselves, since these are known for various different values of m. The table of primitive polynomials contains is provided below. Here the polynomials are represented in terms of natural numbers (this is the approach used in MatLab, I also use it in the C code). To convert the number into actual polynomial we use its binary representation, e.g. for $m = 8$ we choose the polynomial 369, since $369 = 101110001$, the corresponding polynomial is

$$D^8 + D^6 + D^5 + D^4 + 1, \tag{27}$$

m	Polynomial
2	7
3	11, 13
4	19, 25
5	37, 41, 47, 55, 59, 61
6	67, 91, 97, 103, 109, 115
7	131, 137, 143, 145, 157, 167, 171, 185, 191, 193, 203, 211, 213, 229, 239, 241, 247, 253
8	285, 299, 301, 333, 351, 355, 357, 361, 369, 391, 397, 425, 451, 463, 487, 501,
9	529, 539, 545, 557, 563, 601, 607, 617, 623, 631, 637, 647, 661, 675, 677, 687, 695, 701, 719, 721, 731, 757, 761, 787, 789, 799, 803, 817, 827, 847, 859, 865, 875, 877, 883, 895, 901, 911, 949, 953, 967, 971, 973, 981, 985, 995, 1001, 1019
10	1033, 1051, 1063, 1069, 1125, 1135, 1153, 1163, 1221, 1239, 1255, 1267, 1279, 1293, 1305, 1315, 1329, 1341, 1347, 1367, 1387, 1413, 1423, 1431, 1441, 1479, 1509, 1527, 1531, 1555, 1557, 1573,1591, 1603, 1615, 1627, 1657, 1663, 1673, 1717, 1729, 1747, 1759, 1789, 1815, 1821, 1825, 1849, 1863, 1869, 1877, 1881, 1891, 1917, 1933, 1969, 2011, 2035, 2041

Table 4: Primitive polynomials for GF(2^m), $2 \leq m \leq 10$ in decimal representation

and so on. We refer to the first polynomial in a row as the default primitive polynomial.

To generate the look-up table is relatively easy task provided we are familiar with the idea of Linear Feedback Shift register (LFSR). A LFSR is a shift register whose input bit is a linear function of its previous state. The most commonly used linear function of single bits is exclusive-or (XOR). Thus, an LFSR is most often a shift register whose input bit is driven by the XOR of some bits of the overall shift register value.

As an example, consider 8-bit Galois LFSR with the default primitive polynomial 285 with binary representation 100011101. The corresponding LFSR is shown on the figure. Assume the initial content of the register is unity (00000001). When the system is clocked,

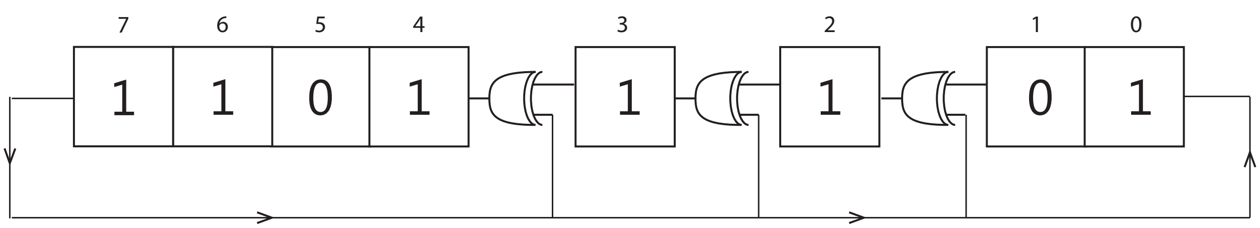

Figure 5: 8-bit LFSR for GF(2^8) with primitive polynomial $D^8 + D^4 + D^3 + D^2 + 1$

bits that are not taps are shifted one position to the left unchanged. The new output bit is the next input bit. The effect of this is that when the output bit is zero all the bits in the register shift to the left unchanged, and the input bit becomes zero. When the output bit is one, the bits in the tap positions all flip (if they are 0, they become 1, and if they are 1, they become 0), and then the entire register is shifted to the left and the input bit becomes

1. For example, the content of the register thus reads,

0. 00000001
1. 00000010
2. 00000100
3. 00001000
4. 00010000
5. 00100000
6. 01000100
7. 10000100
8. 00011101
9. 00111010
10. 01110100

...

Note, that it is at step 9. the output bit becomes 1, and this is when XOR gates take effect. The content of the register thus lists all of 2^8 elements of the field (except zero) in binary format and then repeats itself. It is easy to see that in such a way we consequently generate the elements α^0, α^1, ... , α^{10}, ... , so that when we keep track of the clock the anti-log table is readily obtained. Similarly, we also obtain the log table, and the whole function is no longer than 10 lines of C code!

3 Encoding methods of RS codes

The coding methods are based on the polynomial representation of messages and codewords. For example, consider RS(7,3) code with the code length is $n = 7$, and the message length $k = 3$. The message is thus consists of three 3-bit symbols, e.g. $u = [2\ 5\ 1]$, or equivalently, $u = [\alpha\ \alpha^6\ \alpha^0]$. The corresponding **information polynomial** thus reads,

$$u = \alpha x^2 + \alpha^6 x + \alpha^0. \tag{28}$$

In the encoding procedure the corresponding **code polynomial** $v(x)$ is sought,

$$v(x) = v_{n-1}x^{n-1} + v_{n-2}x^{n-2} + ...v_1 x + v_0. \tag{29}$$

In our example this is a polynomial of degree 6 (3 information symbols + 4 parity checks).

The generator polynomial of the code. The generator polynomial $g(x)$ is the polynomial such that all code polynomials $v(x)$ are the multiples of this polynomial, $v(x) = a(x)g(x)$. The generator polynomial is specified by its roots, called, the **roots of the code**. For the RS(n,k) code the generating polynomial is $n - k$ degree polynomial,

$$g(x) = \prod_{j=b}^{j=n-k-1+b} (x + \alpha^j) = x^{n-k} + g_{n-k-1}x^{n-k-1} + ...g_1 x + g_0, \tag{30}$$

where b value is usually set to 0 or 1. The recursive algorithm is implemented in C to expand the product, and find polynomial coefficients g_i.

Non-systematic / Systematic encoding. Let $u(x)$ denote the information polynomial, $g(x)$ is the generator polynomial. Then encoding of a code word is either non-systematic or systematic:

- Non-systematic encoding

$$v(x) = u(x)g(x), \tag{31}$$

- Systematic encoding

$$v(x) = x^{n-k}u(x) + x^{n-k}u(x) \bmod g(x). \tag{32}$$

Note, that we use systematic encoding is the C code. The codeword in systematic form is given by (in vector form)

$$v = [v_{n-1}, v_{n-2}, ..., v_1, v_0] = [u_{k-1}, u_{k-2}, ..., u_1, u_0, p_{n-k-1}, p_{n-k-2}, ..., p_1, p_0], \tag{33}$$

where we have k informational symbols and $n-k$ parity check symbols. Note that x^{n-k} factor in the expression $x^{n-k}u(x)$ simply shifts the symbols in $u(x)$ positions to the left $(n-k)$. To find the reminder polynomial $p(x) = x^{n-k}u(x) \bmod g(x)$ we have to perform division of the two polynomials.

This can be implemented using the corresponding LFSR. Consider the example of division of the two polynomials, let the input symbols are data symbols as before $u = [2\ 5\ 1]$. The generator polynomial is $g(x) = \alpha^3 + \alpha x + x^2 + \alpha^3 x^3 + x^4$, so that $g_0 = \alpha^3$, $g_1 = \alpha$, $g_2 = \alpha^0$, $g_3 = \alpha^3$. Initially, we set the content of the register to zero. At the first clock cycle we feed the

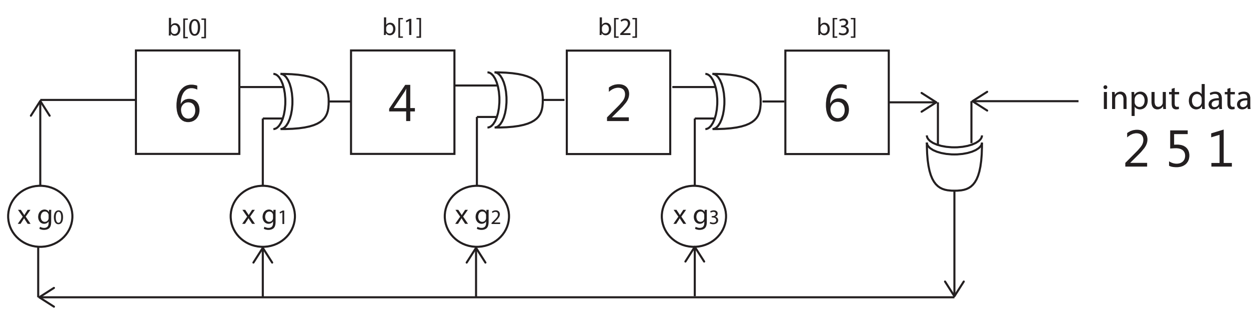

Figure 6: 4-bit LFSR polynomial divider circuit over GF(2^3)

first information symbol, add it with the content of the last bit to generate the feedback. The feedback is further multiplied by the corresponding coefficients of the generator polynomial and XOR'ed with the the previous values in the register. After k steps the content of the register represents the reminder of the division of the two polynomials. For example, the content of the register for the above values

0. 0 0 0 0 input 2
1. 6 4 2 6 input 5
2. 5 0 7 7 input 1
3. 1 2 6 6

provides the reminder in k steps. Finally, for the information sequence [2 5 1] we thus found the corresponding codeword [2 5 1 6 6 2 1]. The implementation of the above circuit is the core part of the encoding method.

4 Decoding methods of RS codes

The decoding procedure consists of several steps which can be summarized in the following diagram. In the following we will discuss the above stages in a consecutive manner.

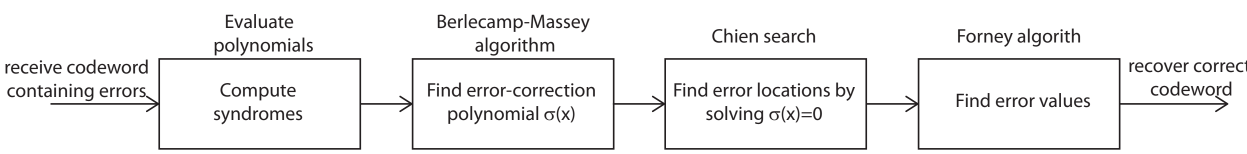

Figure 7: Architecture of the RS decoder with GF(2^m) arithmetics

Assume the decoder receives the codeword of length n which contains errors, so that it can be represented in polynomial form,

$$r(x) = v(x) + e(x), \tag{34}$$

where $v(x)$ denotes the correct codeword, and $e(x)$ is the error polynomial. For example, in our previous example the original message was [2 5 1]. The encoded codeword sent through the channel is then [2 5 1 6 6 2 1], or in polynomial form,

$$\begin{aligned} v(x) &= 2x^6 + 5x^5 + x^4 + 6x^3 + 6x^2 + 2x + 1 = \quad (35)\\ & \alpha x^6 + \alpha^6 x^5 + \alpha^0 x^4 + \alpha^4 x^3 + \alpha^4 x^2 + \alpha x + \alpha^0. \quad (36)\end{aligned}$$

The RS(7,3) code is capable of correcting up to 2 errors. Assume that after transmission through the noisy channel the decoder receives the noisy signal [2 5 3 6 2 2 1], so that

$$\begin{aligned} r(x) &= 2x^6 + 5x^5 + 3x^4 + 6x^3 + 2x^2 + 2x + 1 = v(x) + e(x), \quad (37)\\ e(x) &= 2x^4 + 4x^2. \quad (38)\end{aligned}$$

The purpose of the decoder is thus to determine both error locations and error values. This information is contained in the error polynomial $e(x)$.

Syndromes The syndromes are the values of the received polynomial $r(x)$ at the errors of the code. If the received codeword contains no errors all the syndromes are zeros. For example, we received the codeword $r(x) = 2x^6 + 5x^5 + 3x^4 + 6x^3 + 2x^2 + 2x + 1$, and the roots of RS(7,3) code are $\alpha, \alpha^2, \alpha^3, \alpha^4$, then the syndromes

$$\begin{aligned} S_1 &= r(\alpha) = 1 && (39) \\ S_2 &= r(\alpha^2) = 1 && (40) \\ S_3 &= r(\alpha^3) = 7 && (41) \\ S_4 &= r(\alpha^4) = 0 && (42) \end{aligned}$$

Say, we want to evaluate these by hand, then we obtain, for instance,

$$\begin{aligned} S_1 = r(\alpha^2) &= \alpha\alpha^{12} + \alpha^6\alpha^{10} + \alpha^3\alpha^8 + \alpha^4\alpha^6 + \alpha\alpha^4 + \alpha\alpha^2 + \alpha^0 = && (43) \\ &= \alpha^6 + \alpha^2 + \alpha^4 + \alpha^3 + \alpha^5 + \alpha^3 + \alpha^0 = && (44) \\ &= 5 \oplus 4 \oplus 7 \oplus 1 = 7 = \alpha^5. && (45) \end{aligned}$$

We can also verify that when the received signal is intact, $v(\alpha^i) = 0$, $i = 1..4$. The implementation of these operations is straightforward using previously generated look-up tables.

4.1 Berlecamp-Massey algorithm

The goal of the Berlecamp-Massey algorithm (BMA) is to produce the **error-locator polynomial** $\sigma(x)$, such that the inverse roots of this polynomial point out the locations of the error occurrences in the received codeword $r(x)$. Other algorithms known to perform this task are Euclidean algorithm and PGZ decoder, however, BMA is the most common one to be implemented in C.

The BMA algorithm is initialized with $\sigma(x) = 1$ (connection polynomial), $\rho(x) = x$ (correction term), $i = 1$ (syndrome sequence counter), $l = 0$ (register length). The BMA algorithm is provided in the form of diagram below. Consider the implementation of BMA in the particular case of the syndrome sequence from the above example, $S_1 = 1$, $S_2 = 1$, $S_3 = 7$, $S_4 = 0$.

- $i = 0$ $\quad \sigma(x) = 1$, $l = 0$, $\rho(x) = x$
- $i = 1$ $\quad d = S_1 = 1$,

$$\begin{aligned} \sigma_{new}(x) = \sigma(x) + d\rho(x) = 1 + x \\ 2l = 0 < i,\ l = i - l = 1, \\ \rho(x) = \sigma(x)/d = 1, \\ \rho(x) = x\rho(x) = x, \\ \sigma(x) = \sigma_{new}(x) = 1 + x \end{aligned}$$

- $i = 2$ $\quad d = S_2 + \sigma_1 S_1 = 1 + 1 \times 1 = 0$

$$\rho(x) = x\rho(x) = x^2$$

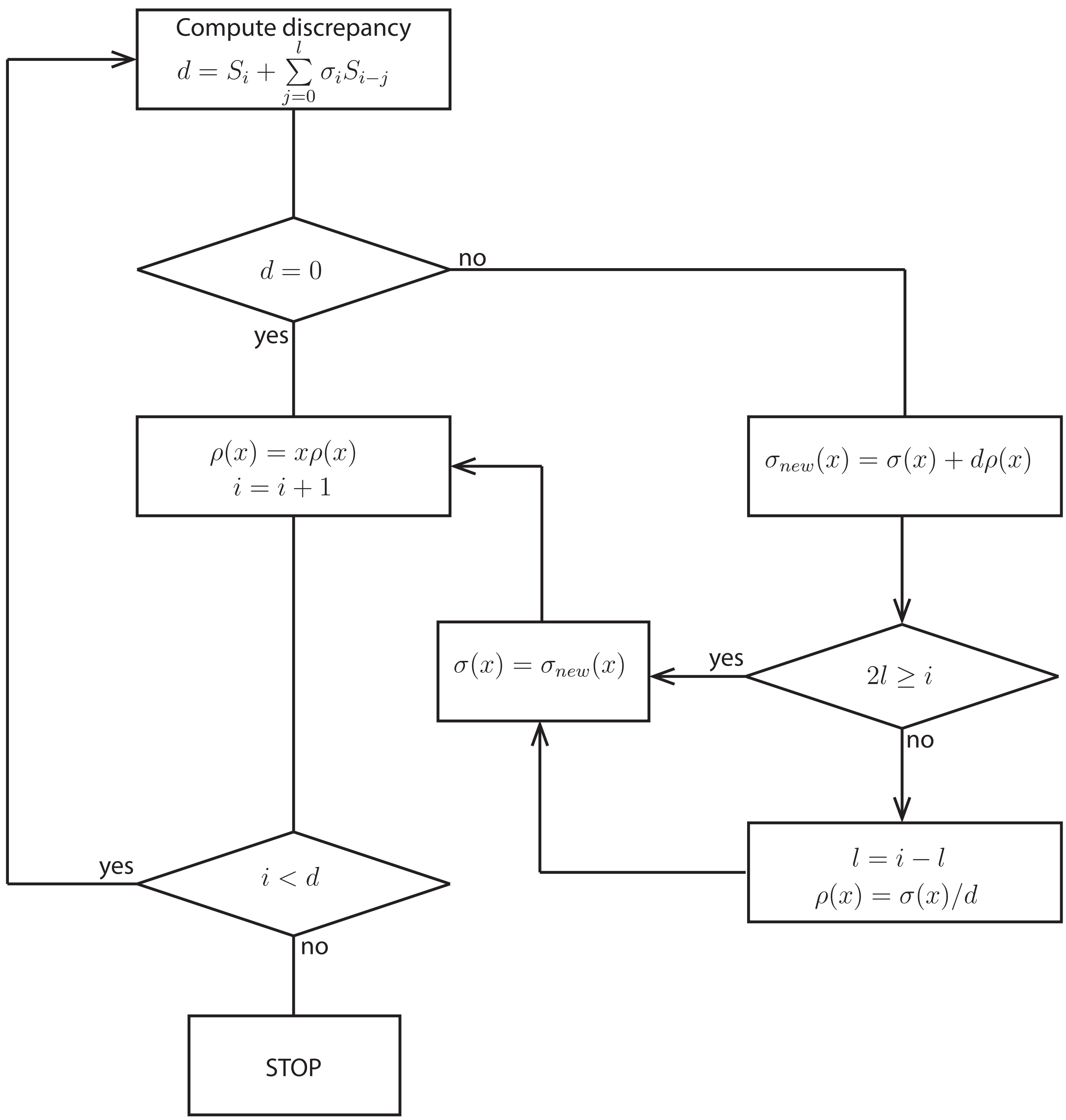

Figure 8: Diagram of BMA algorithm

- $i = 3 \qquad d = S_3 + \sigma_1 S_2 = 7 + 1 \times 1 = 6$

$$\begin{aligned} \sigma_{new}(x) = \sigma(x) + d\rho(x) = 1 + x + 6x^2 \\ 2l < i,\ l = i - l = 2, \\ \rho(x) = \sigma(x)/d = (1+x)/6 = 3 + 3x, \\ \rho(x) = x\rho(x) = 3x + 3x^2, \\ \sigma(x) = \sigma_{new}(x) = 1 + x + 6x^2 \end{aligned}$$

- $i = 4 \qquad d = S_4 + \sigma_1 S_3 + \sigma_2 S_2 = 0 + 1 \times 7 + 6 \times 1 = 1$

$$\begin{aligned} \sigma_{new}(x) = \sigma(x) + d\rho(x) = 1 + x + 6x^2 + 3x + 3x^2 = 1 + 2x + 5x^2 \\ 2l = 4 = i, \\ \rho(x) = x\rho(x) = 3x^2 + 3x^3, \\ \sigma(x) = \sigma_{new}(x) = 1 + 2x + 5x^2 \end{aligned}$$

- $i = 5 > d \qquad$ STOP

4.2 Chien search

The BMA thus returned the following error-locator polynomial:

$$\sigma(x) = 1 + 2x + 5x^2 = 1 + \alpha x + \alpha^6 x, \tag{46}$$

which can be factorized as follows,

$$\sigma(x) = (1 + \alpha^2 x)(1 + \alpha^4 x), \tag{47}$$

so we may now predict the errors at positions 2 and 4 (compare the encoded codeword [2 5 1 6 6 2 1] and the received codeword [2 5 3 6 2 2 1]). Note that the numbering here is from the right starting with position zero.

In general, the roots of the error-locator polynomial $\sigma(x)$ are sought using the trial-and-error procedure called **Chien search**. The sequence of all nonzero finite elements 1, α, α^2,...,α^{2^m-2} is generated, and then the condition $\sigma(\alpha^i) = 0$ is tested. This procedure is simple to implement both in the software and hardware versions.

4.3 Forney Algorithm

To compute the error values at the positions predicted by the Chien search we use the **Forney algorithm**. Consider the RS(n, k) code with the roots $\alpha^b, \alpha^{b+1}, ..., \alpha^{b+n-k-1}$, where $b = 0$, or $b = 1$. Then, the error values e_j at positions j can be calculated as follows,

$$e_j = \frac{(\alpha^j)^{2-b}\,\Lambda(\alpha^{-j})}{\sigma'(\alpha^{-j})}, \tag{48}$$

where the **error evaluator polynomial** is defined as follows,

$$\Lambda(x) = \sigma(x)S(x) \bmod x^{n-k+1}, \tag{49}$$

in terms of the error-locator polynomial $\sigma(x)$, and the **syndrome polynomial** $S(x)$,

$$S(x) = 1 + S_1 x + S_2 x^2 + ... + S_{n-k} x^{n-k}. \tag{50}$$

The expression σ' in the denominator (48) implies the **formal derivative** of the error-locator polynomial $\sigma(x) = \sum_{i=0}^{\nu} \sigma_i x^i$,

$$\sigma' = \sum_{i=1}^{\nu} i \cdot \sigma_i x^{i-1}, \tag{51}$$

where i is an integer, and σ_i is an element of the finite field. The operator $\cdot$ represents ordinary multiplication (repeated addition in the finite field) and not the finite field's multiplication operator, so that

$$i \cdot \sigma_i = \begin{cases} 0, & \text{if } i \text{ is even} \\ \sigma_i, & \text{if } i \text{ is odd} \end{cases} \tag{52}$$

and therefore, the implementation of the formal derivative is a simple task, e.g. in our example,

$$\sigma(x) = 1 + \alpha x + \alpha^6 x^2, \tag{53}$$
$$\sigma'(x) = \alpha + 2 \cdot \alpha^6 x = \alpha. \tag{54}$$

Example Consider the RS(7,3) code with the roots α^1, α^2, α^3, and α^4. The error-locator polynomial is $\sigma(x) = 1 + \alpha x + \alpha^6 x^2$, and the syndrome polynomial reads $S(x) = 1 + x + x^2 + \alpha^4 x^3$. The error locations are determined above are $j = 2$ and $j = 4$.

Using the Forney algorithm we obtain $\Lambda(x) = 1 + \alpha^3 x + \alpha^4 x^2$, and consequently,

$$e_2 = \frac{\alpha^2 \left(1 + \alpha^3 \alpha^{-2} + \alpha^4 \alpha^{-4}\right)}{\alpha} = \alpha\alpha = \alpha^2 = 4; \tag{55}$$
$$e_4 = \frac{\alpha^4 \left(1 + \alpha^3 \alpha^{-4} + \alpha^4 \alpha^{-8}\right)}{\alpha} = \alpha^3 \alpha^5 = \alpha = 2. \tag{56}$$

It is easy to see that the original [2 5 1 6 6 2 1] codeword is thus recovered from the received [2 5 3 6 2 2 1],

$$[2\ 5\ (3+2)\ 6\ (2+4)\ 2\ 1] = [2\ 5\ 1\ 6\ 6\ 2\ 1]. \tag{57}$$

Note that the numbering used in the C code (from the left) is different from the above (from the right).

5 Final remarks

Theoretical grounds of the error correcting codes theory are provided in many sources, [1, 2] to name a few. More detailed explanations are provided in [1], however, for practical engineering purposes [2] suites better. The companion web site of the book [3] contains many codes which are useful for the topics covered in the text. In general, there are several different implementations available for the RS coding and encoding. Note that, Galois finite fields are well implemented in MatLab libraries. MatLab RS implementations are really short codes based on these built-in subroutines.

When it comes to writing a low-level C code it is necessary to become familiar with the LFSR theory, since many important routines written in C are in fact realizations of the corresponding LFSR algorithms [4]. In this report the role of LSFR for C implementations is emphasized, and explained in a comprehensive manner. Each step is supplied with corresponding numerical examples.

The result of the sample run of the C code for RS(255,239) is attached below.

References

[1] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes*. NorthHolland, 1977.

[2] Robert H. Morelos-Zaragoza. *The Art of Error Correcting Coding* [*second edition*]. John Wiley & Sons, Ltd. 2006.

[3] This web site contains computer programs in C/C++ language and Matlab scripts to simulate basic algorithms for encoding/decoding and analysis of the important classes of error correcting codes that are covered in the textbook,
`http://the-art-of-ecc.com`

[4] Kewal K. Saluja. *Linear feedback shift registers theory and applications* [*lecture notes*]. University of Wisconsin-Madison. 1991.

```
REED_SOLOMON CODE RS(2^m-1,k) ENCODER AND DECODER.

Enter parameters m (symbol length) and k (message length):
8 239
Reed-Solomon code RS(255,239) over the field GF(2^8)
generated by the primitive polynomial: x^8+x^4+x^3+x^2+1
is capable of correcting of up to 8 errors.

The roots of the code are: a^1, a^2, a^3, a^4, a^5, a^6, a^7, a^8, a^9, a^10, a^11, a^12, a^13, a^14, a^15, a^16.

Table of GF(2^8)
-------------------------------------

  0   00000001     1    -1
  1   00000010     2     0
  2   00000100     4     1
  3   00001000     8    25
  4   00010000    16     2
  5   00100000    32    50
  6   01000000    64    26
  7   10000000   128   198
  8   00011101    29     3
  9   00111010    58   223
 10   01110100   116    51
 11   11101000   232   238
 12   11001101   205    27
 13   10000111   135   104
 14   00010011    19   199
 15   00100110    38    75
 16   01001100    76     4
 17   10011000   152   100
 18   00101101    45   224
 19   01011010    90    14
 20   10110100   180    52
 21   01110101   117   141
 22   11101010   234   239
 23   11001001   201   129
 24   10001111   143    28
 25   00000011     3   193
 26   00000110     6   105
 27   00001100    12   248
 28   00011000    24   200
 29   00110000    48     8
 30   01100000    96    76
 31   11000000   192   113
 32   10011101   157     5
 33   00100111    39   138
 34   01001110    78   101
 35   10011100   156    47
 36   00100101    37   225
 37   01001010    74    36
 38   10010100   148    15
 39   00110101    53    33
 40   01101010   106    53
 41   11010100   212   147
 42   10110101   181   142
 43   01110111   119   218
 44   11101110   238   240
 45   11000001   193    18
 46   10011111   159   130
 47   00100011    35    69
 48   01000110    70    29
 49   10001100   140   181
 50   00000101     5   194
 51   00001010    10   125
 52   00010100    20   106
 53   00101000    40    39
 54   01010000    80   249
 55   10100000   160   185
 56   01011101    93   201
 57   10111010   186   154
 58   01101001   105     9
 59   11010010   210   120
 60   10111001   185    77
 61   01101111   111   228
 62   11011110   222   114
 63   10100001   161   166
 64   01011111    95     6
 65   10111110   190   191
 66   01100001    97   139
 67   11000010   194    98
 68   10011001   153   102
 69   00101111    47   221
 70   01011110    94    48
 71   10111100   188   253
 72   01100101   101   226
 73   11001010   202   152
 74   10001001   137    37
 75   00001111    15   179
 76   00011110    30    16
 77   00111100    60   145
 78   01111000   120    34
 79   11110000   240   136
 80   11111101   253    54
 81   11100111   231   208
 82   11010011   211   148
 83   10111011   187   206
 84   01101011   107   143
 85   11010110   214   150
 86   10110001   177   219
 87   01111111   127   189
 88   11111110   254   241
 89   11100001   225   210
 90   11011111   223    19
 91   10100011   163    92
 92   01011011    91   131
 93   10110110   182    56
 94   01110001   113    70
 95   11100010   226    64
 96   11011001   217    30
 97   10101111   175    66
 98   01000011    67   182
 99   10000110   134   163
100   00010001    17   195
101   00100010    34    72
102   01000100    68   126
103   10001000   136   110
104   00001101    13   107
105   00011010    26    58
106   00110100    52    40
107   01101000   104    84
108   11010000   208   250
109   10111101   189   133
110   01100111   103   186
111   11001110   206    61
112   10000001   129   202
113   00011111    31    94
114   00111110    62   155
115   01111100   124   159
116   11111000   248    10
117   11101101   237    21
118   11000111   199   121
119   10010011   147    43
120   00111011    59    78
121   01110110   118   212
122   11101100   236   229
123   11000101   197   172
124   10010111   151   115
125   00110011    51   243
126   01100110   102   167
127   11001100   204    87
128   10000101   133     7
129   00010111    23   112
130   00101110    46   192
131   01011100    92   247
132   10111000   184   140
133   01101101   109   128
134   11011010   218    99
135   10101001   169    13
136   01001111    79   103
137   10011110   158    74
138   00100001    33   222
139   01000010    66   237
140   10000100   132    49
141   00010101    21   197
142   00101010    42   254
143   01010100    84    24
144   10101000   168   227
145   01001101    77   165
146   10011010   154   153
147   00101001    41   119
148   01010010    82    38
149   10100100   164   184
150   01010101    85   180
151   10101010   170   124
152   01001001    73    17
153   10010010   146    68
154   00111001    57   146
155   01110010   114   217
156   11100100   228    35
157   11010101   213    32
158   10110111   183   137
159   01110011   115    46
160   11100110   230    55
161   11010001   209    63
162   10111111   191   209
163   01100011    99    91
164   11000110   198   149
165   10010001   145   188
166   00111111    63   207
167   01111110   126   205
168   11111100   252   144
169   11100101   229   135
170   11010111   215   151
171   10110011   179   178
172   01111011   123   220
173   11110110   246   252
174   11110001   241   190
175   11111111   255    97
176   11100011   227   242
177   11011011   219    86
178   10101011   171   211
179   01001011    75   171
180   10010110   150    20
181   00110001    49    42
182   01100010    98    93
183   11000100   196   158
184   10010101   149   132
185   00110111    55    60
186   01101110   110    57
187   11011100   220    83
188   10100101   165    71
189   01010111    87   109
190   10101110   174    65
191   01000001    65   162
192   10000010   130    31
193   00011001    25    45
194   00110010    50    67
195   01100100   100   216
196   11001000   200   183
197   10001101   141   123
198   00000111     7   164
199   00001110    14   118
200   00011100    28   196
201   00111000    56    23
202   01110000   112    73
203   11100000   224   236
204   11011101   221   127
205   10100111   167    12
206   01010011    83   111
207   10100110   166   246
208   01010001    81   108
209   10100010   162   161
210   01011001    89    59
211   10110010   178    82
212   01111001   121    41
213   11110010   242   157
214   11111001   249    85
215   11101111   239   170
216   11000011   195   251
217   10011011   155    96
218   00101011    43   134
219   01010110    86   177
220   10101100   172   187
221   01000101    69   204
222   10001010   138    62
223   00001001     9    90
224   00010010    18   203
225   00100100    36    89
226   01001000    72    95
227   10010000   144   176
228   00111101    61   156
229   01111010   122   169
230   11110100   244   160
231   11110101   245    81
232   11110111   247    11
233   11110011   243   245
234   11111011   251    22
235   11101011   235   235
236   11001011   203   122
237   10001011   139   117
238   00001011    11    44
239   00010110    22   215
240   00101100    44    79
241   01011000    88   174
242   10110000   176   213
243   01111101   125   233
244   11111010   250   230
245   11101001   233   231
246   11001111   207   173
247   10000011   131   232
248   00011011    27   116
249   00110110    54   214
250   01101100   108   244
251   11011000   216   234
252   10101101   173   168
253   01000111    71    80
254   10001110   142    88
255   00000000     0   175
-------------------------------------
Generator polynomial (independent term first):
g(x) = a^136 a^240 a^208 a^195 a^181 a^158 a^201 a^100 a^11 a^83 a^167 a^107 a^113 a^110 a^106 a^121 a^0

Randomly generated message of length 239 sent:
2 5 1 1 192 98 141 195 46 54 245 38 121 62 100 132 157 82 227 19 13 238 145 197 2 51 152 108 18 46 97 34 4 93 4 232 227 111 3
 141 81 76 55 210 30 85 10 203 25 237 231 136 38 56 239 245 145 34 238 255 162 164 141 248 114 67 201 61 252 217 163 22 119 1
51 25 221 228 75 86 99 227 223 38 143 202 34 172 72 248 204 87 149 103 188 18 3 129 131 187 72 53 38 215 37 73 32 132 145 242
 167 174 19 162 48 134 112 33 53 48 134 223 54 188 120 137 254 88 11 218 39 196 80 180 224 6 132 111 238 94 52 113 241 124 16
4 254 233 241 114 82 248 228 119 237 106 6 238 47 242 200 216 36 54 203 159 161 142 151 249 144 9 145 15 136 178 39 223 161 1
7 78 222 134 244 46 226 203 111 27 64 129 33 200 49 8 185 183 123 37 185 244 77 84 122 3 106 96 112 23 137 48 176 133 252 228
 175 119 145 33 77 229 182 168 18 193 224 46 183 114 45 104 232 132 72 101 72 158 141 32 182 180

Encoded codeword of length 255
2 5 1 1 192 98 141 195 46 54 245 38 121 62 100 132 157 82 227 19 13 238 145 197 2 51 152 108 18 46 97 34 4 93 4 232 227 111 3
 141 81 76 55 210 30 85 10 203 25 237 231 136 38 56 239 245 145 34 238 255 162 164 141 248 114 67 201 61 252 217 163 22 119 1
51 25 221 228 75 86 99 227 223 38 143 202 34 172 72 248 204 87 149 103 188 18 3 129 131 187 72 53 38 215 37 73 32 132 145 242
 167 174 19 162 48 134 112 33 53 48 134 223 54 188 120 137 254 88 11 218 39 196 80 180 224 6 132 111 238 94 52 113 241 124 16
4 254 233 241 114 82 248 228 119 237 106 6 238 47 242 200 216 36 54 203 159 161 142 151 249 144 9 145 15 136 178 39 223 161 1
7 78 222 134 244 46 226 203 111 27 64 129 33 200 49 8 185 183 123 37 185 244 77 84 122 3 106 96 112 23 137 48 176 133 252 228
 175 119 145 33 77 229 182 168 18 193 224 46 183 114 45 104 232 132 72 101 72 158 141 32 182 180 135 140 138 111 176 247 20 1
93 89 88 213 57 139 141 34 57

Encoded codeword of length 255 passed through the noisy channel
2 5 1 1 192 98 141 195 46 54 245 38 121 62 100 132 157 82 227 19 13 238 145 197 2 51 152 108 18 46 97 34 4 93 4 232 227 111 3
 141 81 76 55 210 82 85 10 203 25 237 231 136 38 56 239 245 145 34 238 255 162 164 141 248 114 67 201 61 252 217 163 22 119 1
51 25 221 228 75 140 99 227 223 38 143 202 148 172 72 248 204 87 149 103 188 18 3 129 131 187 72 53 38 215 37 73 32 219 145 2
42 167 174 19 162 48 134 112 33 53 7 134 223 54 188 120 137 254 88 11 218 39 196 80 180 224 6 132 111 238 94 52 113 241 124 1
64 254 233 241 114 82 248 228 119 237 106 6 238 47 242 200 216 36 54 203 159 161 142 151 249 144 9 145 15 136 178 39 223 161
17 78 222 198 244 46 226 203 111 27 64 129 33 200 49 8 185 183 250 37 185 244 77 84 122 3 106 96 112 23 137 48 176 133 252 22
8 175 119 145 33 77 229 182 168 18 193 224 160 183 114 45 104 232 132 72 101 72 158 141 32 182 180 135 140 138 111 176 247 20
 193 89 88 213 57 139 141 34 57

Syndromes are:
S0 = 14      S1 = 82      S2 = 183      S3 = 179      S4 = 224      S5 = 32      S6 = 148      S7 = 62
S8 = 213      S9 = 118      S10 = 58      S11 = 107      S12 = 112      S13 = 165      S14 = 143      S15 = 190


ERROR POSITIONS OBTAINED FROM CHIEN SEARCH. ACTUAL ERROR VALUES PROVIDED FOR TESTING PURPOSES:
ERROR DETERMINED AT POSITION 44:       82 instead of 30      ERROR VALUE CHECK 76
ERROR DETERMINED AT POSITION 78:       140 instead of 86      ERROR VALUE CHECK 218
ERROR DETERMINED AT POSITION 85:       148 instead of 34      ERROR VALUE CHECK 182
ERROR DETERMINED AT POSITION 106:       219 instead of 132      ERROR VALUE CHECK 95
ERROR DETERMINED AT POSITION 118:       7 instead of 48      ERROR VALUE CHECK 55
ERROR DETERMINED AT POSITION 180:       198 instead of 134      ERROR VALUE CHECK 64
ERROR DETERMINED AT POSITION 195:       250 instead of 123      ERROR VALUE CHECK 129
ERROR DETERMINED AT POSITION 224:       160 instead of 46      ERROR VALUE CHECK 142
____________________________________________

ERROR POSITIONS FROM CHIEN SEARCH. ERROR VALUES FROM FORNEY ALGORITHM:

ERROR POSITION 44            ERROR VALUE 76
ERROR POSITION 78            ERROR VALUE 218
ERROR POSITION 85            ERROR VALUE 182
ERROR POSITION 106           ERROR VALUE 95
ERROR POSITION 118           ERROR VALUE 55
ERROR POSITION 180           ERROR VALUE 64
ERROR POSITION 195           ERROR VALUE 129
ERROR POSITION 224           ERROR VALUE 142
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```