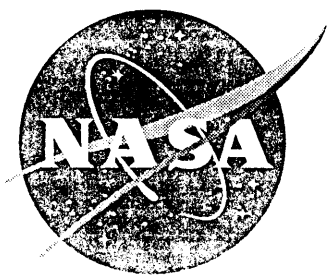
Lyndon B. Johnson Space Center
Houston, Texas 77058

Technical Support Package

Tutorial on Reed-Solomon Error Correction Coding

NASA Tech Briefs
MSC-21834



National Aeronautics and
Space Administration

Technical Support Package

for

TUTORIAL ON REED-SOLOMON ERROR CORRECTION CODING

MSC-21834

NASA Tech Briefs

The information in this Technical Support Package comprises the documentation referenced in **MSC-21834** of *NASA Tech Briefs.* It is provided under the Technology Transfer Program of the National Aeronautics and Space Administration to make available the results of aerospace-related developments considered to have wider technological, scientific, or commercial applications. Further assistance is available from sources listed in *NASA Tech Briefs* on the page entitled "How You Can Benefit From NASA's Technology Utilization Services."

Additional information regarding research and technology in this general area may be found in Scientific and Technical Aerospace Reports (STAR), which is a comprehensive abstracting and indexing journal covering worldwide report literature on the science and technology of space and aeronautics. STAR is available to the public on subscription from

Registration Services
NASA Center for AeroSpace Information
800 Elkridge Landing Road
Linthicum Heights, MD 21090-2934

Telephone: (301) 621-0390, Fax: (301) 621-0134, E-mail: help@sti.nasa.gov

---

Chapter

TABLES

FIGURES

ABSTRACT

This tutorial attempts to provide a frank, step-by-step approach to Reed-Solomon (RS) error correction coding. RS encoding and RS decoding both with and without erasing code symbols will be emphasized. There is no need for this tutorial to present rigorous proofs and extreme mathematical detail. Rather, this tutorial presents the simple concepts of groups and fields, specifically Galois fields, with a minimum of complexity. Before RS codes are presented, other block codes are presented as a technical introduction into coding. A primitive (15,9) RS coding example is then completely developed from start to finish demonstrating the encoding and decoding processes both with and without the soft decision capability. This example includes many, common algorithms necessary to perform RS coding operations. A few other examples are included to further increase understanding. Appendices include RS encoding and decoding hardware design considerations, matrix encoding and decoding calculations, and a derivation of the famous error-locator polynomial. The objective of this tutorial is to present practical information about Reed-Solomon coding in a manner such that people can easily understand it.

INTRODUCTION

What is error correction? The general concept of error correction is restricting the characteristics of source signals in such a manner that sink signals can be processed to reduce noise effects.

What is error correction coding? Error correction coding attaches redundancy, e.g., parity-check symbols, to the data at the system's error correction encoder and uses that redundancy to correct erroneous data at the error correction decoder. In other words, error correction coding is simply restricting the characteristics of the output signals of the system's encoder so that after the signals have been sent to the system's decoder, the decoder will have a very high confidence level of correctly extracting the original source signal for the decoder's corrupted input.

What is the purpose of error correction coding? The purpose of error correction coding might be expressed in a multitude of ways such as (1) increasing the reliability of data communications or data storage over a noisy channel, (2) controlling errors so reliable reproduction of data can be obtained, (3) increasing the overall system's signal-to-noise energy ratio (SNR), (4) reducing noise effects within a system and/or (5) meeting the demand for efficient, reliable, high performance, and economically practical digital data transmission and storage systems. All of these subjective terms can be defined for a particular application.

When we are learning a "new concept" or reviewing a concept that was once understood, we are most often interested in simplicity. In an effort to minimize complexity, this tutorial presents simple examples in clear detail without the need for extensive understanding of complicated mathematics. Once you finish this tutorial, you will have a practical understanding of Reed-Solomon coding.

Some of us are not aware that we all use error correction coding in our daily personal lives. Do you remember times when you really wanted someone to "get the message?" Suppose that you are planning a meeting. You are talking to someone and to be sure that this person heard you indicate the time and place, you repeat the time and place. In order to assure yourself that the person received your exact message, you repeated the same exact message over again. The repetition of the message is a form of error correction encoding; you are adding redundancy to your message. Your intent was to reduce the chance of your listener actually hearing different words than what you were articulating.

Here is another example: You are listening to a soft spoken, articulate speaker in a large auditorium filled with people. You hear this person say, "... Then we all sat up our fellahscopes and viewed Clavius, the largest crater on the near side of our moon and the site of the monolith...." I did not make a typo; you heard "...fellahscopes..." Common sense tells us that this person said "telescopes." How? Well, we performed a decoding operation on our received message. Since there was noise in the room, we did not clearly hear the articulately spoken word "telescopes," but we heard "fellahscopes." The first step in our common sense decoding algorithm is flagging "fellahscopes" as not being a valid word; our language system has redundancy in the sense that there exists invalid words which are never to be used. The second step is to come up with a list of all suspected, valid words which are very close to "fellahscopes." Some solutions are microscope, telescope, oscilloscope, radarscope, and horoscope. We then simply select the closest valid word to "fellahscopes." Since this is an auditory example, "telescopes" sounds closest to "fellahscopes." If this was a textual (or pattern example), then "telescopes" would be closest to "felescopes."

These everyday examples demonstrate what error correction coding is and how it works. Adding error correction capability reduces the chance of decoding some other message than the original message. To add error correction capability, we append redundancy to the message that we want to communicate, and then we transmit (or record) it. Finally, we must be able to receive and decode it.

The Reed-Solomon (RS) codes have been finding widespread applications ever since the 1977 Voyager's deep space communications system. At the time of Voyager's launch, efficient encoders existed, but accurate decoding methods were not even available! The Jet Propulsion Laboratory (JPL) scientists and engineers gambled that by the time Voyager II would reach Uranus in 1986, decoding algorithms and equipment would be both available and perfected. They were correct! Voyager's communications system was able to obtain a data rate of 21,600 bits per second from 2 billion miles away with a received signal energy 100 billion times weaker than a common wrist watch battery!

I want a Dick Tracy audio/video, transmit/receive wristwatch! RS codes have been an integral part of high performance, high productivity, electronic device markets with annual sales expected to top 17 billion American dollars by 1990. RS codes have direct application within many communications markets and nearly all the data storage markets. Some of the more notable markets are the following: In the optical compact disk (CD) markets there are (1) compact disks for high fidelity audio data (i.e., CD players and disks), (2) compact disks for computer data (i.e., CD - read

only memory (CD-ROM) drives and disks), (3) compact disks interactive with a computer to display high fidelity audio, images, and textual data (i.e., CD-I drives and probably disks), (4) compact disks for high fidelity video (and audio) data (i.e., CD-V players and probably disks), (5) compact disks for data which also have write capability for the user (i.e., WORM drives and probably disks where WORM represents write-once, read-many), and (6) compact disks for data which also have multiple write and erasure capabilities for the user (i.e., erasable optical disk drives and probably disks). In the magnetic media markets there are (1) magnetic tape with multiple write and erasure capabilities for computer data storage and/or high fidelity audio (i.e., DAT drives and tapes where DAT stands for digital audio tape) and (2) magnetic disks with multiple write and erasure capabilities for computer data (i.e., hard disk drives and maybe disks). In the communications markets there are (1) communications over the telephone systems with such applications as advanced facsimile machines (which send and receive imaging data) and high speed modems (which usually send and receive computer data), (2) satellite communications with such applications as the Hubble Space Telescope, the Mobile Satellite Terminals, and the 300 megabits per second (Mbps) return link of the Space Station Freedom/Tracking and Data Relay Satellite System (SSF/TDRSS), and (3) deep space communications with such applications as Voyager, the Galileo Orbiter, the Mars Observer, and the Cassini Titan Orbiter/Saturn Probe.

Today, many error correction coding circuits exist and are easily available in different RS coding architectures from different sources. There are also many single chip codec (encoder / decoder) circuits available with and/or without symbol erasure capability. Some of the most powerful and talked about block codes available today are the (255,255-2t) RS codes. There are even single chip codecs available for many of these (255,255-2t) RS codes. An example of a commercially available single integrated circuit codec chip is the $t \leq 10$, $n=255$ configurations of the (n,n-2t) RS codes. These particular single chip codecs can operate in excess of 10 megasymbols per second or rather more than 80 Mbps! For many applications, size, weight, and power considerations of high data rate RS codes are quickly becoming insignificant. Due to the availability, reliability, and performance of today's Reed-Solomon circuits, additional markets, like high definition television (HDTV), should also start to open up.

People are even discovering new, practical uses of Galois fields beyond the error correction (and/or detection), data compression, digital modulation, and cryptography arenas. Some of these arenas are in controls and digital signal processing. For example, not only are there binary and ternary discrete Fourier transforms

(DFTs), but there are also P-ary DFTs where P is a prime number. In today's ever increasing complex and technological world, sometimes the math does not fit into the physical system and sometimes the physical system does not keep up with the math. Sometimes there must be full duplex communications between the coding engineers and the implementation engineers.

The Reed-Solomon error correction codes were introduced by Irving S. Reed and Gustave Solomon in 1960. Their work was independent of other similar works like the work by Bose, Chaudhuri, and Hocquenghem (i.e., the BCH codes). Even though the RS codes are a subgroup of the BCH codes, RS codes have pillaged and burned many of its forbearers and peers in efficiency, practicality, and rates. RS codes have generated many useful and widespread applications. A lot of credit goes to Reed and Solomon.

This tutorial is organized with a conscious effort to present the material in a clear, concise, and simple manner. A universal error correction coding notation semi-exists. I will try to keep the notation as standard and as clear as possible. For a list of the notation used, please refer to the notation section.

This tutorial is organized into five chapters. The material within the chapters and even the chapters themselves are designed to allow skimming if the person already knows the material. Considerable effort has been expended to make each chapter self-contained beside the numerous cross-references linking the entire tutorial together. I try to present the specific definitions as needed and locate them near to the needs. In order to develop an understandable presentation, some specific definitions of terms appear much later within the chapter than the first usage. However, all these design considerations allow all the important details, along with its prerequisite details, to be presented to a beginner in a logical (and condensed!) manner.

One of the best ways to demonstrate how something works is to perform an example from start to finish. Throughout chapters 3,4,5, and appendix C, a primitive (15,9) RS code with some arbitrary inputs will be used as the main example. This particular code was chosen because it has a code rate k/n greater than one half and yet is still powerful enough and small enough to demonstrate. All the encoding and decoding stages will be demonstrated. This demonstration includes working through some of the different algorithms available at each stage obtaining equivalent results. Also, the case of encoding and decoding using symbol erasure (i.e., soft decision) capability will be demonstrated. This example starts out showing all the necessary mathematical rigor, but as this example progresses and similar operations are repeated, only the important results will be shown.

Since all the essential mathematical rigor will be shown at least once, the arithmetic that is not shown is left as exercises for the reader.

In chapter 1 we will learn how to perform Galois field (GF) arithmetic. In the past we have learned algebra (infinite field manipulation), calculus (summation using algebra in a different application), complex arithmetic (two dimensional algebra), Boolean algebra (manipulating binary elements according to rules similar to algebra), and now finally we get to learn GF algebra (finite field algebra using most of the standard elementary algebraic rules). Within this chapter we will derive the GF(16) implementation needed to work our (15,9) RS coding example.

In chapter 2 we will learn about the basics of block codes for "coding" applications. Within this chapter we introduce some terminology, concepts, definitions, structure, and history of error correction codes. For truly complete and absolutely accurate material, we should refer to authoritative texts of which some are in the reference and recommended reading sections. This chapter should provide a general literacy of error correction coding. Hopefully, it reads easily for the beginner and yet is pleasing enough for the experienced person.

In chapter 3 we will learn how to encode Reed-Solomon codes. Here we actually work the (15,9) RS example for the encoding process.

In chapter 4 we will learn how to decode Reed-Solomon codes. Here we actually work the (15,9) RS example for the decoding process.

In chapter 5 we will learn how to design the coding system when we have the symbol erasure capability. Here we work this primitive (15,9) RS example modified to show the power of erasing symbols.

In appendix A we will learn how to encode RS codes using hardware. State tables, equations, and worked out examples help us to understand the encoder's shift register circuit.

In appendix B we will learn how to decode RS codes using hardware. A general discussion of some of the shortcuts and design considerations help us start thinking of how to design a practical decoding system.

In appendix C we will learn how to perform the RS coding operations using matrices. Matrix calculations are probably more familiar to us than finite field polynomial calculations. We can decode using only matrices, but we still face the challenge of determining the estimate of the error.

In appendix D we will learn how to derive the ever popular error-locator polynomial $\sigma(X)$. Here we should see why we often work more with the reciprocal of $\sigma(X)$ [denoted as $\sigma_r(X)$] than $\sigma(X)$.

Some readers prefer details first and completed definitions second. These readers may desire to read appendix D before they read the chapters and the appendices. Appendix D is a very brief general mathematical overview of RS coding.

I want to make Reed-Solomon coding easy to learn. I also want to present enough detail so we may become and stay fairly literate in Reed-Solomon coding. Hopefully this document has enough redundancy in it such that people will receive a good understanding of Reed-Solomon coding. For your information, this tutorial went through two review cycles. Therefore, maybe it is "error free!!" I have seriously tried to reduce the number of errors in technical content, but I am sure some still remain. If anyone would happen to discover any noteworthy errors within this tutorial and would let me know, I will be appreciative.

After you have finished this tutorial, I hope you will feel that this tutorial is helpful and useful to yourself and the people you work with.

I wish to specifically thank the following people for their help in developing this tutorial: Bill Lindsey who served on the second review cycle and gave me an interested, detailed, and technically accurate critique; Phil Hopkins for his help in teaching me the finer points of error correction coding; and Rod Bown for his written and spoken comments which helped me to extensively rewrite the first chapter.

DISCLAIMER: This tutorial is NOT a directive in any form.

CHAPTER 1
GALOIS FIELD ALGEBRA

Galois field (GF) algebra, sometimes referred to as ground field (GF) algebra, is similar to high school algebra or arithmetic except that GF algebra operates within a finite field. Take the case of base ten, integer arithmetic. We can take the element denoted 7, sum with the element denoted 8, and obtain the element 15. If we take some integer and either add, subtract, or multiply it to another integer, we always result with some element in the infinite set. However, in GF algebra it is possible to take the element 7, sum with the element 8, and obtain the resulting element only within a finite number of elements. In GF arithmetic the result of this example is not 7,8, or 0. The result of this example may well be any one of the following elements: 1, 2, 3, 4, 5, 6, 9, A, B, C, D, apples, oranges, ..., the last element in the field. You can not assign all the results of an operation, given all the possible inputs, any way you desire. Algebraic laws will develop the addition and multiplication tables for us.

To learn about Galois field algebra, we must first learn the algebraic laws governing our Galois (or finite) field. These laws are the standard algebraic laws. These laws may have, however, become so familiar to us, that some of us may have even forgotten them! We got so into the habit of only being concerned with the results that we forgot about the underlying algebraic laws which govern the entire system; we just memorized our addition and multiplication tables. Let us first present the basic definitions, theorems, and properties needed to understand GF arithmetic. Most of sections 1.1 and 1.2 are rewritten from Error Control Coding: Fundamentals and Applications by Shu Lin and Daniel J. Costello, Jr. In section 1.3 we use the definitions previously presented in sections 1.1 and 1.2 to derive the ground field GF(2). GF(2) is the ground field of the extended Galois field $GF(2^m)$ that we use in most block error correction codes. In section 1.4 we derive $GF(2^m) = GF(2^4) = GF(16)$ from GF(2). This GF(16) is the field that we are going to use for the (15,9) Reed-Solomon example. Some of the mathematical structure of $GF(2^m)$ is examined. This structure includes some different field element representations and some different field implementations of $GF(2^m)$. This section also includes examples of adding, subtracting, multiplying, and dividing field elements. Then the final section presents all the underlying algebraic structure necessary to create $GF(P^m)$. GF arithmetic is the arithmetic of coding for the RS coding world.

1.1 GROUPS

Let G be a set of elements. A binary operation * on G is a rule that assigns to each pair of elements A and B a uniquely defined third element C = A*B in G. When such a binary operation * is

defined on G, we say that G is closed under *. Also, a binary operation * on G is said to be associative if, for any A, B, and C in G: A*(B*C) = (A*B)*C. Definition 1 defines a group.

DEFINITION 1:
A set G (on which a binary operation is defined) is defined to be a group if the following conditions are satisfied:

a. The binary operation * is associative.

b. G contains an identity element I such that, for any A in G, A*I = I*A = A.

c. For any element A in G, there exists an inverse element A' in G such that A*A' = A'*A = I.

A group G is said to be commutative if its binary operation * also satisfies the following condition:

A*B = B*A, for all A and B in G.

We should also make a note of the following two theorems derived from definition 1: THEOREM 1 is that the identity element I in a group G is unique. THEOREM 2 is that the inverse element A' of a group element is unique.

This information should be all that we need to know about groups to perform GF arithmetic.

1.2 FIELDS

Roughly speaking, a field is a set of elements in which we can do addition, subtraction, multiplication, and division without leaving the set. Addition and multiplication must satisfy the commutative, associative, and distributive laws. Definition 2 defines a field.

DEFINITION 2:
Let F be a set of elements on which two binary operations called addition "+" and multiplication "·" are defined. The set F together with the two binary operations "+" and "·" is a field if the following conditions are satisfied:

a. F is a commutative group under addition "+". The identity element with respect to addition I_{add} is called the zero element or the additive identity I_{add} of F and is denoted by 0 (zero).

b. The set of non-zero elements in F is a commutative group under multiplication "·". The identity element with respect to multiplication I_{mult} is called the unit (or unity) element or the multiplicative identity I_{mult} of F and is denoted by 1 (one).

c. Multiplication "·" is distributive over addition "+"; that is, for any three elements A,B, and C in F: $A \cdot (B+C) = (A \cdot B)+(A \cdot C)$.

It follows from definition 2 that a field consists of at least two elements, the additive identity I_{add} and the multiplicative identity I_{mult}. Soon, we will show that a field of these two elements alone does exist.

The number of elements in a field is called the order of the field. A field with a finite number of elements is called a finite field. In a field, the additive inverse of an element A is denoted by -A, and the multiplicative inverse of A (provided that $A \neq 0$) is denoted by A^{-1}. Subtracting a field element B from another field element A is defined as adding the additive inverse -B of B to A [i.e., A - B is defined as A + (-B)]. If B is a non-zero element, dividing A by B is defined as multiplying A by the multiplicative inverse B^{-1} of B (i.e., A / B is defined as $A \cdot B^{-1} = AB^{-1}$).

We should also make a note of the following five properties of definition 2: PROPERTY 1 is that for every element A in a field, $A \cdot 0 = 0 \cdot A = 0$. PROPERTY 2 is that for any two non-zero elements A and B in a field, $A \cdot B \neq 0$. PROPERTY 3 is that $A \cdot B = 0$ and $A \neq 0$ implies B=0. PROPERTY 4 is that for any two elements A and B in a field, $-(A \cdot B) = (-A) \cdot B = A \cdot (-B)$. PROPERTY 5 is that for $A \neq 0$, $A \cdot B = A \cdot C$ implies B=C.

It is standard practice to either indicate multiplication by its multiplication symbol "·" or by writing the elements adjacent to each other [i.e., $A \cdot B = (A) \cdot (B) = (A)(B) = AB$]. Throughout the rest of this tutorial I will represent multiplication as much as possible by the most common practice of adjacent elements.

We should now know enough about fields to develop one.

1.3 BINARY FIELD GF(2)

At this point, we should have learned enough about groups and fields and reviewed enough of the basic algebraic laws to go ahead and develop a finite field. To demonstrate the idea of finite fields, we start off presenting the simplest case, modulo-2 arithmetic. We will first present the binary group over addition and then over addition and multiplication.

1.3.1 Binary Group

Consider the set of two integers, G={0,1}. Let us define a binary operation, denoted as addition "+", on G as follows:

Modulo-2 addition:

+	0	1
0	0	1
1	1	0

Notice that this can be implemented with a single EXCLUSIVE-OR gate! Anyway, this binary operation is called modulo-2 addition. Let us prove that this is a group G:

Is G closed?
YES.

PROOF:
A+B = C, for all set elements A and B with the result C also being a set element.

0+0 ?=? 0 Yes, and C=0 is also a set element.
0+1 ?=? 1 Yes, and C=1 is also a set element.
1+0 ?=? 1 Yes, and C=1 is also a set element.
1+1 ?=? 0 Yes, and C=0 is also a set element.

Is G associative?
YES.

PROOF:
A+(B+C) = (A+B)+C, for all A, B, and C.

0+(0+0) ?=? (0+0)+0 Yes.
0+(0+1) ?=? (0+0)+1 Yes.
0+(1+0) ?=? (0+1)+0 Yes.
0+(1+1) ?=? (0+1)+1 Yes.
1+(0+0) ?=? (1+0)+0 Yes.
1+(0+1) ?=? (1+0)+1 Yes.
1+(1+0) ?=? (1+1)+0 Yes.
1+(1+1) ?=? (1+1)+1 Yes.

Therefore, definition 1, part a has been verified.

Does G contain an additive identity element I_{add}?
YES, $I_{add}=0$.

PROOF:
$A+I_{add} = I_{add}+A = A$, for all A.

0+0 ?=? 0+0 ?=? 0 Yes.
1+0 ?=? 0+1 ?=? 1 Yes.

Therefore, definition 1, part b has been verified.

Does G contain an additive inverse element A' for each set element A?
YES, the additive inverse element A' for each element A is the set element A itself.

PROOF:
$A+A' = A'+A = I_{add}$, for all A.

0+0 ?=? 0+0 ?=? 0 Yes.
1+1 ?=? 1+1 ?=? 0 Yes.

Therefore, definition 1, part c has been verified. Therefore, we proved that this set {0,1} is a Group G={0,1} under modulo-2 addition.

Is G commutative?
YES.

PROOF:
A+B = B+A, for all A and B.

0+0 ?=? 0+0 Yes.
0+1 ?=? 1+0 Yes.
1+0 ?=? 0+1 Yes.
1+1 ?=? 1+1 Yes.

Therefore, this group G={0,1} is not only a group, but also a commutative group under modulo-2 addition.

1.3.2 Binary Field

Now, since we have modulo-2 addition "+" defined over a binary group, let us develop a binary field. We need to define modulo-2 multiplication "·".

Consider the same set of two integers, F={0,1}. Let us define another binary operation, denoted as multiplication "·", on F as follows:

Modulo-2 multiplication:

·	0	1
0	0	0
1	0	1

Notice that this operation can be implemented with a single AND gate! Anyway, this binary operation is called modulo-2 multiplication. Let us prove that this set F={0,1} is a field under modulo-2 addition and multiplication:

Is F a commutative group under addition?
YES, previously shown in F=G={0,1}.

Is the additive identity element I_{add} of F called the zero element denoted by 0?
YES, previously shown in F=G={0,1}.

Therefore, definition 2, part a has been verified.

Are the non-zero elements a commutative group under multiplication?
YES.

PROOF:
Let G = non-zero elements of F={0,1}; let G={1}.
Is G={1} closed?
YES.

PROOF:
$A \cdot B = C$, for all set elements A and B with the result C also being a set element.

$1 \cdot 1 \stackrel{?}{=} 1$ Yes, and C=1 is also a set element.

Is G={1} associative?
YES.

PROOF:
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$, for all A, B, and C.

$1 \cdot (1 \cdot 1) \stackrel{?}{=} (1 \cdot 1) \cdot 1$ Yes.

Does G={1} contain a multiplicative identity element I_{mult}?
YES, $I_{mult}=1$.

PROOF:
$A \cdot I_{mult} = I_{mult} \cdot A = A$, for all A.

$1 \cdot 1 \stackrel{?}{=} 1 \cdot 1 \stackrel{?}{=} 1$ Yes.

Does G={1} contain an inverse element A' for each element A in the set?
YES, A'=1.

PROOF:
$A \cdot A' = A' \cdot A = I_{mult}$, for all A.

$1 \cdot 1 \stackrel{?}{=} 1 \cdot 1 \stackrel{?}{=} 1$ Yes.

Is G={1} commutative?
YES.

PROOF:
$A \cdot B = B \cdot A$, for all A and B.

$1 \cdot 1 \stackrel{?}{=} 1 \cdot 1$ Yes.

Is the multiplicative identity element I_{mult} of F called the unit element and denoted by 1?
YES, previously shown in F=G={1}.

Therefore, definition 2, part b has been verified.

So far we have shown that G={0,1} is a commutative group under modulo-2 addition AND G={1} is a commutative group under multiplication. We have also shown that the additive identity element I_{add} is denoted by 0 (zero) and that the multiplicative identity element I_{mult} is denoted by 1 (one). To prove that F={0,1} is a field, we now only have to prove that multiplication is distributive over modulo-2 addition.

Is multiplication distributive over modulo-2 addition?
YES.

PROOF:
$A \cdot (B+C) = (A \cdot B)+(A \cdot C)$, for all A, B, and C.

$0 \cdot (0+0)$	?=?	$(0 \cdot 0)+(0 \cdot 0)$	Yes.
$0 \cdot (0+1)$	?=?	$(0 \cdot 0)+(0 \cdot 1)$	Yes.
$0 \cdot (1+0)$	?=?	$(0 \cdot 1)+(0 \cdot 0)$	Yes.
$0 \cdot (1+1)$	?=?	$(0 \cdot 1)+(0 \cdot 1)$	Yes.
$1 \cdot (0+0)$	?=?	$(1 \cdot 0)+(1 \cdot 0)$	Yes.
$1 \cdot (0+1)$	?=?	$(1 \cdot 0)+(1 \cdot 1)$	Yes.
$1 \cdot (1+0)$	?=?	$(1 \cdot 1)+(1 \cdot 0)$	Yes.
$1 \cdot (1+1)$	?=?	$(1 \cdot 1)+(1 \cdot 1)$	Yes.

Therefore, definition 2, part c has been verified.

Therefore, since definition 2 was satisfied, the set {0,1} is a field F={0,1} of two elements under modulo-2 addition and modulo-2 multiplication. Remember, a field F consists of at least two elements: the additive identity I_{add} and the multiplicative identity I_{mult}. This modulo-2 field is the minimum field of finite number of elements that we talked about earlier. This modulo-2 field is usually called a binary or 2-ary field and it is denoted by GF(2). The binary field GF(2) plays a crucial role in error correction coding theory and is widely used in digital data transmission and storage systems.

1.4 EXTENSION FIELDS $GF(2^m)$

Since we now know the underlying algebraic structures to perform GF(2) arithmetic, let us talk about extension fields. We are interested in prime finite fields called Galois fields GF(P). In our previous binary operation example we had the minimum number of possible elements which comprised GF(2). Extension fields are $GF(P^m)$ where m=2,3,4,... With the design of error correction coding based systems, we are interested in binary operations. Therefore, we will mainly speak of binary Galois fields GF(2) and the extended binary Galois fields $GF(2^m)$ from now on.

1.4.1 Primitive Polynomials p(x)

Polynomials over the binary field GF(2) are any polynomials with binary coefficients; they are binary polynomials. Each of these polynomials, denoted as f(X), is simply the product of its irreducible factors, i.e., $f(X) = FACTOR_0 \cdot FACTOR_1 \cdot \ldots \cdot FACTOR_{last}$. We can create an extension field by creating a primitive polynomial p(X). A primitive polynomial p(X) is defined to be an irreducible binary polynomial of degree m which divides X^n+1 for $n = P^m-1 = 2^m-1$ and which does not divide X^i+1 for $i<n$. Once a primitive polynomial p(X) is found, then the elements of the Galois field can

be generated. Any primitive polynomial p(X) can construct the $P^m=2^m$ unique elements including a 0 (zero or null) element and a 1 (one or unity) element. A degree m polynomial f(X) over GF(2) is defined to be irreducible over GF(2) if f(X) is not divisible by any polynomial over GF(2) of degree greater than zero, but less than m. Let us now test to see if the following binary polynomial f(X) is a primitive polynomial p(X). We must show that it is both an irreducible polynomial and also divides X^n+1 appropriately.

First, let us test to see if $f(X) = X^4+X+1$ is irreducible.

$$\begin{aligned} f(0) &= 0^4+0+1 \\ &= (0\cdot0\cdot0\cdot0)+(0+1) = (0\cdot0)\cdot(0\cdot0)+(1) \\ &= (0)\cdot(0)+1 = (0)+1 \\ &= 1 \\ &\neq 0 \end{aligned}$$

Therefore, (X+0)=(X-0) is not a factor (0 is not a root).

$$\begin{aligned} f(1) &= 1^4+1+1 \\ &= (1\cdot1\cdot1\cdot1)+(1+1) \\ &= (1\cdot1)\cdot(1\cdot1)+(0) \\ &= (1)\cdot(1) \\ &= 1 \\ &\neq 0 \end{aligned}$$

Therefore, (X+1)=(X-1) is not a factor (1 is not a root).

Since a factor of degree one does not exist for this degree four f(X), then factors of degree three also do not exist for f(X). This fact is shown as follows:

$$f(X) = X^4+X+1 \neq (X^3+\ldots)(X+\ldots)$$

(X+...) is of degree one and is not a factor. Therefore, if $(X^3+\ldots)$ is irreducible, then it is not possible for $(X^3+\ldots)$ of degree three to be a factor.

Next, we should try to find a factor of degree two.

$$\begin{aligned} X^2 &= X\cdot X \\ &= X\cdot X+0X+0 \\ &= X\cdot X+(0+0)X+(0\cdot0) \\ &= (X+0)(X+0) \\ &= (X+0)^2 \end{aligned}$$

Therefore, X^2 is not a factor because (X+0) is not a factor.

$$\begin{aligned} X^2+1 &= X\cdot X+1 \\ &= X\cdot X+0X+1 \\ &= X\cdot X+(1+1)X+(1\cdot1) \\ &= (X+1)(X+1) \\ &= (X+1)^2 \end{aligned}$$

Therefore, (X^2+1) is not a factor because $(X+1)$ is not a factor.

$$\begin{aligned} X^2+X &= X\cdot X+X \\ &= X\cdot X+1X+0 \\ &= X\cdot X+(0+1)X+(0\cdot 1) \\ &= (X+0)(X+1) \end{aligned}$$

Therefore, (X^2+X) is not a factor because $(X+0)$ and $(X+1)$ are not factors.

Now we need to determine if X^2+X+1 is a factor of $f(X) = X^4+X+1$.

Is X^2+X+1 a factor of $f(X) = X^4+X+1$?

$$\begin{array}{r|l} & \quad\quad\quad\quad\quad\quad 1 \\ X^2+X+1 & X^2+X+X^2+X+1 \\ & X^4\quad\quad\quad\quad\quad +X+1 \\ & \underline{X^4+X^3+X^2} \\ & \quad\quad X^3+X^2+X \\ & \quad\quad \underline{X^3+X^2+X} \\ & \quad\quad\quad\quad\quad\quad 1 \end{array}$$

Remember in GF(2) arithmetic, the additive identity of an element is that element itself. Therefore, subtraction in GF(2) is equivalent to addition in GF(2)!! From the above example, $(X^4) - (X^4+X^3+X^2) = (X^4) + (X^4+X^3+X^2) = X^4+X^4+X^3+X^2 = X^3+X^2$; then bring down the X to form X^3+X^2+X and so on like we usually do division. Most handheld calculators will not help you here!

Since there is a non-zero remainder, X^2+X+1 is not a factor of $f(X)$. Since there are no other possible second degree factors to check, there are no second degree factors which divide $f(x)$.

Since no factors of degree less than f(x) could be found for this binary polynomial $f(X)$, $f(X) = X^4+X+1$ IS IRREDUCIBLE.

Since we have shown that $f(X)$ is irreducible, we must now show that $f(X) = X^4+X+1$ divides $X^n+1 = X^{15}+1$ where $n = P^m-1 = 2^m-1 = 15$ and that $f(X)$ does not divide X^i+1 for $i<n$. This proof will show that this irreducible polynomial $f(X)$ is a primitive polynomial $p(X)$. So let's run through a few iterations. Let us start with X^i+1 of order higher than $f(X)$.

$$\begin{array}{r|l} & \quad\quad\quad\quad X^2+X+1 \\ f(X) = X^4+X+1 & X+X^4+X+1 \\ & X^5+\quad\quad\quad\quad\quad\quad +1 \\ & \underline{X^5+\quad\quad\quad\quad X^2+X} \\ & \quad\quad\quad\quad\quad\quad X^2+X+1 \end{array}$$

The remainder of this division is not zero and therefore $f(X)$ does not divide into X^5+1. So let us try the next higher value for i.

$$f(X) = X^4 + X + 1 \;\Big|\; \begin{array}{l} X^3+X^2+1 \\ X^2 + X^4+X+1 \overline{)\; X^6 + \qquad\qquad\qquad\qquad + 1} \\ \underline{X^6 \qquad\qquad + X^3 + X^2} \\ \qquad\qquad\qquad\quad X^3 + X^2 \qquad + 1 \end{array}$$

Again the remainder of the division is not zero and therefore f(X) does not divide X^6+1. In a like manner f(X) does not divide X^i+1 for the remaining values of i (i=7,8,9,...,14) until i = n = 15 = 2^m-1. Let's show the results of this division.

$$f(X) = X^4 + X + 1 \;\Big|\; \begin{array}{l} X^{11}+X^8+X^7+X^5+X^3+X^2+X+1 \\ X^{15} \qquad\qquad\qquad +1 \\ \underline{X^{15} \qquad\qquad\qquad +1} \\ \qquad\qquad\qquad\quad 0 \end{array}$$

Notice there is a zero remainder; $f(X) = X^4+X+1$ does divide X^n+1. Therefore, since we have shown that this irreducible f(X) divides X^n+1 and not X^i+1 for i<n, THIS IRREDUCIBLE, BINARY POLYNOMIAL f(X) IS ALSO PRIMITIVE; $p(X)=X^4+X+1$.

1.4.2 Field Symbols α^i

Since we have a primitive polynomial $p(X)=X^4+X+1$ of degree m=4, we can now generate our Galois field $GF(P^m) = GF(2^m) = GF(2^4) = GF(16)$ from our field generator polynomial $F(X)=X^4+X+1$; F(X) can simply be any primitive polynomial p(X) of degree m. Since we want to generate the $GF(2^4)=GF(16)$, we need any fourth order p(X).

To construct the field, let us take our field generator polynomial F(X) and perform a recursive process.

Let me first refer back to GF(2). Notice that if we add the unity symbol 1 to the highest symbol in GF(2), which just so happens to be 1 also, we get the lowest symbol 0. It is recursive in the sense that it wrapped around, started back over from its highest symbol to its lowest symbol:

0 = 0	The lowest element.
0+1 = 1	Add the unity element to the lowest element and the result is the element 1.
1+1 = 0	Add the unity element to the previous result and we are back to the lowest element.

Now let's use this interesting fact along with a newly introduced element, alpha $\alpha=\alpha^1$. α^i (read as alpha-to-the-i) will denote each individual element within our GF(16). So in order to develop our field (or alphabet), set "the primitive element $\alpha(X)$", often denoted simply as "α", equivalent to $0X^{m-1} + 0X^{m-2} + \ldots + 1X + 0 = X$. We will complete a recursive multiplication process similar to the previous GF(2) addition example. What we will do is keep taking consecutive powers of "the primitive element alpha"

until the field elements start to repeat.

Because we are using the extension of GF(P)=GF(2), the first P=2 elements of $GF(P^m)=GF(2^m)$ are the same as GF(P)=GF(2); i.e., the null and unity elements in $GF(2^m)$ are the same as the null and unity elements of GF(2). Therefore, $\alpha^{-\infty} = 0 = 0$ and $\alpha^0 = 1 = 1$.

$0 = 0$ ⟶ $0 = 0$
$1 = 1$ ⟶ $1 = 1$

Now we set $\alpha(X) = \alpha = X$ to obtain the 4-tuple $j_3X^3+j_2X^2+j_1X+j_0$ for each element (or symbol) α^i of GF(16):

$\alpha = X$ ⟶ $\alpha = X$
$\alpha^2 = \alpha\cdot\alpha = X\cdot X = X^2$ ⟶ $\alpha^2 = X^2$
$\alpha^3 = \alpha\cdot\alpha^2 = X\cdot X^2 = X^3$ ⟶ $\alpha^3 = X^3$
$\alpha^4 = \alpha\cdot\alpha^3 = X\cdot X^3 = X^4 = ??$ ⟶ $\alpha^4 = ??$

What do we do now to change X^4 into the appropriate m-tuple, i.e., the appropriate 4-tuple? Well, we simply take the modulo function of the result, e.g., $\alpha^4 = \alpha\cdot\alpha^3 = X\cdot X^3 = X^4 = X^4 \bmod F(X)$. One of the ways to perform this modulo function is to set our fourth degree F(X) to zero and obtain the 4-tuple equivalent to X^4. Working this out we obtain

$$F(X) = X^4+X+1 = 0$$
$$X^4 = -X-1 = (-X)+(-1) = (X)+(1) = X+1$$

Therefore, $\alpha^4 = \alpha\cdot\alpha^3 = X\cdot X^3 = X^4 = X^4 \bmod F(X) = X+1$. It should be noted that $\alpha^{-\infty} = 0 = 0 \bmod F(X) = 0$, $\alpha^0 = 1 = 1 \bmod F(X) = 1$, $\alpha^1 = \alpha = X = X \bmod F(X) = X$, $\alpha^2 = X^2 = X^2 \bmod F(X) = X^2$, and $\alpha^3 = X^3 = X^3 \bmod F(X) = X^3$. Let us continue this recursive process by doing a little algebra.

$\alpha^4 = \alpha\cdot\alpha^3 = X\cdot X^3 = X^4 = X^4 \bmod F(X) = X+1$ ⟶ $\alpha^4 = X+1$

$\alpha^5 = \alpha\cdot\alpha^4 = X(X+1) = X^2+X$ ⟶ $\alpha^5 = X^2+X$

$\alpha^6 = \alpha\cdot\alpha^5 = X(X^2+X) = X^3+X^2$ ⟶ $\alpha^6 = X^3+X^2$
or
$\alpha^6 = \alpha^2\cdot\alpha^4 = X^2(X+1) = X^3+X^2$ or $\alpha^6 = X^3+X^2$

$\alpha^7 = \alpha\cdot\alpha^6 = X(X^3+X^2) = X^4+X^3 = (X+1)+X^3$ ⟶ $\alpha^7 = X^3+X+1$
or
$\alpha^7 = \alpha^2\cdot\alpha^5 = X^2(X^2+X) = X^4+X^3$ or $\alpha^7 = X^3+X+1$
or
$\alpha^7 = \alpha^3\cdot\alpha^4 = X^3(X+1) = X^4+X^3$ or $\alpha^7 = X^3+X+1$

In the same manner the following are obtained:

$\alpha^8 = X^2+1$ $\alpha^{11} = X^3+X^2+X$ $\alpha^{14} = X^3+1$
$\alpha^9 = X^3+X$ $\alpha^{12} = X^3+X^2+X+1$
$\alpha^{10} = X^2+X+1$ $\alpha^{13} = X^3+X^2+1$

Notice that the recursive process repeats itself once we create more than the 2^m unique field elements. Let's show this repetition by examples.

$\alpha^{15} = \alpha \cdot \alpha^{14} = X(X^3+1) = X^4+X = (X+1)+X = 1 \quad \longrightarrow \quad \alpha^{15} = \alpha^0 = 1$
$\alpha^{16} = \alpha \cdot \alpha^{15} = X(1) = X \quad \longrightarrow \quad \alpha^{16} = \alpha^1 = X$
$\alpha^{17} = \alpha \cdot \alpha^{16} = X(X) = X^2 \quad \longrightarrow \quad \alpha^{17} = \alpha^2 = X^2$
etc.

This is our finite field made up of 2^m unique symbols generated from F(X) using the primitive symbol alpha $\alpha(X) = X^1 = X$. These unique symbols are labeled as $0, 1, \alpha, \alpha^2, \ldots, \alpha^{n-1}$. It should be noted that sometimes the 0 symbol is denoted by $\alpha^{-\infty}$, 1 by α^0, and α by α^1. The remaining symbols $(\alpha^2, \alpha^3, \ldots, \alpha^{n-1})$ are always denoted the standard way.

Table 1.4.2-1 summarizes the field representations so far.

TABLE 1.4.2-1. - GF(16) ELEMENTS WITH $F(X)=X^4+X+1$ USING $\alpha(X)=X$

GF(16) elements	Power representation	Polynomial representation
0	0	0
1	1	1
α	X	X
α^2	X^2	X^2
α^3	X^3	X^3
α^4	X^4	X+1
α^5	X^5	X^2+X
α^6	X^6	X^3+X^2
α^7	X^7	X^3+X+1
α^8	X^8	X^2+1
α^9	X^9	X^3+X
α^{10}	X^{10}	X^2+X+1
α^{11}	X^{11}	X^3+X^2+X
α^{12}	X^{12}	X^3+X^2+X+1
α^{13}	X^{13}	X^3+X^2+1
α^{14}	X^{14}	X^3+1
$[\alpha^{15}=\alpha^0=1]$	$[X^{15}=X^0=1]$	[1]
$[\alpha^{16}=\alpha^1=\alpha]$	$[X^{16}=X^1=X]$	[X]
$[\alpha^{17}=\alpha^2]$	$[X^{17}=X^2]$	$[X^2]$
[etc.]	[etc.]	[etc.]

The modulo method that we are using to develop the field elements α^i can be performed directly from $\alpha(X)$ mod $F(X) = (i_3X^3+i_2X^2+i_1X+i_0)$ mod $F(X)$. The modulo function is a basic mathematical function; A mod B is simply calculated by dividing A by B with the result being the remainder. The field generator

polynomial is still $F(X) = X^4+X+1$.

$0 = 0 = 0 \bmod F(X) = 0$
$1 = 1 = 1 \bmod F(X) = 1$
$\alpha = X = X \bmod F(X) = X$
$\alpha^2 = (X)^2 = X^2 = X^2 \bmod F(X) = X^2$
$\alpha^3 = (X)^3 = X^3 = X^3 \bmod F(X) = X^3$
$\alpha^4 = (X)^4 = X^4 = X^4 \bmod F(X) = ??$

Calculation of $X^4 \bmod F(X)$:

$$\begin{array}{l|l} F(X) = X^4 + X + 1 & \begin{array}{l} \quad\quad X+1 \\ 1 + \dfrac{}{F(X)} \\ \hline X^4 \\ \underline{X^4 + X + 1} \\ \quad\quad X + 1 \end{array} \end{array}$$

$$\alpha^4 = \overline{\text{REM}}\left[1 + \frac{X+1}{F(X)}\right] = X + 1$$

Therefore,

$\alpha^4 = (X)^4 = X^4 = X^4 \bmod F(X) = X+1$

In the same manner the following were calculated and verified:

$\alpha^5 = X^5 = X^5 \bmod F(X) = X^2+X$
$\alpha^6 = X^6 = X^6 \bmod F(X) = X^3+X^2$
$\alpha^7 = X^7 = X^7 \bmod F(X) = X^3+X+1$
$\alpha^8 = X^8 = X^8 \bmod F(X) = X^2+1$
$\alpha^9 = X^9 = X^9 \bmod F(X) = X^3+X$
$\alpha^{10} = X^{10} = X^{10} \bmod F(X) = X^2+X+1$
$\alpha^{11} = X^{11} = X^{11} \bmod F(X) = X^3+X^2+X$
$\alpha^{12} = X^{12} = X^{12} \bmod F(X) = X^3+X^2+X+1$
$\alpha^{13} = X^{13} = X^{13} \bmod F(X) = X^3+X^2+1$
$\alpha^{14} = X^{14} = X^{14} \bmod F(X) = X^3+1$

$\alpha^{15} = 1$
$\alpha^{16} = X$
etc.

Although the first procedure is easier, we can follow either of these procedures to obtain the same 2^m symbols in the extended Galois field $GF(2^m)$ (see table 1.4.2-1). We should notice that if we followed either procedure too long, i.e., solving for more than 2^m symbols, then we should find that $\alpha^{15} = \alpha^0 = 1$, $\alpha^{16} = \alpha^1 = \alpha$, $\alpha^{17} = \alpha^2$, ..., $\alpha^{(i+jn)} = \alpha^{((i+jn) \bmod n)} = \alpha^i$ where j is an integer and $n=2^m-1$. In other words, continuing this procedure for more than 2^m unique symbols will only result in repeating the polynomial representation of the symbols.

Thus, we finished developing the field of 2^m unique symbols in $GF(2^m)$. THE FIELD GENERATOR $F(X)=X^4+X+1$ AND THE PRIMITIVE ELEMENT $\alpha(X)=X$ WILL BE USED THROUGHOUT THE REMAINING CHAPTERS.

1.4.3 Different Symbol Representations

The Galois field elements (or symbols) can probably be represented in hundreds of useful and effective ways. Usually the vector representation is the cleanest and the easiest to perform additive calculations on. The vectors are simply constructed with the null character 0 representing the absence of the X^j at a certain j=0,1, 2,...,m-1; i.e., the GF(2^m) field elements $\alpha^i = j_3X^3+j_2X^2+j_1X+j_0$. The unity character 1 represents the presence of the X^j. It does not matter which direction you choose to write the vectors as long as you are consistent. For example, suppose $F(X)=X^4+X+1$ and a primitive element X is given: Mathematicians usually prefer writing $\alpha^6 = X^2+X^3$ = (0011) while application engineers usually prefer $\alpha^6 = X^3+X^2$ = (1100). In this tutorial, I will always be consistent in writing the representations the way I usually do it: $\alpha^6 = X^3+X^2$ = (1100).

TABLE 1.4.3-1. - EQUIVALENT ELEMENT REPRESENTATIONS

GF(16) symbols	Polynomial representation	Vector (or m-tuple) representation
0	0	(0000)
1	1	(0001)
α	α	(0010)
α^2	α^2	(0100)
α^3	α^3	(1000)
α^4	$\alpha+1$	(0011)
α^5	$\alpha^2+\alpha$	(0110)
α^6	$\alpha^3+\alpha^2$	(1100)
α^7	$\alpha^3+\alpha+1$	(1011)
α^8	α^2+1	(0101)
α^9	$\alpha^3+\alpha$	(1010)
α^{10}	$\alpha^2+\alpha+1$	(0111)
α^{11}	$\alpha^3+\alpha^2+\alpha$	(1110)
α^{12}	$\alpha^3+\alpha^2+\alpha+1$	(1111)
α^{13}	$\alpha^3+\alpha^2+1$	(1101)
α^{14}	α^3+1	(1001)
$[\alpha^{15}=\alpha^0=1]$	[1]	[(0001)]
$[\alpha^{16}=\alpha^1=\alpha]$	[α]	[(0010)]
$[\alpha^{17}=\alpha^2$]	[α^2]	[(01000]
[etc.]	[etc.]	[(etc.)]

So far I have presented three equivalent ways to represent a finite field symbol. These ways are shown in table 1.4.3-1. Compare table 1.4.2-1 with table 1.4.3-1. Since we chose the special case of setting the primitive element $\alpha(X)$ equivalent to X to generate the field, we often will represent the field elements α^i in terms

of the α^j instead of the X^j. We will denote the elements α^i as the symbols of GF(2^m). These are common practices and they help to simplify some of the notation in the following chapters; compare table 1.4.3-1 with table 1.4.2-1. Because there are so many possible representations for the one and only one GF(2^m) for each m, many people prefer to simply denote the field elements as symbols. A dictionary can define a symbol as an arbitrary or conventional sign used in writing or printing which relates to a particular field to represent operations, quantities, elements, relations, or qualities; in other words, symbols are a part of the notation used to represent the elements within our GF(16).

Similarly, sometimes polynomials are written in row matrix form as a shorthand form. For example, if $p(X)=1+X+X^4$, then $p(X)=[11001]$. Again, I will remain with the notation such that $p(X) = X^4+X+1 = [10011]$.

The exponent (or power) and vector (or m-tuple) representations are the most popular. Multiplication by hand is easily performed using the power representation and addition using the vector representation. However, they are all equivalent representations.

The cyclic (shift register) nature of the elements in GF(2^m) is interesting. Notice that α^5 is α^4 with one left shift and α^6 is either α^4 with two left shifts or α^5 with one; e.g., α^5 = arithmetic shift left of (0011) = (0110). Since the most significant binary-tuple of α^6 = (1100) is a "1", α^7 is α^4 plus α^6 shifted left; $\alpha^7 = \alpha^4$ + arithmetic shift left of α^6 = (0011) + (1000) = (1011). For details of how to work with shift register circuits (SRC), please refer to a text or later refer to appendices A and B.

Most people say there is one and only one primitive element to generate the one and only one GF(2^m) for each m. They are correct; there is one and only one $\alpha(X)$, but $\alpha(X)$ might be X, or X^2, or X+1, etc. TABLE 1.4.3-1 WILL BE USED THROUGHOUT THE FOLLOWING CHAPTERS AS THE GF(16) NEEDED TO WORK OUR PRIMITIVE RS (15,9) EXAMPLE.

The standard way of generating the field elements α^i is by using $\alpha(X) = \alpha = X$ as demonstrated in this section. In chapter 3 we will discuss RS encoding and we will need to be aware that "other" primitive elements exist other than $\alpha = \alpha(X) = X$. Section 1.4.4 demonstrates there are other primitive elements besides $\alpha(X)=X$. It also indicates that the field elements α^i can be generated using these other primitive elements which may be helpful in some implementations. If one does not care to read the next section, then note the comments of this paragraph and skip over to section 1.4.5.

1.4.4 Isomorphic GF(2^m) Implementations

There is one and only one GF(2). There is one and only one GF(2^4)=GF(16). In fact, there is one and only one

finite field GF(2^m) for each m. However, not only are there many different representations for each finite field element (e.g., m-tuple representation or power representation), but there are also many ways to implement the elements of GF(2^m).

The earlier sections presented the most popular, most common, and probably the easiest way of explaining the field generation. This was done by setting the primitive element alpha $\alpha(X)=\alpha$ to the polynomial $i_3X^3+i_2X^2+i_1X+i_0 = X$; X is a primitive element of GF(16) using $F(X)=X^4+X+1$. By generating the field, i.e., generating the $P^m=2^m$ unique symbols, the process assigned particular patterns of 1's and 0's (see the vector representation) to each of the GF(16) symbols; the process generated one particular implementation of the one and only GF(16). Now in this section, I want to communicate that different implementations are often preferred when we try to apply the field into physical systems and/or into computational systems.

All the possible primitive elements of the one and only one GF(16) using $F(X)=X^4+X+1$ are X, X^2, X+1, X^3+X+1, X^2+1, X^3+X^2+X, X^3+X^2+1, and X^3+1. When a primitive polynomial is used as the field generator, primitive elements are the prime (or relatively prime) powers of the primitive element $\alpha(X)=X$ to one less than the size of the field. In other words, refer to table 1.4.2-1 and notice that α^3, α^5, α^6, α^9, α^{10}, and α^{12} are not primitive elements because 3,5,6,9,10, and 12 are not relatively prime to $q-1 = 2^m-1 = 15 = 3\cdot5$; $\alpha(X)=X$, $\alpha(X)=X^2$, $\alpha(X)=X+1$, $\alpha(X)=X^3+X+1$, $\alpha(X)=X^2+1$, $\alpha(X)=X^3+X^2+X$, $\alpha(X)=X^3+X^2+1$, and $\alpha(X)=X^3+1$ are all primitive elements because 2,4,7,8,11,13, and 14 (from α^2, α^4, α^7, α^8, α^{11}, α^{13}, and α^{14} of table 1.4.2-1) are relatively prime to $q-1 = 15 = 3\cdot5$. It should be noted that all the non-zero, non-unity elements of the GF(4), the GF(8), the GF(32), the GF(128), and some of the other higher degree GF(2^m)'s are primitive elements because the (q-1)'s are primitive, i.e., 3,7,31,127, etc. are prime numbers.

Now, let me work an example of a GF(16) implementation different than what is shown in table 1.4.2-1 (and table 1.4.3-1). For learning purposes, let us use the same F(X) as used to generate our field in table 1.4.2-1, but this time let us use $\alpha(X) = i_3X^3+i_2X^2+i_1X+i_0 = X^2$ instead of $\alpha(X) = i_3X^3+i_2X^2+i_1X+i_0 = X$. All right, set $\alpha(X) = X^2$ and develop an implementation different than when $\alpha(X) = X^1 = X$.

$0 = 0$
$1 = 1$

Now we set $\alpha(X)=X^2$ to obtain the 4-tuple $j_3X^3+j_2X^2+j_1X+j_0$:

$$\begin{aligned}\alpha &= X^2\\ &= X^2 \bmod F(X)\\ &= X^2\end{aligned}$$

$$\alpha^2 = \alpha \cdot \alpha = X^2 \cdot X^2$$
$$= X^4 \bmod F(X)$$
$$= X+1$$

$$\alpha^3 = \alpha \cdot \alpha^2 = X^2(X+1)$$
$$= (X^3+X^2) \bmod F(X)$$
$$= (X^3 \bmod F(X)) + (X^2 \bmod F(X))$$
$$= X^3+X^2$$

$$\alpha^4 = \alpha \cdot \alpha^3 = X^2(X^3+X^2)$$
$$= (X^5+X^4) \bmod F(X)$$
$$= (X^5 \bmod F(X)) + (X^4 \bmod F(X))$$
$$= (X^2+X) + (X+1)$$
$$= X^2+1$$

$$\alpha^5 = \alpha \cdot \alpha^4 = X^2(X^2+1)$$
$$= (X^4+X^2) \bmod F(X)$$
$$= (X^4 \bmod F(X)) + (X^2 \bmod F(X))$$
$$= (X+1) + (X^2)$$
$$= X^2+X+1$$

$$\alpha^6 = \alpha \cdot \alpha^5 = X^2(X^2+X+1)$$
$$= (X^4+X^3+X^2) \bmod F(X)$$
$$= (X^4 \bmod F(X)) + (X^3 \bmod F(X)) + (X^2 \bmod F(X))$$
$$= (X+1) + (X^3) + (X^2)$$
$$= X^3+X^2+X+1$$

$$\alpha^7 = \alpha \cdot \alpha^6 = X^2(X^3+X^2+X+1)$$
$$= (X^5+X^4+X^3+X^2) \bmod F(X)$$
$$= X^5 \bmod F(X) + X^4 \bmod F(X) + X^3 \bmod F(X) + X^2 \bmod F(X)$$
$$= (X^2+X) + (X+1) + (X^3) + (X^2)$$
$$= X^3+1$$

We can start to get an intuitive feeling that even though there are many implementations playing around with the structure of the field, there is one and only one GF(2^m) for each m. Completing the procedure for the remaining elements, we get the following remaining implementation:

$\alpha^8 = X$
$\alpha^9 = X^3$
$\alpha^{10} = X^2+X$
$\alpha^{11} = X^3+X+1$
$\alpha^{12} = X^3+X$
$\alpha^{13} = X^3+X^2+X$
$\alpha^{14} = X^3+X^2+1$
$\alpha^{15} = 1$

These results are listed in table 1.4.4-1. Notice that the field elements α^i within table 1.4.4-1 have a different implementation (or representation) than the field elements α^i within table 1.4.2-1. Even though there are many possible implementations of GF(16), mathematically there is one and only one GF(16). We should notice that when we do NOT use $\alpha(X)=X$ as our primitive

element, we will not develop a table similar to table 1.4.3-1, i.e., we will NOT develop $\alpha^i = j_{m-1}\alpha^{m-1} + j_{m-2}\alpha^{m-2} + \ldots + j_1\alpha + j_0$ representations, but we can develop $\alpha^i = j_{m-1}X^{m-1} + j_{m-2}X^{m-2} + \ldots + j_1X + j_0$ representations.

TABLE 1.4.4-1. - GF(16) ELEMENTS WITH $F(X)=X^4+X+1$ USING $\alpha(X)=X^2$

GF(16) symbols	Polynomial representation	Vector (or m-tuple) representation
0	0	(0000)
1	1	(0001)
α	X^2	(0100)
α^2	$X+1$	(0011)
α^3	X^3+X^2	(1100)
α^4	X^2+1	(0101)
α^5	X^2+X+1	(0111)
α^6	X^3+X^2+X+1	(1111)
α^7	X^3+1	(1001)
α^8	X	(0010)
α^9	X^3	(1000)
α^{10}	X^2+X	(0110)
α^{11}	X^3+X+1	(1011)
α^{12}	X^3+X	(1010)
α^{13}	X^3+X^2+X	(1110)
α^{14}	X^3+X^2+1	(1101)
$[\alpha^{15}=\alpha^0=1]$	[1]	[(0001)]
$[\alpha^{16}=\alpha^1=\alpha]$	[X^2]	[(0100)]
$[\alpha^{17}=\alpha^2]$	[$X+1$]	[(0011)]
[etc.]	[etc.]	[(etc.)]

Not only do we have different primitive elements to cause isomorphic implementations, but we also have a minimum of two primitive polynomials for any $GF(2^m)$, i.e., a primitive polynomial $p(X)$ and it's reciprocal $p_r(X)$ where $p_r(X) = X^m p(X^{-1})$.

We keep talking about primitive polynomials, but did you know that we do not even need to use a $p(X)$ to generate the $GF(2^m)$? For example, we could generate the one and only GF(16) using the irreducible, but non-primitive polynomial $F(X) = X^4+X^3+X^2+X+1$ and a primitive element $\alpha(X) = X+1$. However, one of the reasons we usually use primitive polynomials is that $\alpha(X)=X$ will always be a primitive element of any primitive polynomial $p(X)$.

Some implementations consist of generating the field using one of the previous implementations and then biasing the elements, e.g., generate the field and then to obtain the implementation of how the 1's and 0's are assigned to each element, set $\alpha_{new}^{i} = \alpha^{j}\alpha_{old}^{i}$ where $i = -\infty,0,1,2,\ldots,2^{m}-2$ and j is an integer.

Besides all these implementations there are many more. We used a polynomial base, i.e., consecutive powers of the primitive element. There are other possible bases which are useful in computations and/or physical system implementations.

It should be noted that most people prefer to just say there are many different representations for each unique $GF(2^m)$ for each m. Simply try to use a standard representation which makes the most sense to you, but remember your system implementation.

Overall, there is one and only one $GF(2^m)$ for each m. There are many implementations for each $GF(2^m)$. Some implementations are easier to understand, some are more useful in computational implementations, while some are more useful in physical system implementations. FROM THIS POINT ON, REFER ONLY TO THE MOST COMMON IMPLEMENTATION OF GF(16) FOUND IN TABLE 1.4.3-1.

1.4.5 Addition and Subtraction Within $GF(2^m)$

Addition in the extended Galois field $GF(2^m)$ can be performed by one of two methods. The most common method is by exclusive-oring the elements' vector representations position by position. This is simply performing modulo-2 addition; we are not using carry arithmetic. The least common method is by adding their polynomial representations together. It is interesting to realize that these two methods are equivalent! For example $\alpha^8+\alpha^5 = (\alpha^2+1)+(\alpha^2+\alpha) = \alpha+1 = \alpha^4$ is equivalent to α^8 XOR α^5 = (0101) XOR (0110) = (0011) = α^4. Remember that subtraction and addition are equivalent in $GF(2^m)$ arithmetic (i.e., $\alpha^8+\alpha^5 = (\alpha^2+1)+(\alpha^2+\alpha) = (\alpha^2+\alpha^2)+\alpha+1 = (0)+\alpha+1 = \alpha+1 = \alpha^4$ is equivalent to α^8 XOR α^5 = (0101) XOR (0110) = (<0 XOR 0><1 XOR 1><0 XOR 1><1 XOR 0>) = (<0+0><1+1><0+1><1+0>) = (0011) = α^4).

Using the vector addition method:

$$\begin{array}{r} \alpha^4 = 0011 \\ \underline{\alpha^8 = 0101} \\ \alpha^4 \text{ XOR } \alpha^8 = 0110 = \alpha^5 \end{array}$$

Using the polynomial addition method:

$\alpha^4 + \alpha^8 = (\alpha+1) + (\alpha^2+1)$
$= \alpha + \alpha^2$
$= \alpha^5$

$\alpha^8 + \alpha^4 = (\alpha^2+1) + (\alpha+1)$
$= \alpha^2 + \alpha$
$= \alpha^5$

Since subtraction is identical to addition:

$\alpha^4 + \alpha^8 = \alpha^4 - \alpha^8$
$= -\alpha^4 + \alpha^8$
$= -\alpha^4 - \alpha^8$

$\alpha^8 + \alpha^4 = \alpha^8 - \alpha^4$
$= -\alpha^8 + \alpha^4$
$= -\alpha^8 - \alpha^4$

Therefore, $\alpha^8 + \alpha^5 = \alpha^5 + \alpha^8 = \alpha^4$
$\alpha^8 + \alpha^4 = \alpha^4 + \alpha^8 = \alpha^5$
$\alpha^5 + \alpha^4 = \alpha^4 + \alpha^5 = \alpha^8$

TABLE 1.4.5-1. - ADDITION/SUBTRACTION TABLE USED IN GF(2^4)

0	1	α	α^2	α^3	α^4	α^5	α^6	α^7	α^8	α^9	α^{10}	α^{11}	α^{12}	α^{13}	α^{14}
1	0	α^4	α^8	α^{14}	α	α^{10}	α^{13}	α^9	α^2	α^7	α^5	α^{12}	α^{11}	α^6	α^3
α		0	α^5	α^9	1	α^2	α^{11}	α^{14}	α^{10}	α^3	α^8	α^6	α^{13}	α^{12}	α^7
α^2			0	α^6	α^{10}	α	α^3	α^{12}	1	α^{11}	α^4	α^9	α^7	α^{14}	α^{13}
α^3				0	α^7	α^{11}	α^2	α^4	α^{13}	α	α^{12}	α^5	α^{10}	α^8	1
α^4					0	α^8	α^{12}	α^3	α^5	α^{14}	α^2	α^{13}	α^6	α^{11}	α^9
α^5						0	α^9	α^{13}	α^4	α^6	1	α^3	α^{14}	α^7	α^{12}
α^6							0	α^{10}	α^{14}	α^5	α^7	α	α^4	1	α^8
α^7								0	α^{11}	1	α^6	α^8	α^2	α^5	α
α^8									0	α^{12}	α	α^7	α^9	α^3	α^6
α^9										0	α^{13}	α^2	α^8	α^{10}	α^4
α^{10}											0	α^{14}	α^3	α^9	α^{11}
α^{11}												0	1	α^4	α^{10}
α^{12}													0	α	α^5
α^{13}														0	α^2
α^{14}															0

To save time throughout the remainder of this tutorial, addition/subtraction tables have been developed. Since $\alpha^i+\alpha^j = \alpha^j+\alpha^i$, only half of the following tables have been filled in an effort to be easier on the eye and thus speed the calculations. I suggest that either table 1.4.5-1 or its more condensed version, table 1.4.5-2, be copied and used as a bookmark. Table 1.4.5-2 is condensed from table 1.4.5-2 by denoting 0 as $-\infty$, 1 as 0, α as 1, α^2 as 2, ..., α^{14} as 14.

TABLE 1.4.5-2. - GF(16) ADDITION/SUBTRACTION TABLE USED

-∞	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	-∞	4	8	14	1	10	13	9	2	7	5	12	11	6	3
1		-∞	5	9	0	2	11	14	10	3	8	6	13	12	7
2			-∞	6	10	1	3	12	0	11	4	9	7	14	13
3				-∞	7	11	2	4	13	1	12	5	10	8	0
4					-∞	8	12	3	5	14	2	13	6	11	9
5						-∞	9	13	4	6	0	3	14	7	12
6							-∞	10	14	5	7	1	4	0	8
7								-∞	11	0	6	8	2	5	1
8									-∞	12	1	7	9	3	6
9										-∞	13	2	8	10	4
10											-∞	14	3	9	11
11												-∞	0	4	10
12													-∞	1	5
13														-∞	2
14															-∞

1.4.6 Multiplication and Division Within GF(2^m)

As in the case of addition and subtraction in GF(2^m), we also have two methods to perform multiplication and division. The most common method is by summing the symbols' exponents modulo 2^m-1 (or modulo n) and the least common method is again the polynomial method.

Using the exponent mod n multiplication method:

$$\alpha^5\alpha^2 = \alpha^{5+2}$$
$$= \alpha^7$$

Again using the exponent mod n multiplication method:

$$\alpha^5\alpha^{14} = \alpha^{5+14}$$
$$= \alpha^{19}$$
$$= \alpha^{19 \bmod 15}$$
$$= \alpha^4$$

Another method of performing the modulo function for multiplication or division is to keep multiplying or dividing by α^{15}, which is unity, until we obtain a symbol within the finite field.

$$\alpha^5\alpha^{14} = \alpha^{5+14}$$
$$= \alpha^{19}$$
$$= \alpha^{19 \bmod 15}$$
$$= \alpha^{19}/\alpha^{15} \qquad \text{for } \alpha^{15} = \alpha^{-15} = \alpha^0 = 1$$
$$= \alpha^4$$

Using the exponent mod n division method:

$$\begin{aligned}\alpha^5/\alpha^2 &= \alpha^5\alpha^{-2}\\ &= \alpha^{5+(-2)}\\ &= \alpha^3\end{aligned}$$

Using the exponent mod n division method for an inverse symbol α^{-i}:

$$\begin{aligned}\alpha^5/\alpha^{14} &= \alpha^5\alpha^{-14}\\ &= \alpha^{5+(-14)}\\ &= \alpha^{-9}\\ &= \alpha^{-9 \bmod 15}\\ &= \alpha^6\end{aligned}$$

Or we can again use the multiply or divide by unity method:

$$\begin{aligned}\alpha^5/\alpha^{14} &= \alpha^5\alpha^{-14}\\ &= \alpha^{5+(-14)}\\ &= \alpha^{-9}\\ &= \alpha^{-9 \bmod 15}\\ &= \alpha^{-9}\alpha^{15} \qquad \text{for } \alpha^{15} = \alpha^{-15} = \alpha^0 = 1\\ &= \alpha^6\end{aligned}$$

Using the polynomial multiplication method:

$$\begin{aligned}\alpha^5\alpha^2 &= (\alpha^2+\alpha)\alpha^2\\ &= \alpha^2\alpha^2+\alpha^1\alpha^2\\ &= \alpha^{(2+2)}+\alpha^{(1+2)}\\ &= \alpha^4+\alpha^3\\ &= (\alpha+1)+\alpha^3\\ &= \alpha^3+\alpha+1\\ &= \alpha^7\end{aligned}$$

Another example using the polynomial multiplication method:

$$\begin{aligned}\alpha^5\alpha^{14} &= (\alpha^2+\alpha)(\alpha^3+1)\\ &= \alpha^2\alpha^3+\alpha^2+\alpha^1\alpha^3+\alpha\\ &= \alpha^5+\alpha^2+\alpha^4+\alpha\\ &= (\alpha^2+\alpha)+\alpha^2+(\alpha+1)+\alpha\\ &= \alpha+1\\ &= \alpha^4\end{aligned}$$

Using the polynomial division method:

$$\begin{aligned}
\alpha^5/\alpha^2 &= \alpha^5\alpha^{-2} \\
&= \alpha^5(\alpha^{-2 \bmod 15}) \\
&= \alpha^5\alpha^{13} \\
&= (\alpha^2+\alpha)(\alpha^3+\alpha^2+1) \\
&= \alpha^2\alpha^3+\alpha^2\alpha^2+\alpha^2+\alpha^1\alpha^3+\alpha^1\alpha^2+\alpha \\
&= \alpha^5+\alpha^4+\alpha^2+\alpha^4+\alpha^3+\alpha \\
&= \alpha^5+\alpha^3+\alpha^2+\alpha \\
&= (\alpha^2+\alpha)+\alpha^3+\alpha^2+\alpha \\
&= \alpha^3
\end{aligned}$$

Again using the polynomial division method:

$$\begin{aligned}
\alpha^5/\alpha^{14} &= \alpha^5\alpha^{-14} \\
&= \alpha^5(\alpha^{-14 \bmod 15}) \\
&= \alpha^5\alpha \\
&= (\alpha^2+\alpha)\alpha \\
&= \alpha^2\alpha+\alpha^1\alpha^1 \\
&= \alpha^3+\alpha^2 \\
&= \alpha^6
\end{aligned}$$

Multiplication is easily performed by adding the exponents modulo n and noting that $\alpha^i \cdot \alpha^{-\infty} = (\alpha^i)(0) = 0$. A multiplication table is left as an exercise.

1.5 DIFFERENT ALGEBRAIC STRUCTURES

In Reed-Solomon coding we are only interested in Galois field algebra. However, it is interesting to understand the overall picture of structure. Table 1.5-1 summarizes the relationships of all of the different algebraic structures constructed in order for us to be able to construct the extension Galois field $GF(2^m)$. Notice that a semigroup is a subset of a monoid which is a subset of a group which is a subset of a communicative (or abelian) group and on and on up to a Galois (or finite) field being a subset of an extension field. Table 1.5-1 is edited from JPL publication 77-23, "Review of Finite Fields: Applications to Discrete Fourier Transforms and Reed-Solomon Coding", by Wong, Truong, Benjauthirt, Mulhall, and Reed.

TABLE 1.5-1. - RELATIONSHIPS BETWEEN ALGEBRAIC STRUCTURES

Algebraic structure	Properties
Semigroup	One operation, say addition "+", closed and associative
Monoid	Also with an additive identity element I_{add}
Group	Also with an additive inverse element -A
Commutative or abelian group	Also commutative for addition "+"
Ring	Also with another operation, say multiplication "·", closed and associative. Also addition "+" and multiplication "·" are distributive. Note: A ring is a commutative group under addition "+" and a semigroup under multiplication "·".
Commutative ring	Also commutative for multiplication "·"
Commutative ring with unity element	Also with the unity element 1 (one) for with addition "+" and multiplication "·" Note: A commutative ring with unity element is a commutative group under addition "+" and a monoid under multiplication "·".
Field	Also every non-zero element has a multiplicative inverse A^{-1} and that $AA^{-1}=1$, where 1 is the identity (or unity) element I_{mult} for multiplication "·". Note: A field is a commutative group under addition "+" and its non-zero elements form a multiplicative group.
Finite field or Galois field	Also with finite number of elements
Extension field	Also with the only possible finite fields $GF(P^m)$ where $GF(P^m)$ is the extension field of GF(P), GF(P) is the finite field (or Galois field or ground field), P is prime, and m is an integer

1.6 SUMMARY

This chapter should have gone into enough detail to answer most if not all questions about Galois field algebra. It did present enough material to be able to thoroughly proceed and work the following (15,9) RS coding example. Hopefully, this chapter answered all the questions from those who are being introduced into finite field algebra and coding for the first time. The ones who are still interested in further study of coding mathematics would be served by reading the coding bibles or referring to some other authoritative text.

Now we should be ready to perform operations on block codes, especially non-binary BCH, cyclic, and linear block codes known as Reed-Solomon codes.

CHAPTER 2
BLOCK CODES

Before we talk Reed-Solomon (RS), it is best to first talk about its great, great grandparents called block codes. In this chapter we start with a general block error correction coding system with few specifics. Then in section 2.2, we construct a little perfect (3,1) block code introducing terminology, concepts, and some definitions. We then proceed into the next section where some of the codes are defined and some of the ancestry is presented. In section 2.4 we combine a random error correcting code with a burst error correcting code. Also, burst error correction improvement, block code modification, and synchronization are briefly discussed. Section 2.5 discusses the error correction and detection domains using the analogy of zones.

2.1 BLOCK ERROR CORRECTION CODING SYSTEM

In general, coding is taking k symbols as input to an encoder producing n output symbols. These symbols are transmitted over a channel and the result input into a decoder. The output of the decoder is usually the decoded version of the original k symbols. In general, n in respect to k can be >,=,<, and/or any function of those. When n>k, we may have a system with some n-k additional symbols. An application of this can be to add parity-check, a form of redundancy, to the original data for error correction and/or detection applications. When n=k we may have a scrambling application. When n<k, we may have a compression application. In the RS world the word "coding" means "coding for the application of increased communication reliability through error correction capability with n>k."

We should also note that in coding we are not interested in the "meaning" of the message in the sense that we can do something earth shattering with these data. Rather, we are interested in the sense that we can replicate at the output of the decoder what was input to the encoder. Some data in, some corresponding data out. Some particular garbage in, some corresponding garbage out.

In general, a block error correction encoding system is simply a mapping of elements in an ordered set (denoted as a k-tuple) into a unique, ordered set with more elements (denoted as a n-tuple); the encoding process annexes redundancy to the message. The idea behind a block, error correction decoding system is simply a mapping of the received n-tuple into its nearest, valid n-tuple

(which corresponds to a unique k-tuple); the decoding process removes the redundancy to recover the original message. If the received n-tuple is correctly mapped into the original, encoded n-tuple, then the decoded k-tuple is guaranteed to be the original k-tuple. The procedure is that (1) the k-tuple is mapped into the n-tuple, (2) the n-tuple is transmitted (or recorded), (3) the n'-tuple (which is the n-tuple added with the channel error induced by some type of noise) is received (or played back), and (4) the k-tuple is then hopefully decoded from the n'-tuple by a mapping algorithm.

In the encoding process for a systematic code, the k-tuple is mapped into the n-tuple by taking the k-tuple's symbols (synonymous to elements) and usually appending additional symbols for the purpose of error correction and/or detection. For a cyclic code the additional symbols which are appended to the k-tuple are generated by taking the location shifted k-tuple modulo the generator.

If the error correcting capability of the code is not exceeded, then the decoder is guaranteed to correctly decode the n'-tuple into the k-tuple. In other words, in a noisy communication channel it is SOMETIMES possible to correct ALL the errors which occurred! We can sometimes guarantee that the decoder's output will be EXACTLY what was transmitted (or recorded)!! If the error correction capability is exceeded, then the decoder will usually do one of two things; it will either detect that the error correction capability was exceeded or it will decode into an incorrect set. If the decoder decoded into an incorrect set, then a decoder error results. If this decoder error cannot be detected, then it is an undetectable decoder error. If the error correction capability was sensed as exceeded, then the decoder might be designed to send an automatic repeat request (ARQ) signal and/or pass the noisy n-tuple, denoted n'-tuple, through the system. Also, if the code is systematic, we can at least recover the noisy message, denoted k'-tuple, from the n'-tuple. For many applications, passing the n'-tuple through the system is not desirable, e.g., possible privacy concerns.

2.2 A PERFECT (3,1) BLOCK CODE

Let me explain a block coding system in another way. Let us use an example. Assume that we want to transmit either an "on" message or an "off" message. This can be realized in transmitting a binary symbol (or a digital bit; be aware that many people define "bits" as a measurement unit of "information"). A binary symbol with a "1" is used to indicate the "on" and a "0" to indicate the "off".

We send either 1 or 0 out over our communication channel and hopefully receive at our destination the same message that we had sent. However, sometimes we don't receive the same message that we sent because channel noise and even other noise was able to infiltrate our system and inject errors into our message. This is where error correction coding might come in.

Error correction coding provides us control of these errors so we as communication or as data storage engineers can obtain "reliable transmission (or storage) of data." Claude Shannon in 1948 demonstrated that not only by "proper" modulation and demodulation of information, but also by "proper" encoding and decoding of information, any arbitrary high, but non-unity, probability of decoding the received block of data (the units are symbols) into our original information (the units are bits) can theoretically be realized. The real problem is to approach Shannon's limit by designing algorithms and then applying these algorithms and theory into practical systems.

We don't get something for nothing. We must either decrease our information rate and/or decrease the energy associated per transmitted symbol and/or increase our power and/or increase our bandwidth. Often such considerations as antenna power, bandwidth, and the modulation technique used are already cast into concrete. So the communication system with error correction capability is often designed at the expense of reducing our information rate and adding a little power to operate the additional hardware. However, often this power is negligible compared to other alternatives such as increasing the antenna power. Also, notice that coding does require a serial insertion into the communications channel and thus will add to the propagation delay. However, this delay is usually negligible compared to the other propagation delays in the system. And then there are the size and weight requirements that need to be addressed.

Wait one moment. Instead of keeping the symbol rate constant and decreasing the information rate, we can increase the symbol rate and keep the information rate constant. When I generally think of communications systems, I usually think of them as a function of only two parameters: signal-to-noise ratio and bandwidth. If we compare an uncoded system with a coded system for the same information rate (i.e., bits per second), the coded system will have a higher symbol rate at the output of the encoder (i.e., symbols per second) than the uncoded system. In other words, the coded system spreads its signal energy over more transmitted symbols within the same bandwidth. The energy associated with each coded symbol is less than the uncoded symbol. Therefore, the symbol error rate of a coded system will be greater than an uncoded system. If the decoding of an error correction code has a better

performance than an uncoded system (i.e., the coded system having redundancy overcomes the higher symbol error rate better than the uncoded system with its lower symbol error rate without having redundancy) at fairly high-to-low bit error rate regions, then we obtain a coding gain in signal-to-noise energy. If the resultant code has a worse performance, then it is a bad error correction code. An error correction coding system will have worse performance at very high bit error rates than an uncoded system. However, in an error correction system we can fairly easily adjust the coding gain to whatever we need for the operating regions.

In summary, error correction coding may not require any additional antenna power or bandwidth; a coding gain can be obtained over the same bandwidth by either decreasing the information rate or by modulation techniques (which are usually more complicated and are designed to spread the available signal energy over more symbols). Error correction coding can even cause the overall power and bandwidth considerations to be relaxed.

Referring back to our case of the "on" and "off" messages, let us now add redundancy to the message. (Sometimes the messages are called the data field or the information field. Be aware that different people mean different things when they talk about "information;" I will refer to the messages as either messages or as data fields.) Instead of transmitting a 1 to indicate "on" and a 0 to indicate "off," let us use more than this minimum of one binary symbol to represent these two messages. Let us now say that we want to use three binary symbols to represent these same two messages; we are adding redundancy. There are eight possible states, but we are only going to transmit two of them.

Let us develop this simple block code as an example to demonstrate coding. Let us randomly pick say a three symbol sequence of binary code symbols to represent "on." Let us choose [101] for "on." Usually to construct the most efficient code, choose the representation for "off" to be as different as possible from [101]. Let us then represent "off" as [010]; [010] is the code word for "off." A code word pertaining to a block code is defined to have a block length of n symbols representing (or corresponding) to the message length of k symbols where each unique code word is unique to each message. In this example, the code word [101] has a block length n=3 symbols, a message length k=1 symbol, and the one and only [101] is unique to "on." The number of 1's in any binary word is its weight w. The number of positions which are different between two words of the same length is its distance d. In this example the weight w of the word [101] is two and the weight w of the word [010] is one; w[101]=2 and w[010]=1. Also, the distance d between the words [101] and [010] is three; d[101,010]=3. Notice that they differ in three locations. Also, notice that

d[101,010] = w[101+010] because of modulo-2 addition. In this example d[101,010] just so happened to equal w[101] + w[010]; generally d[xxx,yyy] ≠ w[xxx] + w[yyy]. The minimum distance d_{min} is defined to be the distance between the two closest code words. Since we only have two code words in this example, d_{min} = d[101,010] = d[010,101] = 3. By representing a message with more binary symbols than is necessary to label all of the messages, we are adding redundancy to the message. [101] and [010] are code words that label our messages "on" and "off" respectfully. Now notice that all of the other possible words, denoted as non-code words, are invalid in that they will NEVER (never say never) be transmitted; [000], [001], [011], [100], [110], and [111] are denoted as non-code words. However, in a noisy channel it is likely to receive these invalid words. This is due to the noise corrupting our original code word representing our message.

Now let us try to decode the received word in the presence of noise. If we receive a [101] we will assume that we transmitted "on." If we receive a [010] we will assume that we transmitted "off." If we receive anything else we will pick the closest match to either [101] or [010]. There, that is our decoding algorithm; that is maximum likelihood decoding (MLD). A maximum likelihood decoder (MLD) is defined as a decoder whose code word estimate is determined by maximizing the conditional received word probability given that a code word has been transmitted. It also assumes additive white Gaussian noise (AWGN) and a memoryless channel. A MLD simply decodes the received word into its closest code word measured by its symbol distance d.

But who is to say that errors cannot change one transmitted code word into being closer to a different code word, or even being that different code word itself? Actually errors can, but the probability of it doing so can be made extremely small in comparison. The idea of receiving a non-code word which is nearest a single code word and then proceeding to decode it to be the message represented by that code word, is called MLD.

If we receive [101] we decode it into "on" and if [010] then "off." Seems like a given, does it not!? It seems as if no noise was injected into our code word. Maybe noise was injected and maybe not; there is no way that the decoder can absolutely know. For example: what happens if we want to communicate "on" over our communication channel? Well, we transmit [101]. But now let's say that noise infiltrated our system and our receiver received [010]; an error in each location just happened to occur. The decoder takes the [010] and says that it is identical to the representation for "off." It then proceeds to incorrectly decode the [010] into the message "off" and pushes it down the line; our decoder seems happy. An undetectable error has just occurred; it is called an

undetected decoding or decoder error. The reason why we still go ahead and use this MLD idea is that the probability of having three errors in our received code word is much less than having two errors, is very much less than having one error, and is very, very much less than having no errors. Equivalently, the probability of having no errors in our received code word is much more than having one error, is very much more than having two errors, is very, very much more than having three errors, and so forth. Since our decoder only has the word it received through our noisy coding channel at its disposal, it can figure out all the possible combination of errors to get it to any of the code words. Using MLD the decoder selects the combination of errors which has the fewest number of errors necessary to get it to the nearest code word. This pattern of errors is the MOST PROBABLE ERROR PATTERN, but MLD does NOT guarantee it to be the ACTUAL ERROR PATTERN. Since no one has come up with a better idea to determine the estimate of the noise and that MLD can be proven to be optimum in an AWGN channel, MLD is still the best method to follow. MLD can give excellent results.

Table 2.2-1 is given to demonstrate why this simple block code example can correct one error symbol or fewer in its block length of three and no others. Notice that more than one error symbol within a received word results in an improperly decoded code word. This is because all the possible received patterns (or words) have already been mapped into their correct code words using the MLD principle. Since the probability of having less errors is much greater than having more errors, the received patterns that have the least number of errors are mapped first. We then map any remaining patterns that have more errors until we run out of possible received patterns to map. For our example, there are no remaining received patterns that correspond to more than a single error symbol.

Also notice that in table 2.2-1 ALL received words with more than one error symbol are replicated in the received words with less than or equal to one error symbol. Since ALL single or fewer error symbols can be mapped correctly into its nearest code word and since NOT ALL (in fact NONE in this example) double error symbols can be mapped correctly into its nearest code word, this particular code is a single error correction (t=1) code. Since this code makes undetectable decoding errors for all error symbols greater than t symbols, this code is perfect. There's nothing all that great about a perfect code. In fact, a perfect code utilizing all of its error correction capability is not necessarily a good code in terms of communication efficiency. All the syndromes are used for error correction capability and thus none are reserved for error detection capability; perfect codes have no error detection capability unless some error correction capability is forfeited.

A code with minimum distance d_{min} can be used as a t error correcting, t_d additional error detecting code if and only if $2t+t_d+1 = d_{min}$. If we use all the error correction capability of this small example, i.e., t=1, then $t_d = d_{min}-2t-1 = 0$ because $d_{min}=3$ as discussed earlier. If we decreased the error correction capability of our single error correcting example by one, i.e., $t_{new} = t_{old}-1 = 0$, then $t_{d,new} = d_{min}-2t_{new}-1 = 2$; in other words this results in a zero error correcting, double error detecting code. When we decrease the error correction capability of a perfect code, we usually denote the resultant code as a non-perfect code.

TABLE 2.2-1. - DECODED WORDS AS A FUNCTION OF ERROR PATTERN

Number of errors	Error word	Received word [101] / [010] transmitted	Decoded code word [101] / [010] transmitted
0	[000]	[101] / [010]	[101] / [010]
1	[100]	[001] / [110]	[101] / [010]
1	[010]	[111] / [000]	[101] / [010]
1	[001]	[100] / [011]	[101] / [010]
2	[110]	[011] / [100]	[010] / [101]
2	[011]	[110] / [001]	[010] / [101]
2	[101]	[000] / [111]	[010] / [101]
3	[111]	[010] / [101]	[010] / [101]

All (n,1) block codes can be perfect if n is odd; this example is a perfect (3,1) block code. If this example was a non-perfect code, like most others, then the code would have some error detecting capability without sacrificing any error correction capability. Error detection is obtained by recognizing that some syndromes are invalid in that they are never to be used. More on syndromes will be presented later.

In this example, code symbols are only one digital bit (or binary symbol) in length. Therefore, this example is a binary block error correction code.

This particular (3,1) block code is a random error correcting code; it can correct some bit errors distributed randomly within a bit (or code symbol) stream. It is not a burst error correcting code; it cannot correct bursts of errors which are random errors that

occur very, very near to each other.

This particular (3,1) block code is not a linear block code. Linear block codes have the property that every code word summed with any code word is a code word. Notice that all binary based linear block codes have at least the all-zero code word because any code word added to itself is the all-zero code word using modulo-2 arithmetic. This (3,1) block code doesn't have the all-zero code word and therefore is grounds enough to ostracize it from being linear. Also notice that if we take all the code words in our (3,1) block code and sum them all together we receive the word [111]. [111] is not a code word; this is also grounds for ostracism. Death to non-linear codes!

In this example any double or triple error pattern can decode into its nearest code word, BUT the decoder would not be able to decode it into the correct code word. Notice that if the decoder did try to correct the double and triple error patterns into the original code word that was transmitted, then the no error case and the single error cases would not be able to decode correctly. Using MLD we will correct all single and fewer errors. Since this example is a perfect code, any pattern of more than t errors will cause decoder errors. Summarizing for this perfect code example, all errors look like either a single error or no error because the decoder is using the MLD principle. Since all error patterns with one or fewer errors decodes into the correct code word, we have a single error correcting code. Since it happens to be a perfect code, more than t errors cannot be decoded correctly or even detected; more than t errors in a perfect code produces undetectable decoding errors.

2.3 LINEAR BLOCK CODES

Figure 2.3-1 presents the relationships between many different types of codes. From this diagram we can see how RS codes relate to other error correcting codes. There are two types of error correction codes: tree codes and block codes. Even though we can sometimes combine them in certain ways using characteristics of the other, we still have only two types of error correction codes.

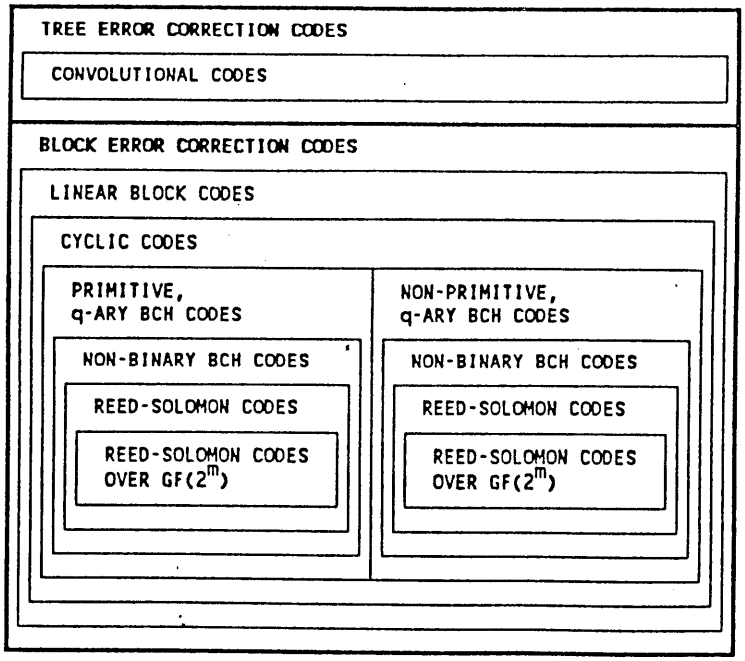


Figure 2.3-1. - General Venn diagram of error correction codes.

Reed-Solomon codes are non-binary, BCH, cyclic, linear block error correction codes.

The major characteristics of linear block codes are a block architecture, optional systematic structure, and all code words are sums of code words. It has a block length of n symbols and a message length of k symbols. If the code is systematic, then it also has an unaltered data field of k symbols independent of the associated parity-check field of n-k symbols.

Cyclic codes are a subset of linear block codes. They have the same characteristics as other linear block codes, but with an additional characteristic; every cyclic shift of a code word is also a code word. Cyclic codes are easier to encode and decode into systems than linear block codes. The encoding operation (similar to the first decoding stage) can be implemented into either a SRC or a linear sequential circuit. Also, the decoders' implementations become more practical due to the increase in the cyclic codes' algebraic structure.

P-ary BCH codes are a special case of q-ary BCH codes which are a subset of cyclic codes. P-ary BCH codes' code word symbols and code word generator polynomial g(X) coefficients are from GF(P) for P being a prime number. The field elements and the code word generator's roots of P-ary BCH codes are from $GF(q)=GF(P^m)$

for q being the order of the field and m being an integer greater than one. They have the same characteristics as other cyclic codes, but with an additional characteristic; P-ary BCH codes can fairly easily be implemented into systems with any error correction capability t of t symbols along with particular choices of the message length k of k symbols and the block length n of n symbols. Also, BCH codes can be either primitive or non-primitive. Primitive BCH codes are defined as codes whose block length n is P^m-1. Non-primitive BCH codes have a block length n other than $n=P^m-1$; e.g., a shortened BCH code is a non-primitive code because it has a shorter block length n which divides P^m-1. In general, designing encoders and decoders for multiple error correcting P-ary BCH codes is easier than for many other cyclic codes.

Binary BCH codes are the most often used of the many P-ary BCH codes; binary BCH codes are simply 2-ary BCH codes. The code word symbols of binary BCH codes are binary; they are from GF(P)=GF(2). The field elements used in binary BCH codes are non-binary; they are from $GF(q) = GF(P^m) = GF(2^m)$ for q being the order (or size) of the field and for m being an integer greater than one. Also, the code word generator polynomial g(X) has binary (i.e., 2-ary) coefficients from GF(P)=GF(2) and the code word generator's roots are from $GF(P^m)=GF(2^m)$. A t error correcting ($t<2^{m-1}$), primitive, binary BCH code has the following parameters:

block length:	$n = 2^m-1$	code symbols
number of parity-checks:	$n-k \leq mt$	code symbols
minimum distance:	$d_{min} \geq 2t+1$	code symbols

These codes have some inherent error detection capability without sacrificing any of the error correction capability. If some error correcting capability is sacrificed for an additional error detection capability t_d, then the resultant t error correcting, t_d additional error detecting (t_d is an even number), primitive, binary BCH code would have the following parameters:

block length:	$n = 2^m-1$	code symbols
number of parity-checks:	$n-k \leq m(t+(t_d/2))$	code symbols
minimum distance:	$d_{min} \geq 2t+t_d+1$	code symbols

P-ARY BCH CODES ARE ACTUALLY q-ARY BCH CODES. Binary BCH codes are actually a special case of q-ary BCH codes. Also, non-binary BCH codes are simply all q-ary BCH codes which are not binary BCH codes.

A t error correcting, q-ary BCH code (with the code symbols and the generator's coefficients being from GF(q), the field elements and the generator's roots being from $GF(q^c)$, and c being an integer greater than one) has the following parameters:

block length:	$n = q^c-1$ code symbols
number of parity-checks:	$n-k \leq 2ct$ code symbols
minimum distance:	$d_{min} \geq 2t+1$ code symbols

P-ary BCH codes can be derived from q-ary codes by simply setting q=P to be a prime number and c=m to be an integer greater than one. This means that for P-ary BCH codes, the generator's roots (and the field elements) are from $GF(q^c)=GF(P^c)=GF(P^m)$ and the generator's coefficients (and the code symbols) are from GF(q)=GF(P). For binary BCH codes the generator's roots (and the field elements) are from $GF(q^c)=GF(P^c)=GF(P^m)=GF(2^m)$ and the generator's coefficients (and the code symbols) are from GF(q)=GF(P)=GF(2).

Now, I would like to finally talk a little more about Reed-Solomon codes! RS codes can be derived from q-ary codes by simply setting $q=P^m$ to be a power of a prime number and c to be 1. This means that for RS codes the generator's roots (and the field elements) are from $GF(q^c)=GF((P^m)^c)=GF((P^m)^1)=GF(P^m)$ and the generator's coefficients (and the code symbols) are from $GF(q)=GF(P^m)$. A t error correcting, primitive RS code has the following parameters:

block length:	$n = q-1 = P^m-1$ code symbols
number of parity-checks:	$n-k = 2t$ code symbols
minimum distance:	$d_{min} = 2t+1$ code symbols

The code symbols of a binary based RS code are non-binary; they are from $GF(q)=GF(P^m)=GF(2^m)$, not GF(q)=GF(P)=GF(2). For binary based (i.e., P=2) RS codes, the generator's roots (and the field elements) are from $GF(q^c)=GF((P^m)^c)=GF((P^m)^1)=GF(P^m)=GF(2^m)$ and the generator's coefficients (and the code symbols) are from $GF(q)=GF(P^m)=GF(2^m)$. A binary based, t error correcting, primitive RS code has the following parameters:

block length:	$n = q-1 = 2^m-1$ code symbols
number of parity-checks:	$n-k = 2t$ code symbols
minimum distance:	$d_{min} = 2t+1$ code symbols

For a primitive RS code, once the extension m and the base P are determined, then the block length n is automatically set. Then once either the error correction capability t or the message length k is determined for a primitive RS code, the other is respectively set. For example, if $q = P^m = 2^m = 2^3 = 8$ (P=2 denotes binary based), then this can either be a (7,5) or a (7,3) or a (7,1) RS code depending if t = 1, 2, or 3 symbols respectfully.

It should be noted that RS codes have very unique and powerful features: RS codes satisfy the Singleton bound $d_{min} \leq n-k+1$ because for a RS code, $d_{min} = 2t+1 = n-k+1$. Therefore, RS codes are MDS or synonymously called optimal. It is also worth pointing out that

the block length n of a RS code over GF(q)=GF(P^m) can be extended to either q or q+1 while still maintaining the MDS condition. Also, a (n,k) RS code can be shortened to be a (n-l,k-l) RS code (for l even and l<k) while maintaining the MDS condition. In other words, some non-primitive RS codes are also MDS. Another nice RS feature is that the designed minimum distance is exactly equivalent to the actual minimum distance d_{min}, i.e., d_{min}=2t+1 not $d_{min}\geq$2t+1. Typically, when RS codes are designed into a system we use binary based (P=2) RS codes. Just like any other linear block code, RS codes can be either systematic or non-systematic. Usually if systematic structure is easily implemented into the system and does not decrease the coding gain, we do it. RS codes are not only very powerful burst error correcting codes, but can also be powerful random error correcting codes.

2.4 SOME MORE RELATED TOPICS

There are several other areas of the system with which the error correction system must interface. We must be concerned with choosing the correct code or combination of codes to most efficiently meet or exceed the engineering problems of noise and channel capacity. We must be concerned with the implementation architecture and where the coding circuitry is located within the system. We must be concerned with synchronizing to our message. Besides these concerns and concerns which fall under these, there may be other concerns to seriously consider. This section will briefly address interleaving, modifying block codes, burst errors, concatenation, and synchronization.

The error correction capability of burst error codes, concatenation codes, and random error correcting codes can increase if interleaving is performed. The purpose of block interleaving (in regard to error correction coding) is to average out the bursts of burst errors over several code words. Interleaving can be done by simply shuffling the encoder's output (or encoders' outputs) to an interleave depth I. Instead of transmitting one code word followed by another, we will transmit the first symbol of the first code word, the first symbol of the second code word, ..., the first symbol of the Ith code word. Then we will transmit the second symbol of the first code word, the second symbol of the second code word, ..., the second symbol of the Ith code word. Then we will keep repeating this until the nth symbol of the first code word, the nth symbol of the second code word, ..., the nth symbol of the Ith code word has been transmitted. We then repeat this process by taking another set of I code words and interleaving them the same way. This algorithm is the usual method of block interleaving. If the code is systematic, then all the consecutive (n-k)I parity-

check symbols will follow all the consecutive kI message (or data) symbols. The decoder must perform a similar operation of de-interleaving (or de-shuffling). Besides block interleaving some other useful interleaving algorithms worth mentioning are convolutional interleaving (not related to convolutional codes) and helical interleaving. In general, interleaving is simply efficiently shuffling the symbols around to average out the very long burst errors over more code words.

Block codes can be modified in six ways: The block length n can be increased by attaching additional parity-check symbols (denoted as extending) or by attaching additional message symbols (denoted as lengthening). The block length n can be decreased by removing parity-check symbols (denoted as puncturing) or by removing message symbols (denoted as shortening). The last two ways are when the block length n does not change, but the number of code words is increased (denoted as augmenting) or decreased (denoted as expurgating). Modified codes are sometimes about the same level of encoding and decoding complexity as the original, unmodified code. There are many ways to perform these modifications. Some modifications effect the error correction capability and some do not. To understand more about how to specifically modify a specific code, the reader should reference a more detailed text than this tutorial.

Let a burst error length b be defined as the number of bits from the first bit error in a bit stream to another bit error which is within a particular portion of the bit stream such that there may be some non-error bits in the burst error and such that there are all non-error bits between consecutive burst errors. A RS code can correct a maximum burst error length b_{max} of length $b_{max} = (It-1)m+1$ bits within an interleaved block system of I code words being the block of Imn bits. If interleaving is not performed, then I=1. A RS code can correct any combinations (or patterns) of t or fewer errors within each code word. If interleaving is used, then a RS code can correct most combinations (or patterns) of "It" or fewer errors within the frame of I code words being the block of Imn bits. RS codes are very powerful burst error correcting codes and can also be made to be very powerful random error correcting codes.

Usually noise possesses both random characteristics and burst characteristics. This results in random errors and burst errors within the received (or play back) data. Since codes are better at either burst errors or random errors, concatenation between different codes are often performed. Usually concatenation is when an excellent random error code is used as the inner code and an excellent burst error code is used as the outer code. Convolutional error correction codes are powerful random error

correcting codes. When random errors within the channel become more and more like burst errors, the convolutional decoder (Viterbi, sequential, or majority logic decoders) usually generate burst decoding errors. These convolutional, burst decoding errors could then be corrected by using a good burst error correcting code, such as a RS code, as the outer code. The data sequence is that data are input to the outer code encoder, then its output is input to the inner code encoder, then its output is transmitted (or recorded), corrupted by noise, and then input (or played back) to the inner code decoder, then its output is input to the outer code decoder, and then finally its output is either the decoded message (or data) or the decoded message and corresponding parity-check. Often this concatenation design results in superior performance compared to a single code having some random error and some burst error correcting capability. However, we pay for concatenation with a decreased overall code rate.

Cyclic block codes require synchronization words for the decoder. With cyclic block codes we usually attach synchronization words onto the beginning of the transmitted code words. Usually this is done synchronously and periodically by attaching one synchronization word to every code word to be transmitted. However, if the code words are interleaved, then we usually attach one sync word to every I code words. Sync words typically do not use error correction coding, but are typically designed to a particular channel in an optimum manner. Cyclic block codes usually are designed with sync words not only to sync to the non-binary symbols (if the code is one with non-binary symbols), but also to the first symbol of the code word (or first symbol of the I code words). However, synchronization for cyclic codes can be established and maintained without using any sync words. These types of designs are more complicated, require additional processing hardware, increase the propagation delay, and are less efficient today than using sync words. In comparing a block code and a tree code, tree codes such as convolutional codes often do not require sync words to acquire synchronization. Convolutional codes have an inherent coding sync capability; most convolutional codes are self-synchronizing codes. This coding sync is not a code word (or block) sync, a code symbol sync, an interleaved sync, a frame sync, a packet sync, or a header sync; it is just a sync for the code to hopefully be able to decode the received bit stream containing errors back into the original message. So, cyclic block codes require synchronization determined either by appending synchronization words to code words or by a lot of additional processing. Obtaining synchronization for cyclic codes by additional processing and not using sync words does not allow a bit stream to be random; cyclic codes are not self-synchronizing because only certain types of data can be transmitted. Tree codes require synchronization usually determined from its self-

synchronizing structure, but they usually need more redundancy for the same amount of coding gain compared to efficient cyclic codes.

2.5 WHITE, BLACK, AND GRAY ZONES

A perfect code only has a single white decision zone for error correction. If the received word of a perfect code is closest to one particular code word than any other code word AND is within a distance of t symbols away from it (i.e., T≤t), then the received word is in the white zone. Received words are decoded using MLD. We should understand by now that MLD DOES NOT GUARANTEE TO ALWAYS DECODE INTO THE ACTUAL MESSAGE THAT WAS TRANSMITTED. However, it DOES GUARANTEE to always correctly decode the actual message that was transmitted IF T≤t actual errors were injected into the code word. If T>t actual errors occurred in a perfect code, non-erasure system, then the decoder would make an undetectable decoding error. Actually, it is possible to add a little error detection capability to a perfect code while retaining most or some the error correction capability of the code. However, doing this transforms the perfect code into a non-perfect code. This can be done by denoting some code words as invalid, thus not using the full t error correction capability; this would create a black zone.

A code which is not perfect has a white zone, has a black zone, and might have a gray zone. Reed-Solomon codes are not perfect!!! RS codes have white, black, and gray zones. If the received word is T≤t ($T \leq t_E+t_E$" for erasure systems) actual errors (the units are code symbols or just symbols) or away from its nearest single code word, then it is in the white zone; this received word is guaranteed to be decoded correctly. If the received word is T>t ($T > t_E+t_E$" for erasure systems) actual errors away from its nearest single code word AND can be correctly decoded into this code word (this is a function of the particular code itself), then it is in the gray zone. However, even though it is possible to correctly decode words within a gray zone, it is not usually realized into systems. Gray zones are not usually used because received words within this gray zone are usually difficult to find and decode. Therefore, almost all of the time, candidate words for the gray zone are treated as if they are in the black zone. The last zone is the black zone. If the received word is not in either the white zone or the gray zone, then it is in the black zone; this received word is not able to be decoded. However, if the received word is in the black zone, then the received word can be flagged as T>t ($T > t_E+t_E$" for erasure systems) errors have definitely occurred within it; non-perfect error correction codes have some degree of error detection while retaining their full error correction capability. The decoder can not correct any errors in the black

zone, but if desired, the noisy message can be extracted from a systematic received word. This may or may not be good enough. If passing either the received word or the noisy message from a systematic code through the system is not desired, then it might be desired for the system to ARQ and/or throw the received word in the trash or to the bit bucket!

It should also be noted that a non-perfect code (with t error symbol correction capability and some error detection capability) can be designed into a code with less error correction, less undetectable decoding errors, and more error detection. This is done by shrinking the white zone and increasing the black zone.

When we decode only using the white and the black zones, we are performing "bounded distance decoding." When we try to correctly decode- by also using the entire gray zone, we are performing "complete decoding." Symbol erasure is NOT complete decoding.

In general, this zone idea helps us to graphically visualize the concept of error correction coding. The white zone is typically known as the error correction domain, the gray zone as the error correction domain beyond the distance of t symbols (or beyond the distance of t_E+t_E" symbols for an erasure system), and the black zone as the error detection domain of an error correcting code.

2.6 SUMMARY

We got to see a super simple (3,1) block code example. It was systematic in the sense that some portion of the code word always contained the unaltered message, i.e., "ON" = "1" from the code word "ON" = [010] and "OFF" = "0" from the code word "OFF" = [101]. It is systematic, is not linear, and does not have a gray or black zone, but it is a "perfect" code!

We also have been introduced to how block error correction generally works. We have been introduced into decoding errors, MLD, distance and weight, random and burst errors, concatenated codes, synchronization, error correction and/or detection zones, and the famous BCH codes which include the Reed-Solomon codes. Some detailed definitions which have not yet been discussed entirely will be discussed in the following chapters.

Now, enough introductory material. The remaining chapters work some of the many RS coding algorithms for the case of our primitive (15,9) RS code example.

CHAPTER 3
REED-SOLOMON ENCODING

Let's get into RS coding! There are many error correction coding algorithms around, but we mainly want to consider very efficient (or powerful) random and burst error correcting codes - RS codes. RS codes are BCH codes which are a subset of cyclic block codes. Cyclic block codes are a subset of linear block codes which are a subset of block codes which are a subset of error correction codes in general. Therefore, RS codes are the great, great grandchildren of block codes (see figure 2.3-1).

Within this chapter we will start working our (15,9) RS code example. We will be able to apply the material learned or reviewed from chapters 1 and 2. A RS block diagram of the encoder is presented along with the parameters and equations necessary to construct our transmitted words. And now for the fun!

3.1 REED-SOLOMON ENCODER

Since we explained a general block coding system in chapter 2, let us now talk about RS coding in particular. Assume the parity-check information CK(X) is obtained from the message information M(X) by the modulo-g(x) function.

$$CK(X) = X^{n-k}M(X) \bmod g(X)$$

or CK(X) could equivalently be found as:

$$\frac{X^{n-k}M(X)}{g(X)} = Q(X)g(X) + CK(X)$$

where X^{n-k} is the displacement shift, M(X) is the message, Q(X) is the quotient, g(X) is the generator, and CK(X) is the parity-check.

The code word C(X) that we will transmit is comprised of the parity-check information CK(X) appended systematically onto the message information, $C(X) = X^{n-k}M(X) + CK(X)$. The X^{n-k} purpose is to shift the message M(X) to higher ground in order that the message M(X) does not overlap and add to the parity-check CK(X) within the code word $C(X) = X^{n-k}M(X) + X^{n-k}M(X) \bmod g(X)$. This is done to retain the systematic structure. Systematic structure is defined as simply taking our message symbols and appending parity-check symbols to it without changing our message symbols. This is part

of the encoding process. The degree of g(X) is n-k and the degree of $X^{n-k}M(X)$ is either 0 [if M(X)=0] or from n-k to n-1 [if M(X)≠0]. Notice that the X^{n-k} factor in the parity-check $CK(X) = X^{n-k}M(X) \bmod g(X)$ forces the modulo function for all non-zero messages M(X), e.g., even when M(X)=1, $X^{n-k}M(X) \bmod g(X) = X^{n-k} \bmod (X^{n-k}+\ldots)$ must be calculated. The degree of $CK(X) = X^{n-k}M(X) \bmod g(X)$ is from 0 to n-k-1. Therefore, since n>k, the check information CK(X) never overlaps the message information M(X). Thus, systematic structure is retained. The message M(X) is in the k highest locations within the code word C(X); the parity-check CK(X) is in the n-k lowest locations.

If we desire to use a non-systematic code for some reason, then often we use $C(X)_{\text{non-systematic}} = g(X)M(X)$.

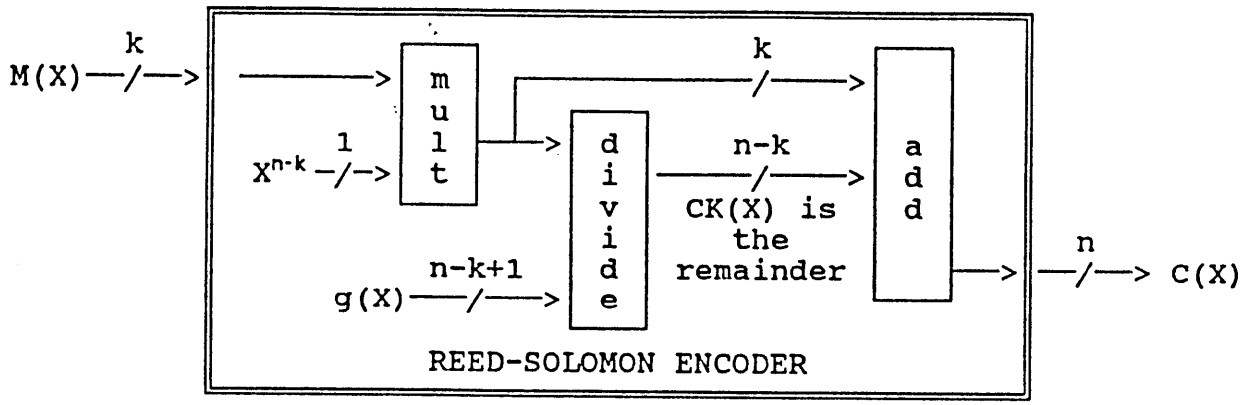


Figure 3.1-1. - Block diagram of a Reed-Solomon encoder.

Now, enough words. Let us finally see what a RS encoder looks like. Figure 3.1-1 presents the block diagram for this RS encoder and table 3.1-1 presents its associated polynomials.

TABLE 3.1-1. - POLYNOMIAL DEFINITIONS OF A RS ENCODER

message (or data or information) M(X) consisting of message symbols M_i: $M(X) = M_{k-1}X^{k-1} + M_{k-2}X^{k-2} + \ldots + M_1X + M_0$

generator (or code word generator) g(X) consisting of generator symbols g_i: $g(X) = X^{2t} + g_{2t-1}X^{2t-1} + \ldots + g_1X + g_0$

TABLE 3.1-1. - Continued

parity-check CK(X) consisting of parity-check symbols CK_i:

$$CK(X) = X^{n-k}M(X) \bmod g(X)$$
$$= CK_{n-k-1}X^{n-k-1} + CK_{n-k-2}X^{n-k-2} + \ldots + CK_1X + CK_0$$

code word C(X) consisting of code word symbols C_i:

$$C(X) = X^{n-k}M(X) + CK(X)$$
$$= X^{n-k}M(X) + X^{n-k}M(X) \bmod g(X)$$
$$= M_{k-1}X^{n-1} + \ldots + M_0X^{n-k} + CK_{n-k-1}X^{n-k-1} + \ldots + CK_0$$
$$= C_{n-1}X^{n-1} + C_{n-2}X^{n-2} + \ldots + C_1X + C_0$$

3.2 (n,k) RS CODES

Given a single value of the Galois field extension m, i.e., $GF(P^m)$, a set of RS codes with varying error correction capabilities, block lengths, and rates can be constructed. The P^m unique code symbols are constructed from the field generator polynomial F(X) and the primitive element $\alpha(X)$. The parity-check information is obtained using the generator polynomial g(X) with roots from $GF(P^m)$. A (n,k) RS code is defined given values for m, n, and g(X). However, when we get into the implementation we need to also know P (which is almost always 2), F(X) [which is almost always a primitive polynomial p(X)], $\alpha(X)$ [which is almost always $X=\alpha$], and α^G [which is any primitive element of $GF(P^m)$ using F(X) and is almost always set to α^1 in order to simplify the notation].

Table 3.2-1 lists all the RS codes in $GF(2^m)$ for $m \leq 4$. The bracketed (1,1) code shown in the table is presented to show that any (k,k) code is not an error correction (and/or detection) code; it is not adding redundancy. Notice the (3,1) RS code. I believe this code is valid in that it can be encoded and decoded using the standard procedures, yet it is edited from most (maybe all) coding literature. I believe this is due to the fact that a (3,1) RS code does not have a long enough block length to produce a substantial decrease in error rate (or increase in SNR). A (3,1) RS code seems to be the smallest RS code possible. It should lend itself to be a good scholastic exercise. It can correct a maximum burst error of one symbol within its block length of three symbols; it can correct a maximum burst error of two digital bits (or binary symbols) if the burst error occurred within a single symbol. It is a single symbol correction code and can use the same field generator as its code word generator polynomial.

From the table notice that a (n,k) RS code requires two parity-checks per error; one is for the location within the code word and the other is for the error value at that location. That

is, n-k=2t.

From table 3.2-1 it should also be noted that as m increases the number of possible RS codes increases in an exponential manner! Thus, once the block length is determined from some fairly large m, we can pick a code with a desirable pair of rate versus correction capability. The rate (or code rate) r of a code is the ratio of the number of message symbols k to the block length n; r = k/n. Let the block correction BC of a code be the ratio of the number of correctable symbols t to the block length n; BC = t/n. Let the average message correction MC_{avg} of a code be the ratio of the average number of correctable message symbols $t_H = (k/n)(t) = rt$ to the number of message symbols k; $MC_{avg} = t_H/k = BC$. It is desirable, but impossible, for the rate r to approach 100 percent while the error correction capability t approaches n. Usually a rate parameter and a coding gain parameter is of prime functional importance.

TABLE 3.2-1. - RS CODES OVER GF(2^m) FOR m≤4

m	n	k	t	r	BC
[1]	[1]	[1]	[0]	[100.0%]	[00.0%]
2	3	1	1	33.3%	33.3%
3	7	5	1	71.4%	14.3%
3	7	3	2	42.9%	28.6%
3	7	1	3	14.3%	42.9%
4	15	13	1	86.7%	6.7%
4	15	11	2	73.3%	13.3%
4	15	9	3	60.0%	20.0%
4	15	7	4	46.7%	26.7%
4	15	5	5	33.3%	33.3%
4	15	3	6	20.0%	40.0%
4	15	1	7	6.7%	46.7%

3.3 (15,9) RS PARAMETERS

A primitive RS code has the following parameters over GF(P^m):

block length = n = P^m-1 (units are symbols)
parity-check length = n-k = 2t (units are symbols)
minimum distance = d_{min} = 2t+1 (units are symbols)

All right, let us select a code from table 3.2-1. Let us use the (15,9) RS code as our example that we are going to work throughout this tutorial. The (15,9) RS code is a classical choice for an instructional example.

People usually compare codes in two viewpoints: The first is the theoretically possible viewpoint and the second is the let us implement it viewpoint. Coding engineers often compare codes by trying to maximize channel capacity which automatically brings in factors of high code rate, SNR, message throughput, and detectable and undetectable decoding errors. Others are concerned with these factors too, but still must hold a different viewpoint. Implementation engineers often compare codes by a subjective function of need, performance, risk, and the allocation of resources.

Because this is a tutorial and is meant to help people understand, the classic (15,9) RS code is chosen to be demonstrated. Let us construct our (15,9) RS coding example with the parameters as shown in the following table.

TABLE 3.3-1. - THE PRIMITIVE (15,9) RS EXAMPLE PARAMETERS

block length	$n = 15$ symbols
message length	$k = 9$ symbols
code rate	$r = k/n = 60\%$
parity-check symbol length	$n-k = 6$ symbols
minimum code distance	$d_{min} = n-k+1 = 7$ symbols
error correction capability	$t = (n-k)/2 = 3$ symbols
block correction capability	$(n-k)/2n = 20\%$
average message correction capability	$(n-k)/2n = 20\%$
Galois field order	$q = n+1 = 16$ symbols

This is a binary based system (i.e., P=2) because $P^m = q = 16 = (2)(2)(2)(2) = 2^4$ for P being prime. This determines that P=2 states per 2-ary symbol (i.e., 2 states per binary symbol) and that the code symbol length m is 4 binary symbols.

number of code words	$P^{mk} = 2^{36} > 10^{10}$ words
number of received words	$P^m = 2^{60} > 10^{18}$ words
number of undetected decoding errors	$P^{mk} = 2^{36} > 10^{10}$ words
number of error patterns	$P^m = 2^{60} > 10^{18}$ words

$$
\begin{aligned}
g(X) &= [\ (X+\alpha)(X+\alpha^2)][\ (X+\alpha^3)(X+\alpha^4)][\ (X+\alpha^5)(X+\alpha^6)] \\
&= [X^2+(\alpha+\alpha^2)X+\alpha^3][X^2+(\alpha^3+\alpha^4)X+\alpha^7][X^2+(\alpha^5+\alpha^6)X+\alpha^{11}] \\
&= [X^2+\alpha^5X+\alpha^3\ \ \ \ ][X^2+\alpha^7X+\alpha^7\ \ \ \ ][X^2+\alpha^9X+\alpha^{11}\ \ \ \ ] \\
&= [X^4+(\alpha^7+\alpha^5)X^3+(\alpha^7+\alpha^{12}+\alpha^3)X^2+(\alpha^{12}+\alpha^{10})X+\alpha^{10}][X^2+\alpha^9X+\alpha^{11}] \\
&= (X^4+\alpha^{13}X^3+\alpha^6X^2+\alpha^3X+\alpha^{10})(X^2+\alpha^9X+\alpha^{11}) \\
&= X^6+(\alpha^9+\alpha^{13})X^5+(\alpha^{11}+\alpha^7+\alpha^6)X^4+(\alpha^9+1+\alpha^3)X^3+(\alpha^2+\alpha^{12}+\alpha^{10})X^2+(\alpha^{14}+\alpha^4)X+\alpha^6 \\
&= X^6 + \alpha^{10}X^5 + \alpha^{14}X^4 + \alpha^4X^3 + \alpha^6X^2 + \alpha^9X + \alpha^6
\end{aligned}
$$

Therefore, the generator polynomial $g(X) = X^6+\alpha^{10}X^5+\alpha^{14}X^4+\alpha^4X^3+\alpha^6X^2+\alpha^9X+\alpha^6$. Remember that we used α, α^2, α^3, ..., α^{2t} as our roots of $g(X)$. These roots will be used in chapter 4.

If we had wanted to, we could have constructed our generator polynomial $g(X)$ as also being a self-reciprocating polynomial $f(X)_{s-r} = X^i f(X^{-1})$. Self-reciprocating polynomials have equivalent jth and i-jth coefficients. The $g(X)$ that we are going to use is not a $f(X)_{s-r}$ because $1 \neq \alpha^6$, $\alpha^{10} \neq \alpha^9$, and/or $\alpha^{14} \neq \alpha^6$. An example of a self-reciprocal polynomial is $f(X)_{s-r} = X^6+\alpha^{10}X^5+\alpha^{14}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^{10}X+1$. Not all $f(X)_{s-r}$ are a valid $g(X)$. However, one is guaranteed to construct a self-reciprocating generator polynomial $g(X)_{s-r}$ by choosing $FCR = 2^{m-1}-t = (n+1)/2-t$ and $\alpha^j=\alpha(X)=\alpha$ for (n,k) primitive RS codes. The 2t roots are the following:

$$\alpha^{2^{(m-1)}-t},\ldots,\alpha^{2^{(m-1)}-1},\alpha^{2^{(m-1)}},\alpha^{2^{(m-1)}+1},\ldots,\alpha^{2^{(m-1)}+t-1}.$$

In other words, the self-reciprocating generator polynomial $g(X)_{s-r}$ is

$$g(X)_{s-r} = (X+\alpha^{2^{(m-1)}-t})\ldots(X+\alpha^{2^{(m-1)}-1})(X+\alpha^{2^{(m-1)}})(X+\alpha^{2^{(m-1)}+1})\ldots(X+\alpha^{2^{(m-1)}+t-1}).$$

The greatest advantage of using a self-reciprocating generator polynomial is that our RS encoder and syndrome decoder require less hardware.

Primitive RS codes are non-binary BCH codes and use the $g(X)$ forms as previously shown. Other codes may use a different $g(X)$. Make sure you are using the correct $g(X)$ for the correct code; RS codes require $g(X)_{RS\_code}$.

3.3.2 Code Word Polynomial C(X)

Now since we have determined our generator $g(X)$, we can construct the parity-check $CK(X)$. Then we can append our message field $M(X)$ to it and thus construct our systematic code word $C(X)$. Let us now continue our (15,9) RS example with say $M(X) = 0X^8+0X^7+0X^6+0X^5+0X^4+0X^3+0X^2+\alpha^{11}X+0 = \alpha^{11}X$ which represents the

TABLE 3.3-1. - Continued

number of correctable error patterns (bounded distance decoding):

$$\sum_{i=0}^{t} \binom{n}{i}(P^m-1)^i = \sum_{i=0}^{t} \binom{n}{i} n^i > \binom{n}{t} n^t = (n!/(t!(n-t)!))n^t > 10^{10} \text{ words}$$

From section 1.4.3:

field generator	$F(X) = X^4+X+1$
the primitive element	$\alpha(X) = X = \alpha$

From section 3.3.1:

code word generator	$g(X) = X^6+\alpha^{10}X^5+\alpha^{14}X^4+\alpha^4X^3+\alpha^6X^2+\alpha^9X+\alpha^6$
the code word generator's primitive element	$\alpha^G = \alpha^1 = \alpha$
the first consecutive root of the code word generator	FCR = 1

3.3.1 Generator Polynomial g(X)

To be able to decide what parity-check information CK(X) to append onto our message M(X) we must first determine our generator g(X) for a primitive RS code.

$$g(X)_{RS\_code} = \prod_{i=FCR}^{FCR+2t-1} (X + (\alpha^G)^i)$$

Where FCR is the power of the first consecutive root in g(X) and α^G is any primitive element of F(X). It should be noted that any primitive element α^j does not have to be the same primitive element as the one used to generate the field; i.e., α^G does not need to be $\alpha(X)=X=\alpha$. For our primitive (15,9) RS example, α^G can be any one of the following primitive elements α, α^2, α^4, α^7, α^8, α^{11}, α^{13}, and α^{14}. For our (15,9) RS example, we will select the code word generator's roots to be consecutive powers of $\alpha^G=\alpha^1=\alpha$. We will also arbitrarily start at FCR=1. The first consecutive root in g(X) is $(\alpha^G)^{FCR} = (\alpha^1)^1 = \alpha^1 = \alpha$ which just so happens to be the same as our primitive element $\alpha(X)=X=\alpha$. If we let $\alpha(X) = (\alpha^G)^{FCR}$, then some of the notation burden is simplified:

$$g(X) = \prod_{i=1}^{2t} (X + \alpha^i) = (X + \alpha)(X + \alpha^2)\ldots(X + \alpha^{2t})$$

$$g(X) = \sum_{j=0}^{2t} g_j X^j = X^{2t}+g_{2t-1}X^{2t-1}+\ldots+g_1X+g_0$$

message [0000000α^{11}0] which is the binary symbol sequence "000000000 00000000000000000011100000."

$$\begin{aligned} C(X) &= X^{n-k}M(X) + CK(X) \\ &= X^{n-k}M(X) + (X^{n-k}M(X)) \bmod g(X) \\ &= (X^6)(\alpha^{11}X) + (X^6)(\alpha^{11}X) \bmod g(X) \\ &= (\alpha^{11}X^7) + (\alpha^{11}X^7) \bmod g(X) \end{aligned}$$

The calculation of $\alpha^{11}X^7$ mod g(X) where $g(X) = X^6+\alpha^{10}X^5+\alpha^{14}X^4+\alpha^4X^3+\alpha^6X^2+\alpha^9X+\alpha^6$ is presented as follows:

$$\begin{array}{r|l} & \alpha^{11}X+\alpha^6 + \dfrac{\alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}}{g(X)} \\ X^6+\alpha^{10}X^5+\alpha^{14}X^4+\alpha^4X^3+\alpha^6X^2+\alpha^9X+\alpha^6 & \alpha^{11}X^7 \\ - & \alpha^{11}X^7+\alpha^6X^6+\alpha^{10}X^5+\ X^4+\alpha^2X^3+\alpha^5X^2+\alpha^2X \\ & \alpha^6X^6+\alpha^{10}X^5+\ X^4+\alpha^2X^3+\alpha^5X^2+\alpha^2X \\ & \alpha^6X^6+\alpha\ X^5+\alpha^5X^4+\alpha^{10}X^3+\alpha^{12}X^2+\ X+\alpha^{12} \\ (\alpha^{10}+\alpha)X^5+(1+\alpha^5)X^4+(\alpha^2+\alpha^{10})X^3+(\alpha^5+\alpha^{12})X^2+(\alpha^2+1)X+\alpha^{12} & = \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12} \end{array}$$

$CK(X) = \alpha^{11}X^7 \bmod g(X) = \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$

Just as it's difficult to place the division calculations in this tutorial, it is sometimes difficult and nerve racking to write out all this algebra using the standard notation. Therefore, I suggest either writing a program to make calculations in a finite field or using the following shorthand form as much as possible. So let us just repeat what we have just done, but use this shorthand notation.

The calculation of 11.7 mod g(X) where g(X) = 0.6,10.5,14.4,4.3, 6.2,9.1,6.0 is as follows:

$$\begin{array}{r|l} & 11.1,6.0 + \dfrac{8.5,10.4,4.3,14.2,8.1,12.0}{g(X)} \\ 0.6,10.5,14.4,4.3,6.2,9.1,6.0 & 11.7 \\ & 11.7,6.6,10.5,\ 0.4,\ 2.3,\ 5.2,2.1 \\ & 6.6,10.5,\ 0.4,\ 2.3,\ 5.2,2.1 \\ & 6.6,\ 1.5,\ 5.4,10.3,12.2,0.1,12.0 \\ (10+1).5,(0+5).4,(2+10).3,(5+12).2,(2+0).1,12.0 & = 8.5,10.4,\ 4.3,14.2,8.1,12.0 \end{array}$$

Using either notation produces the same results; CK(X) = 11.7 mod g(X) = 8.5,10.4,4.3,14.2,8.1,12.0.

Continuing with the (15,9) RS code example:

$$\begin{aligned} C(X) &= \alpha^{11}X^7 + CK(X) \\ &= \alpha^{11}X^7 + \alpha^{11}X^7 \bmod g(X) \\ &= \alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12} \end{aligned}$$

Therefore, the code word C(X) for $M(X)=\alpha^{11}X$ is

$C(X) = \alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$.

It is possible to demonstrate that the linearity principle holds for any RS code word; RS codes are the descendants of linear block codes. To show this, assume $M_1(X) = \alpha^5X^3$ and $M_2(X) = \alpha^{11}X$. Linearity is demonstrated if $C_{1+2}(X) = C_1(X) + C_2(X)$. Solving for $C_1(X)$ with $M_1(X) = \alpha^5X^3$:

$$\begin{aligned} C_1(X) &= (X^6)(\alpha^5X^3) + (X^6)(\alpha^5X^3) \bmod g(X) \\ &= \alpha^5X^9 + \alpha^5X^9 \bmod g(X) \\ &= \alpha^5X^9 + \alpha^{13}X^5+\alpha^{12}X^4+\alpha^5X^3+\alpha^{13}X^2+\alpha^2X+\alpha^{12} \end{aligned}$$

$C_2(X)$ is what we calculated earlier for $M_2(X) = M(X) = \alpha^{11}X$:

$$C_2(X) = \alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$$

Solving for $C_{1+2}(X)$ with $M_{1+2}(X) = M_1(X)+M_2(X) = \alpha^5X^3+\alpha^{11}X$:

$$\begin{aligned} C_{1+2}(X) &= (X^6)(\alpha^5X^3+\alpha^{11}X) + (X^6)(\alpha^5X^3+\alpha^{11}X) \bmod g(X) \\ &= (\alpha^5X^9+\alpha^{11}X^7) + (\alpha^5X^9+\alpha^{11}X^7) \bmod g(X) \\ &= \alpha^5X^9+\alpha^{11}X^7 + \alpha^3X^5+\alpha^3X^4+\alpha^8X^3+\alpha^2X^2+X \end{aligned}$$

Now to see if this linearity demonstration works, we need to check that $C_{1+2}(X) = C_1(X) + C_2(X)$.

$$\begin{aligned} C_{1+2}(X) \;?=?\; & C_1(X) + C_2(X) \\ ?=?\; & (\alpha^5X^9 + \alpha^{13}X^5+\alpha^{12}X^4+\alpha^5X^3+\alpha^{13}X^2+\alpha^2X+\alpha^{12}) \\ & +(\alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}) \\ ?=?\; & \alpha^5X^9 + \alpha^{11}X^7 + \alpha^3X^5+ \alpha^3X^4+\alpha^8X^3+ \alpha^2X^2+ X \\ ?=?\; & \alpha^5X^9+\alpha^{11}X^7 + \alpha^3X^5+\alpha^3X^4+\alpha^8X^3+\alpha^2X^2+X \\ ?=?\; & C_1(X) + C_2(X) \quad \text{YES!} \end{aligned}$$

Therefore, since $C_{1+2}(X) = C_1(X) + C_2(X) = \alpha^5X^9+\alpha^{11}X^7 + \alpha^3X^5+\alpha^3X^4+\alpha^8X^3 +\alpha^2X^2+X$ in this example, the linearity principal is demonstrated. This is due to the fact that RS codes are cyclic linear block codes. Cyclic block codes have the characteristic that each of the code words are cyclic shifts of others, but not all others. Also, the sum of any two code words is another code word. Since RS codes are both linear and cyclic, it is useful to incorporate these facts into the hardware design (see appendices A and B).

3.4 SUMMARY

Well, we should now all be semi-experts on RS encoding! We know macroscopically what systematic primitive RS error correction coding is; take your message and append on some specific parity-check to it and send it through the coding channel. Refer

to appendix A to know how to construct the encoder using shift registers.

And now on to Reed-Solomon decoding...

CHAPTER 4
REED-SOLOMON DECODING

Chapter 1 was about GF arithmetic, chapter 2 about block codes, chapter 3 about RS encoding, and now we are going to decode what we encoded in chapter 3. The decoding process is to determine our best estimate of the channel error from a set of unique characteristics which form an identifiable error pattern; these characteristics are known as the syndrome components S_i or collectively known as the syndrome s(X) (or as the syndrome matrix s). After the estimated error E(X)' is subtracted from the received word R(X), our estimate of the transmitted code word C(X)' is determined. We call this estimate the nearest code word C(X)'. Remember, if the error correction capability of the code is not exceeded, the decoder always decodes to the original code word C(X)!

Decoding processes are almost always more difficult to understand and to implement into hardware than encoding processes. There are several devices commercially available today which perform some of the RS codes. However, the decoding process usually requires efficient processing and thus usually requires shift register circuits and/or computer-like circuits. Small, less powerful error correcting codes are fairly easy to accommodate even at high data rates. Larger, more powerful codes require more processing and are a challenge to accommodate sustained high data rates. It is sometimes tough when the decoding of high data rate data must be performed in real time.

THE DECODING PROCESS IS A FIVE-STAGE PROCESS:

1. Calculate the syndrome components from the received word.

2. Calculate the error-locator word from the syndrome components.

3. Calculate the error locations from the error-locator numbers which are from the error-locator word.

4. Calculate the error values from the syndrome components and the error-locator numbers.

5. Calculate the decoded code word from the received word, the error locations, and the error values.

The decoding process is a five-stage process. First calculate what is called the syndrome s(X) or equivalently the 2t syndrome

components S_i. The syndrome can be determined in either of two methods: $S_i = R(\alpha^I)$ $(=E(\alpha^I))$ or $s(X) = R(X) \bmod g(X) = \text{REM}\,[R(X)/g(X)]$ where $S_i = s(\alpha^I)$ from $I = Gi$ from $g(X)_{RS\_code}$ within section 3.3.1 (or appendix D). For our classic example of the (15,9) RS code using $\alpha(X)=X$, FCR=1, and $\alpha^G=\alpha^1$, the 2t syndrome components are simply $S_i = R(\alpha^I)$ $(=E(\alpha^I)) = s(\alpha^I)$ for i=1,2,...,T. From all the S_i calculate the error-locator polynomial $\sigma(X)$; this can be calculated in either of two methods: the linear recursion method or the Berlekamp's (and Massey's) Method for error-locator polynomial $\sigma(X)$. From the error-locator polynomial $\sigma(X)$, first calculate the error-locator numbers z_i for i=1,2,...,T and then calculate the error locations x_i for i=1,2,...,T; this can be calculated in either of two methods: the Chien Search Method or the Explicit Method. From the error-locator numbers z_i and the syndrome components S_i, calculate the error values y_i for i=1,2,...,T; this can also be calculated in either of two methods: the direct method or the error evaluating polynomial method. From the error locations x_i and the error values y_i, the estimate of the error E(X)' [or synonymously the decoded error pattern E(X)'] is specified. Finally, the fifth stage is the determination of the nearest code word C(X)' from R(X) and E(X)'.

Sometimes this five-stage process is thought of as three simple steps: Step 1 is to calculate all of the syndrome components. Step 2 is to determine the error-locator polynomial. Step 3 is to determine the decoded error pattern from the error-locator and the syndrome components and then proceed to correct the errors by subtracting the decoded error pattern from the received word. If one prefers to think of the decoding procedure as three steps, then step 1 = stage 1, step 2 = stage 2, and step 3 = stages 3,4, and 5. Step 2 is the most difficult.

It should be noted that sometimes designers like to design a sixth stage (or a fourth step) into the decoding process. This optional sixth stage is calculating the syndrome of the decoded code word C(X)' to ensure that $S_i=0$ for all i; this guarantees that the decoder's output is indeed a code word. If the $S_i \neq 0$ for all i, then something malfunctioned; we shall not have malfunctions!

4.1 REED-SOLOMON DECODER

At the receiver, if we wanted to, we can immediately check for the presence of errors by calculating the parity-check symbols from the received message symbols and comparing this result with the received parity-check symbols. If the two parity-check symbol patterns match, then either there are zero errors in the received word (which equates to the original code word being decoded) OR the

error pattern was identical to a non-zero code word (which equates to a decoded code word different than the original code word). If errors occurred, we get into the fun of finding out where the errors took place and what the values of the errors are. The errors can be injected in either the message part of the code word $X^{n-k}M(X)$ and/or in the parity-check part of the code word CK(X) where $C(X) = X^{n-k}M(X) + CK(X)$. The symbol locations in the received word R(X) where the errors took place are simply denoted as the error locations x_i. The error values at these symbol locations in R(X) are denoted as the respective error values y_i.

The block diagrams of a RS decoder and the coding channel are shown in figure 4.1-1 and its polynomials' representations are given in table 4.1-1. The block diagram seems simple enough, but how does the decoder determine E(X)' given only R(X)? Well, the decoder actually has to determine the error locations x_i, the error values y_i, and then E(X)' is specified! But then how are the x_i and y_i determined? Well, we will get into that later within this chapter.

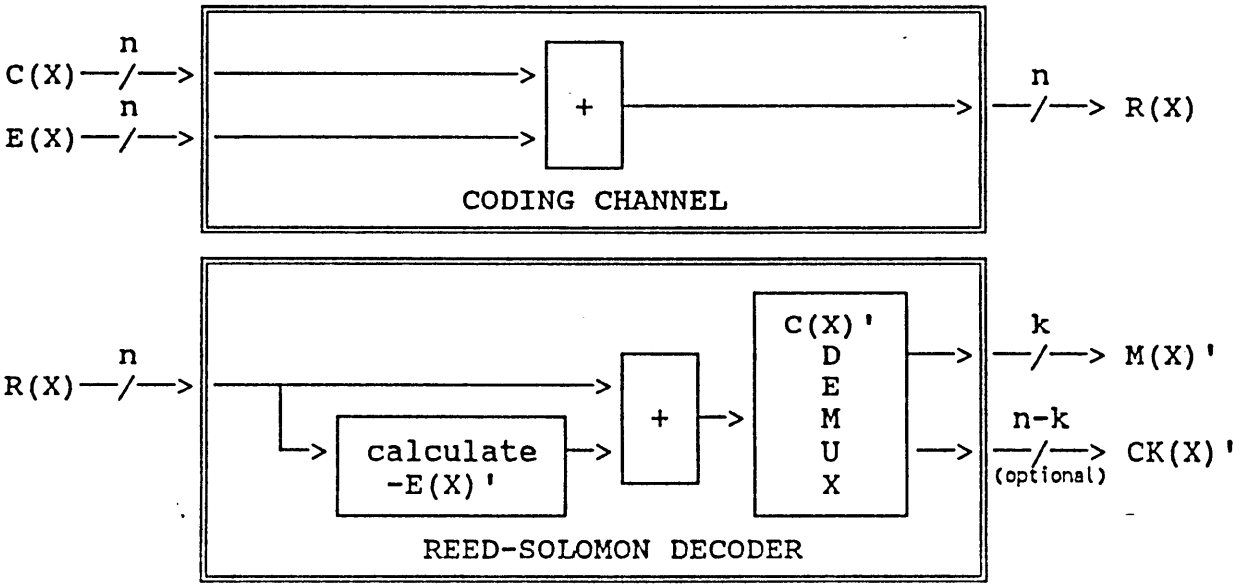


Figure 4.1-1. - Reed-Solomon decoder block diagram.

TABLE 4.1-1. - POLYNOMIAL DEFINITIONS OF A RS DECODER

received word R(X) consisting of received symbols R_i:

$$R(X) = C(X) + E(X)$$
$$= [(C_{n-1}+E_{n-1})X^{n-1} + (C_{n-2}+E_{n-2})X^{n-2} + \ldots + (C_1+E_1)X + (C_0+E_0)]$$
$$= R_{n-1}X^{n-1} + R_{n-2}X^{n-2} + \ldots + R_1X + R_0$$

decode error pattern E(X)' consisting of error locations x_i and error values y_i:

$$E(X)' = y_1x_1 + y_2x_2 + \ldots + y_Tx_T$$

decoded code word C(X)' consisting of code word symbols C_i':

$$C(X)' = R(X) - E(X)' = R(X) + E(X)'$$
$$= X^{n-k}M(X)' + CK(X)'$$
$$= X^{n-k}M(X)' + X^{n-k}M(X)' \bmod g(X)$$
$$= C_{n-1}'X^{n-1} + C_{n-2}'X^{n-2} + \ldots + C_1'X + C_0'$$

decoded message (or data or information) M(X)' consisting of message symbols M_i':

$$M(X)' = C_{n-1}'X^{k-1} + C_{n-2}'X^{k-2} + \ldots + C_{n-k+1}'X + C_{n-k}'$$
$$= M_{k-1}'X^{k-1} + M_{k-2}'X^{k-2} + \ldots + M_1'X + M_0'$$

decoded parity-check CK(X)' consisting of check symbols CK_i':

$$CK(X)' = C_{n-k-1}'X^{n-k-1} + C_{n-k-2}'X^{n-k-2} + \ldots + C_1'X + C_0'$$
$$= CK_{n-k-1}'X^{n-k-1} + CK_{n-k-2}'X^{n-k-2} + \ldots + CK_1'X + CK_0'$$

To determine our best guess of what was transmitted, E(X)' is added [equivalent to GF(2) or $GF(2^m)$ subtraction] to R(X) to correct R(X) into C(X)'. Hopefully C(X)' is C(X); in fact C(X)' is EXACTLY C(X) if the error correction capability is not exceeded!

4.2 SYNDROMES

With $M(X) = \alpha^{11}X$ and $C(X) = \alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$ from section 3.3.2, let us complete the first stage of the (15,9) RS decoding.

Assume some noise was injected into the coding (or communications) channel and as a result some of the binary symbols (or bits) within two code symbols were altered; errors occurred in the X^8 and the X^2 locations.

Suppose $R(X) = \underline{X^8}+\alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\underline{\alpha^3X^2}+\alpha^8X+\alpha^{12}$.

Notice that the coefficients of X^8, part of the RS data source

field [$X^{n-k}M(X)$], and X^2, part of the RS check symbol field [$X^{n-k}M(x)$ mod $g(X)$], has changed from 0 and α^{14} to 1 and α^3 respectfully. This example just so happens to show the right most binary symbol within both of the error symbols as being flipped: vectors (0000) and (1001) to vectors (0001) and (1000) respectfully. REED-SOLOMON CODES NOT ONLY CORRECT SINGLE BIT ERRORS (i.e., binary error symbols) WITHIN A CODE SYMBOL, BUT ALSO ANY NUMBER OF BIT ERRORS WITHIN A SYMBOL! This is why we generally speak of RS codes correcting symbols and not bits; RS codes are burst error correcting codes including some degree of random error correction capability. The output of the RS decoder should strip off these errors from R(X) and result in C(X)'.

The word "syndrome" is defined in a dictionary as a group of signs and symptoms that occur together and characterize a particular abnormality. It is also defined as a set of concurrent things that usually form an identifiable pattern. In the coding application syndrome components S_i are these individual characteristics that characterize a particular error pattern (abnormality). The syndrome s(X) is simply the accumulation of these characteristics where $S_i=s(\alpha^i)$.

The syndrome components S_i can be determined in either of two methods as presented in sections 4.2.1 and 4.2.2.

4.2.1 Method 1: Syndrome Components S_i

$S_i = R(\alpha^I) = R((\alpha^G)^i)$ for i = FCR,FCR+1,...,2t+FCR-1 and for the code word generator primitive element α^G. For our RS example, $S_i=R(\alpha^i)$ for i=1,2,...,2t.

$$R(X) = X^8+\alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^3X^2+\alpha^8X+\alpha^{12}$$

$$\begin{aligned}
S_1 &= R(\alpha) \\
&= (\alpha)^8+\alpha^{11}(\alpha)^7+\alpha^8(\alpha)^5+\alpha^{10}(\alpha)^4+\alpha^4(\alpha)^3+\alpha^3(\alpha)^2+\alpha^8(\alpha)+\alpha^{12} \\
&= \alpha^8 +\alpha^{11}\alpha^7 +\alpha^8\alpha^5 +\alpha^{10}\alpha^4 +\alpha^4\alpha^3 +\alpha^3\alpha^2 +\alpha^8\alpha +\alpha^{12} \\
&= \alpha^8 +\alpha^{18} +\alpha^{13} +\alpha^{14} +\alpha^7 +\alpha^5 +\alpha^9 +\alpha^{12} \\
&= (\alpha^8 +\alpha^3)+(\alpha^{13} +\alpha^{14})+(\alpha^7 +\alpha^5)+(\alpha^9 +\alpha^{12}) \\
&= (\alpha^{13})+(\alpha^2)+(\alpha^{13})+(\alpha^8) \\
&= (\alpha^{13}+\alpha^{13})+(\alpha^2+\alpha^8) \\
&= (0)+(1) \\
&= 1
\end{aligned}$$

$S_2 = R(\alpha^2) = 1$
$S_3 = R(\alpha^3) = \alpha^5$
$S_4 = R(\alpha^4) = 1$
$S_5 = R(\alpha^5) = 0$
$S_6 = R(\alpha^6) = \alpha^{10}$

Therefore, the syndrome components are $S_1=S_2=S_4=1$, $S_3=\alpha^5$, $S_5=0$, and $S_6=\alpha^{10}$. Notice that in this example, i.e., $S_2 = (S_1)^2 = 1$, $S_4 = (S_2)^2 = 1$, $S_6 = (S_3)^2 = \alpha^{10}$, and $S_5 = 0$. $S_{2i} = (S_1)^2$ for RS codes is not the general case; $S_{2i} = (S_1)^2$ occurred because we had a special type of an error pattern. It should also be noted that the syndrome components are not the coefficients of the syndrome polynomial s(X). The syndrome components are of the form $S_i=s(\alpha^i)$ for $\alpha^6=\alpha^1$.

4.2.2 Method 2: Syndrome Polynomial s(X)

The syndrome components S_i can also be found by first determining the syndrome s(X) = REM [R(X)/g(X)] = R(X) mod g(X). This method works for both systematic and non-systematic codes. Then the syndrome components S_i can be found by $S_i = s(\alpha^I)$ for i = FCR,FCR+1,...,FCR+2t-1. For our (15,9) RS example, $S_i = s(\alpha^I)$ for i = 1,2,...,2t. These calculations are left as an exercise! The results of this exercise are the same as the results found in section 4.2.1 and appendix C: $S_1=S_2=S_4=1$, $S_3=\alpha^5$, $S_5=0$, and $S_6=\alpha^{10}$. The S_i and the s(X) can also be calculated using matrices as later presented in appendix C. Remember that in order to ease the hand calculation burden, we can use either the shorthand method of $\alpha^i X^j = i.j$ or write a program to perform the calculations.

4.3 ERROR-LOCATOR POLYNOMIAL $\sigma(X)$

$\sigma_r(X)$ is known as the reciprocal of the error-locator polynomial $\sigma(X)$ where the roots of $\sigma_r(X)$ yield the error-locator (or error-location) numbers z_i. $\sigma(X)$ is known as the error-locator polynomial where the inverse of its roots yield the error-locator numbers z_i. The degree of either $\sigma_r(X)$ or $\sigma(X)$ determines the total number of error symbols T which for non-erasure systems is less than or equal to the error correction capability t. In RS coding, understanding is often easier using $\sigma_r(X)$ to find the z_i rather than $\sigma(X)$.

The syndrome components S_i are known; the error locations x_i and the error values y_i are not known. The S_i are related to the z_i (i.e., also the x_i) and the y_i by the following set of independent

NON-LINEAR simultaneous equations (equation 4.3-1) called the weighted power-sum symmetric functions.

$$S_i = \sum_{j=1}^{T} y_j z_j^{\,i}$$

where i=FCR,FCR+1,...,2t+FCR-1, and where T for non-erasure example is the number of errors $t_E \leq t$. (equation 4.3-1)

In our example, i=1,2,...,T, because FCR=1 is the power of the first consecutive generator root.

Usually there are many solutions to this previous set of NON-LINEAR equations; these solutions are within a distance of t errors. However, there is always only one correct solution within a distance of t symbols; the correct solution is simply the $\sigma_r(X)$ polynomial representing the fewest errors (i.e., the lowest possible value of T) occurring which satisfies equation 4.3-1.

Once $\sigma_r(X)$ is known, the error locations x_i can be determined and then equation 4.3-1 simplifies into a standard set of independent LINEAR simultaneous equations. We all can then solve these LINEAR equations for the y_i!

Berlekamp came up with an efficient iterative algorithm denoted appropriately enough, Berlekamp's iterative algorithm for finding the error-locator polynomial $\sigma(X)$. If you can come up with a better algorithm than Berlekamp's iterative algorithm to find the solution with the fewest terms (or errors) out of a set of many solutions growing exponentially as code length increases, you might become a millionaire!!!

Appendix D presents more details of the link between the error-locator polynomial and the weighted power-sum symmetric functions.

There are two methods we can implement to solve for $\sigma(X)$: Berlekamp's algorithm for $\sigma(X)$ presented in sections 4.3.1.1 and 4.3.1.2 and the linear recursion method for $\sigma(X)$ presented in section 4.3.2.

4.3.1 Method 1: Iterative Algorithm for $\sigma(X)$

There are two equivalent presentations of Berlekamp's algorithm: Berlekamp's algorithm presentation for finding $\sigma(X)$ and the Euclidean division algorithm presentation for finding $\sigma(X)$. One of the most common ways to present Berlekamp's algorithm is in a table format and will be denoted as Berlekamp's algorithm presentation. Berlekamp's algorithm presentation is a little simpler to follow;

the Euclidean or greatest common division (GCD) presentation is a little easier to intuitively understand. Regardless of the presentation style, the algorithms determine the $\sigma(X)$ polynomial with the least number of terms which is the single solution linked to a set of simultaneous NON-LINEAR equations. Since there is more than one possible solution to equation 4.3-1, we use MLD and pick the link $\sigma_r(X)$ with the lowest degree. $\sigma_r(X)$ transforms equation 4.3-1 into a set of solvable LINEAR equations always possessing a single solution. This single solution is simply the decoded error pattern E(X)'.

4.3.1.1 Berlekamp's Algorithm Presentation

Table 4.3.1-1 presents the most common presentation in describing the Berlekamp's iterative algorithm. Following this, table 4.3.1.1-2 is then developed for our primitive (15,9) RS example.

TABLE 4.3.1.1-1. - BERLEKAMP'S ITERATIVE RS ALGORITHM FOR FCR=1

μ	$\sigma^{(\mu)}(X)$	d_μ	h_μ	$\mu-h_\mu$
-1	1	1	0	-1
0	1	$S_{FCR}=S_1$	0	0
1	...	...	...	...
...	...	...	...	...
...	...	...	...	...
2t	$\sigma(X)$	---	---	---

PROCEDURE TO FILL IN TABLE 4.3.1.1-1:

1. If $d_\mu = 0$, then $\sigma^{(\mu+1)}(X) = \sigma^{(\mu)}(X)$ and $h_{\mu+1}=h_\mu$.

2. If $d_\mu \neq 0$, then find a row before the μth row, call it the ρth row, such that $\rho-h_\rho$ has the largest value in its column before the μth row (ρ may be one of several values) and $d_\rho \neq 0$ and then:

$$\sigma^{(\mu+1)}(X) = \sigma^{(\mu)}(X) + d_\mu d_\rho^{-1} X^{(\mu-\rho)} \sigma^{(\rho)}(X)$$
$$h_{\mu+1} = \text{MAX}\,[h_\mu,\ h_\rho+\mu-\rho]$$

3. In either case:

$$d_{\mu+1} = S_{\mu+2} + \sigma_1^{(\mu+1)} S_{\mu+1} + \ldots + \sigma_{h_{(\mu+1)}}^{(\mu+1)} S_{\mu+2-h_{(\mu+1)}}$$

where $\sigma_i^{(\mu+1)}$ are the coefficients of $\sigma^{(\mu+1)}(X) = 1 + \sigma_1^{(\mu+1)}X + \sigma_2^{(\mu+1)}X^2 + \ldots + \sigma_{h(\mu+1)}^{(\mu+1)}X^{h(\mu+1)}$

The following are a few choice calculations used in developing table 4.3.1.1-2 for our primitive (15,9) RS example.

At row μ=1, the value μ=0:

$d_0 = S_{FCR} = S_1 = 1$
$\rho = -1$
$\sigma^{(1)}(X) = \sigma^{(0)}(X) + d_0(d_{-1})^{-1}X^{(0-(-1))}\sigma^{(-1)}(X) = 1 + (1)(1)^{-1}(X^1)(1) = 1 + X$

Continuing at row μ=1 with the value μ=0:

$h_1 = MAX\ [h_0, h_{-1} + 0 - (-1)] = MAX\ [0, 0 + 0 + 1] = MAX\ [0, 1] = 1$
$d_1 = S_2 + \sigma_1^{(1)}S_1 = 1 + (1)(1) = 1 + 1 = 0$

At row μ=2, the value μ=1:

$d_1 = 0$
$\sigma^2(X) = \sigma^1(X) = 1 + X$
$h_2 = h_1 = 1$
$d_2 = S_3 + \sigma_1^2 S_2 = \alpha^5 + (1)(1) = \alpha^5 + 1 = \alpha^{10}$

And so on, and so on, and so on, until $\sigma(X)$ is obtained:

$\sigma(X) = \sigma^{2t}(X) = \sigma^6(X) = 1 + X + \alpha^{10}X^2 = \alpha^{10}X^2+X+1.$

Note that the calculations for $\sigma^{(\mu)}(X)$ and d_μ use finite field math and the calculations for h_μ and $\mu-h_\mu$ use infinite field math. The results are presented in table 4.3.1.1-2.

TABLE 4.3.1.1-2. - EXAMPLE OF THE BERLEKAMP'S ITERATIVE ALGORITHM

μ	$\sigma^{(\mu)}(X)$	d_μ	h_μ	$\mu-h_\mu$	
-1	1	1	0	-1	
0	1	$S_{FCR}=S_1=1$	0	0	
1	$1+X$	0	1	0	(pick ρ = -1)
2	$1+X$	α^{10}	1	1	
3	$1+X+\alpha^{10}X^2$	0	2	1	(pick ρ = 0)
4	$1+X+\alpha^{10}X^2$	0	2	2	
5	$1+X+\alpha^{10}X^2$	0	2	3	
2t=6	$1+X+\alpha^{10}X^2$	---	---	---	

Therefore, the error-locator polynomial $\sigma(X) = \alpha^{10}X^2+X+1$. Then, $\sigma_1=1$ and $\sigma_2=\alpha^{10}$ are from $\sigma(X) = \alpha^{10}X^2+X+1 = \sigma_2X^2+\sigma_1X+\sigma_0$. These σ_i are the coefficients to the error-locator polynomial $\sigma(X)$ and its reciprocal $\sigma_r(X)$. $\sigma(X)$ and $\sigma_r(X)$ are related by the following:

$$\sigma_r(X) = X^T\sigma(X^{-1}) = X^2(1+X^{-1}+\alpha^{10}X^{-2}) = X^2 + X + \alpha^{10}$$

Therefore, the error-locator polynomial reciprocal $\sigma_r(X) = X^2+X+\alpha^{10}$ and the error-locator polynomial $\sigma(X) = \alpha^{10}X^2+X+1$.

We have completed Berlekamp's algorithm. In the next section this same iterative algorithm is presented in a different way; I believe them to be the same algorithm.

4.3.1.2 Euclidean Division Algorithm Presentation

Let $S(X) = \sum_{i=1}^{2t} S_iX^{i-1} = S_1 + S_2X + \ldots + S_5X^4 + S_6X^5 = \alpha^{10}X^5+X^3+\alpha^5X^2+X+1$

Divide X^{2t} by $S(X)$, then $S(X)$ by r_1, r_1 by r_2, r_2 by r_3,..., until the degree of $r_i \leq t$:

Divide X^6 by $S(X)$:

$$\begin{array}{r|l} \alpha^{10}X^5+X^3+\alpha^5X^2+X+1 & \alpha^5X + 0 + \dfrac{\alpha^5X^4+\alpha^{10}X^3+\alpha^5X^2+\alpha^5X}{S(X)} \\ \hline X^6 & \\ \underline{X^6+\alpha^5X^4+\alpha^{10}X^3+\alpha^5X^2+\alpha^5X} & \\ \alpha^5X^4+\alpha^{10}X^3+\alpha^5X^2+\alpha^5X & \end{array}$$

$$\frac{X^6}{S(X)} = \alpha^5 X + \frac{\alpha^5 X^4+\alpha^{10}X^3+\alpha^5 X^2+\alpha^5 X}{S(X)}$$

$$X^6 = (\alpha^5 X)S(X) + (\alpha^5 X^4+\alpha^{10}X^3+\alpha^5 X^2+\alpha^5 X)$$
$$= (q_1\ )S(X) + (\qquad r_1 \qquad)$$

KEEP GOING!!! The degree of $r_1=4 > t=3$.

Divide S(X) by r_1:

$$\frac{S(X)}{r_1} = \alpha^5 X + \alpha^{10} + \frac{1}{r_1}$$

$$S(X) = -(\alpha^5 X + \alpha^{10})r_1 + 1$$
$$= (\quad q_2 \quad)r_1 + r_2$$

STOP!!! The degree of $r_2=0 \leq t=3$.

Put the previous results into the following form:

$$S(X)\sigma(X) = A(X) + X^{2t}B(X)$$

A summary of the previous results is:

$$r_1 = X^6 + q_1 S(X)$$
$$r_2 = S(X) + q_2 r_1$$

Combining to form one equation:

$$r_2 = S(X) + q_2[X^6 + q_1 S(X)]$$
$$= X^6[q_2] + S(X)[1 + q_1 q_2]$$

Substituting values:

$$1 = X^6[\alpha^5 X + \alpha^{10}] + S(X)[1 + (\alpha^5 X)(\alpha^5 X + \alpha^{10})]$$
$$= X^6[\alpha^5 X + \alpha^{10}] + S(X)[\alpha^{10}X^2 + X + 1]$$

Put in proper form of $S(X)\sigma(X) = A(X) + X^{2t}B(X)$.

$$S(X)[\alpha^{10}X^2 + X + 1] = 1 + X^6[\alpha^5 X + \alpha^{10}]$$
$$S(X)[\quad \sigma(X) \quad] = 1 + X^6[\alpha^5 X + \alpha^{10}]$$

Therefore, the error-locator $\sigma(X) = \alpha^{10}X^2+X+1 = \sigma_2 X^2+\sigma_1 X+1$.

Therefore, $\sigma_1=1$ and $\sigma_2=a^{10}$.

$$\sigma_r(X) = X^r\sigma(X^{-1}) = X^2(1+X^{-1}+\alpha^{10}X^{-2}) = X^2 + X + \alpha^{10}$$

Therefore, the error-locator polynomial reciprocal $\sigma_r(X) = X^2+X+\alpha^{10}$ using the Euclidean greatest common divisor algorithm. Notice that the same $\sigma_r(X)$ was correctly obtained as in section 4.3.1.1. Maybe it is really possible to solve a set of NON-LINEAR equations for the "correct" single solution!

4.3.2 Method 2: Linear Recursion Method for $\sigma(X)$

The following is how we use the linear recursion method to solve a set of simultaneous linear equations for $\sigma_r(X)$. The following equation is derived in appendix D.

$$S_i = \sum_{j=0}^{T-1} - S_{i+j-T}\sigma_{T-j} \qquad \text{for } i = T+FCR, T+FCR+1, \ldots, 2t+FCR-1$$

This is simply $S_i = S_{i-T}\sigma_T + S_{i-T+1}\sigma_{T-1} + \ldots + S_{i-1}\sigma_1$ for $i=T+FCR, T+FCR+1, \ldots, 2t+FCR-1$.

Our non-erasure example has $T = t_E \leq t$ unknown error symbols to solve for and FCR=1. In this example, t=3 error symbols so $T \leq 3$ error symbols.

First, we assume that T=0 errors have occurred. For our example, T>0 errors because at least one S_i was nonzero.

Next, we assume that T=1 errors have occurred. We then obtain the following set of equations from the previous general form:

$$S_i = \sum_{j=0}^{0} S_{i+j-1}\sigma_{1-j} = S_{i-1}\sigma_1 \qquad \text{for } i = T+1, T+2, \ldots, 2t$$

When these equations are completely written out they look like this: $S_2 = S_1\sigma_1$, $S_3 = S_2\sigma_1$, $S_4 = S_3\sigma_1$, $S_5 = S_4\sigma_1$, and $S_6 = S_5\sigma_1$. We notice that these equations are insoluble and therefore T>1. For example, if $S_2 = S_1\sigma_1 = \alpha^0\sigma_1 = \alpha^0$, then $\sigma_1 = \alpha^0$; but this cannot be since $S_3 = S_2\sigma_1 = \alpha^0\alpha^0 = \alpha^0 \neq \alpha^5$.

Since T=0 and then T=1 did not solve the equations, then try T=2. This results in trying to solve the following set of equations:

$$S_i = \sum_{j=0}^{1} S_{i+j-2}\sigma_{2-j} \qquad \text{for } i = 3,4,5,6$$

These set of equations result when T=2.

$$S_1\sigma_2 + S_2\sigma_1 = S_3$$
$$S_2\sigma_2 + S_3\sigma_1 = S_4$$
$$S_3\sigma_2 + S_4\sigma_1 = S_5$$
$$S_4\sigma_2 + S_5\sigma_1 = S_6$$

It turns out that it is possible to pick any T=2 equations to solve for the T=2 unknown σ_i values. Let us just pick the first two equations from the preceding set of equations.

$$S_1\sigma_2 + S_2\sigma_1 = S_3$$
$$S_2\sigma_2 + S_3\sigma_1 = S_4$$

To determine whether T=2 errors have occurred, we can at this point simply calculate the determinant of these T=2 equations. If the determinant is non-zero, then T=2 errors have occurred. Otherwise, we should continue this process for increasing T until T≤t.

$$\text{Let } |S\sigma|_2 = \begin{vmatrix} S_1 & S_2 \\ S_2 & S_3 \end{vmatrix}$$

Check for a non-zero determinant of $|S\sigma|_T$; DET $|S\sigma|_T$ is a scalar magnitude called the determinant.

Next, we substitute the syndrome components S_i values from section 4.2.1 to obtain the following results.

$$\sigma_2 + \sigma_1 = \alpha^5$$
$$\sigma_2 + \alpha^5\sigma_1 = 1$$

Now check for a non-zero determinant of $|S\sigma|_2$.

$$\text{DET } |S\sigma|_2 = \text{DET} \begin{vmatrix} S_1 & S_2 \\ S_2 & S_3 \end{vmatrix} = \text{DET} \begin{vmatrix} 1 & 1 \\ 1 & \alpha^5 \end{vmatrix} = \alpha^5 + 1 = \alpha^{10}$$

The determinant is not equal to zero so therefore T=2 error symbols. Remember that in a non-erasure example T≤t. Now solve for the values of σ_i for i = 1,2,...,T = 1,2.

$$\sigma_1 = \frac{\text{DET} \begin{vmatrix} 1 & \alpha^5 \\ 1 & 1 \end{vmatrix}}{\text{DET } |S\sigma|_2} = \frac{1 + \alpha^5}{\alpha^{10}} = \frac{\alpha^{10}}{\alpha^{10}} = 1$$

$$\sigma_2 = \frac{\text{DET} \begin{vmatrix} \alpha^5 & 1 \\ 1 & \alpha^5 \end{vmatrix}}{\text{DET } |S\sigma|_2} = \frac{\alpha^{10} + 1}{\alpha^{10}} = \frac{\alpha^5}{\alpha^{10}} = \alpha^{-5} = \alpha^{10}$$

We can construct the error-locator or its reciprocal at this point. The reciprocal of $\sigma(X)$ is shown below.

$$\sigma_r(X) = \sum_{i=0}^{i=T} \sigma_i X^{T-i} \qquad \text{with } \sigma_0 = 1$$

$$\sigma_r(X) = X^2 + \sigma_1 X + \sigma_2 = X^2 + X + \alpha^{10}$$

Therefore, the error-locator polynomial reciprocal $\sigma_r(X) = X^2+X+\alpha^{10}$ and T=2 errors have occurred. Again we receive the same results as in sections 4.3.1.1 and 4.3.1.2. See appendix B for a hardware design using S_i to determine σ_j.

4.4 ERROR LOCATIONS x_i

We find the error locations x_i by first finding the error-locator numbers z_i. The z_i are the roots of $\sigma_r(X)$ or are the inverse of the roots of $\sigma(X)$. $\sigma(X)$ is as easy to determine as $\sigma_r(X)$; once one is determined, the other is also determined. I prefer to use $\sigma_r(X)$ instead of $\sigma(X)$ for two reasons: The first is to demonstrate that there is usually more than one way to solve a problem; some solutions are more easily applied and/or implemented and/or calculated than others. We must always be on the alert for better ways to do the same thing. The second is that $\sigma_r(X)$ is more clearly defined from the error locations (see appendix D).

There are two equivalent algorithms to find the error locations x_i. They are the Chien search and explicit factorization. They both find the roots in the error-locator.

4.4.1 Method 1: Chien Search

The following is how we perform what is commonly called the Chien search. The Chien search calculates the outputs for all the possible inputs; it is a very simple, brute force, search algorithm. The Chien search determines the roots of either the error-locator polynomial or of its reciprocal. The roots of the reciprocal of the error-locator polynomial $\sigma_r(X)$ are the error-locator numbers z_i.

$$\sigma_r(X) = \prod_{i=1}^{i=T} (X + z_i) \qquad \text{for } z_i \text{ in the form of } \alpha^j$$

Find the roots of $\sigma_r(X)$, i.e., determine $\sigma_r(X)=0$ for

$X = 1, \alpha, \alpha^2, \ldots, \alpha^{n-1}$.

$\sigma_r(X) = X^2+X+\alpha^{10}$

$\sigma_r(1) = (1)^2 + (1) + \alpha^{10} = 1 + \alpha^5 = \alpha^{10}$
$\sigma_r(\alpha) = (\alpha)^2 + (\alpha) + \alpha^{10} = \alpha^2 + \alpha^8 = 1$
$\sigma_r(\alpha^2) = (\alpha^2)^2 + (\alpha^2) + \alpha^{10} = \alpha^4 + \alpha^4 = 0$
$\sigma_r(\alpha^3) = \alpha^6 + \alpha^{12} = \alpha^4$
$\sigma_r(\alpha^4) = \alpha^8 + \alpha^2 = 1$
$\sigma_r(\alpha^5) = \alpha^{10} + 1 = \alpha^5$
$\sigma_r(\alpha^6) = \alpha^{12} + \alpha^7 = \alpha^2$
$\sigma_r(\alpha^7) = \alpha^{14} + \alpha^6 = \alpha^8$
$\sigma_r(\alpha^8) = \alpha + \alpha = 0$

The two error-locator (or error-location) numbers are $z_1=\alpha^2$ and $z_2=\alpha^8$. It does not matter how the z_i for $i = 1,2,\ldots,2t$ are assigned to the roots of $\sigma_r(X)$. I prefer lower indices to correspond to the lower locations in consecutive order and higher indices to higher locations, e.g., I prefer $z_1=\alpha^2$ and $z_2=\alpha^8$ over $z_1=\alpha^8$ and $z_2=\alpha^2$.

The error locations x_i are defined from the error-locator numbers z_i as $x_i = X\text{^}((\log_\alpha z_i)/G)$. Since in our example $\alpha^G=\alpha^1$, then $x_1 = X\text{^}((\log_\alpha z_1)/G) = X\text{^}((\log_\alpha \alpha^2)/1) = X\text{^}(2/1) = X\text{^}(2) = X^2$ and $x_2 = X\text{^}((\log_\alpha z_2)/G) = X^8$.

Since $\sigma_r(X)$ in this example is a polynomial of degree two, it has two and only two unique roots [e.g., $X^2+X+\alpha^{10} = (X+\alpha^2)(X+\alpha^8)$ because $\sigma_r(\alpha^2)=\sigma_r(\alpha^8)=0$]. Therefore, there is no need to solve for the remaining roots; all roots have been found at this point. However, for the sake of completing the Chien search, the calculations for the remaining roots are computed.

$\sigma_r(\alpha^9) = \alpha^8$
$\sigma_r(\alpha^{10}) = \alpha^5$
$\sigma_r(\alpha^{11}) = \alpha$
$\sigma_r(\alpha^{12}) = \alpha$
$\sigma_r(\alpha^{13}) = \alpha^2$
$\sigma_r(\alpha^{14}) = \alpha^4$

Notice that $\sigma_r(0)=\sigma_r(\alpha^{-\infty})$ is never calculated because there are only $n = 2^m-1$ locations in a primitive code. The locations are denoted $X^0=1, X^1=X, X^2, X^3,\ldots,X^{n-1} = 1, X, X^2, X^3,\ldots,X^{n-1}$; in this example, there is no $X^{-\infty}=0$=null position.

From section 4.2 the received word is repeated with the error locations underlined from how we designed our example. These same locations were also able to be determined by the Chien search.

$$R(X) = \underline{X^8}+\alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^3\underline{X^2}+\alpha^8X+\alpha^{12}$$

Error location determined by the Chien search

Error location determined by the Chien search

This section's Chien search results of $x_1=X^2$ and $x_2=X^8$ checks with the code word and the received word from section 4.2.

4.4.2 Method 2: Explicit Factorization

The basic idea here is to change variables in $\sigma(X)$ or $\sigma_r(X)$ so as to change $\sigma(X)$ or $\sigma_r(X)$ into a "standard" form whose factorization is stored in a look up table. These techniques work well for low values of t. In fact, this may even be faster than the Chien search for low values of t. More information that discusses techniques of variable substitution to simplify polynomials can be found in coding and mathematics literature.

4.5 ERROR VALUES y_i

There are two methods that I will present here: the direct solution and the error evaluator polynomial method.

4.5.1 Method 1: Direct Solution

Since T, x_i, and S_i are known, we can solve the set of simultaneous linear equations for y_i. The following equation is repeated from Section 4.3. Reducing the following set of simultaneous NON-LINEAR equations into a LINEAR set, the error values y_i can then be determined.

$$S_i = \sum_{j=1}^{T} y_j z_j^i$$

where i = FCR,FCR+1, ...,2t+FCR-1 and where T for a non-erasure example is the number of errors $t_E \leq t$. (equation 4.3-1)

For our (15,9) RS code example, these weighted power-sum symmetric functions reduce to:

$$S_i = \sum_{j=1}^{2} y_j z_j^i$$

where i=1,2,...,6 and where $T=t_E=2 \leq t=3$ as determined in Section 4.3.

Use equation 4.3-1 to construct the following set of 2t non-linear equations:

$$
\begin{aligned}
y_1(z_1) + y_2(z_2) &= S_1 \\
y_1(z_1)^2 + y_2(z_2)^2 &= S_2 \\
y_1(z_1)^3 + y_2(z_2)^3 &= S_3 \\
y_1(z_1)^4 + y_2(z_2)^4 &= S_4 \\
y_1(z_1)^5 + y_2(z_2)^5 &= S_5 \\
y_1(z_1)^6 + y_2(z_2)^6 &= S_6
\end{aligned}
$$

We only need T=2 of these equations since we have T=2 unknowns (i.e., y_1 and y_2). Pick two equations; I will pick the first two since they look the easiest:

$$
\begin{aligned}
y_1(z_1) + y_2(z_2) &= S_1 \\
y_1(z_1)^2 + y_2(z_2)^2 &= S_2
\end{aligned}
$$

Substitute $z_1=\alpha^2$ and $z_2=\alpha^8$ to transform the preceding set into the following linear set.

$$
\begin{aligned}
y_1(\alpha^2) + y_2(\alpha^8) &= 1 \\
y_1(\alpha^2)^2 + y_2(\alpha^8)^2 &= 1
\end{aligned}
$$

This simplifies to:

$$
\begin{aligned}
y_1\alpha^2 + y_2\alpha^8 &= 1 \\
y_1\alpha^4 + y_2\alpha &= 1
\end{aligned}
$$

Now use the standard Cramer's rule to solve for the coefficients.

$$
\text{DET } |Y| = \text{DET} \begin{vmatrix} \alpha^2 & \alpha^8 \\ \alpha^4 & \alpha \end{vmatrix} = \alpha^3 + \alpha^{12} = \alpha^{10}
$$

$$
y_1 = \frac{\text{DET} \begin{vmatrix} \alpha^8 & 1 \\ \alpha & 1 \end{vmatrix}}{\text{DET } |Y|} = \frac{\alpha^8 + \alpha}{\alpha^{10}} = \frac{\alpha^{10}}{\alpha^{10}} = \alpha^0 = 1
$$

$$
y_2 = \frac{\text{DET} \begin{vmatrix} \alpha^2 & 1 \\ \alpha^4 & 1 \end{vmatrix}}{\text{DET } |Y|} = \frac{\alpha^2 + \alpha^4}{\alpha^{10}} = \frac{\alpha^{10}}{\alpha^{10}} = \alpha^0 = 1
$$

Therefore, $y_1=1$ and $y_2=1$. The first and second error values y_1 and y_2 just so happen to be the same; $y_1=y_2=\alpha^0=1$. It should also be noted that $y_i = 1,\alpha,\alpha^2,\ldots,\alpha^{n-1}$ are all valid error values y_i. Notice that these error values check with the discussion of the symbols in error within section 4.2. We got the correct decoding

4.5.2.2 Hardware Error Evaluator Polynomial

This section describes an algorithm which is sometimes used in implementations.

$$y_i = \frac{\sum_{j=0}^{T-1} S_{T-j}\sigma_{i,j}}{z_i \prod_{j=1}^{T} (z_i + z_j)} \qquad \text{for } i = 1,2,\ldots,T \text{ and for } i \neq j \text{ in the denominator}$$

Where $\sigma_{i,j}$ is defined by:

$$\prod_{j=0}^{T-1} (X + z_j) \qquad \text{for } i \neq j$$

********** OR **********

$$\sum_{j=0}^{T-1} \sigma_{i,j} X^{T-1-j} \qquad \text{for i can= j}$$

Solving for y_1:

$$y_1 = \frac{S_2\sigma_{1,0} + S_1\sigma_{1,1}}{z_1(z_1 + z_2)}$$

$\sigma_{1,j}$: $X + z_2 = \underline{1}X + \underline{z}_2 = \sigma_{1,0}X + \sigma_{1,1}$

Therefore, $\sigma_{1,0}=1$ and $\sigma_{1,1}=z_2=\alpha^8$. Now substitute $\sigma_{1,0}=S_1=S_2=1$ and $\sigma_{1,1}=\alpha^8$ into the preceding y_1 equation.

$$y_1 = \frac{(1)(1) + (1)(\alpha^8)}{\alpha^2(\alpha^2 + \alpha^8)} = \frac{1 + \alpha^8}{\alpha^4 + \alpha^{10}} = \frac{\alpha^2}{\alpha^2} = \alpha^0 = 1$$

Solving for y_2:

$$y_2 = \frac{S_2e_{2,0} + S_1\sigma_{2,1}}{z_2(z_2 + z_1)}$$

$\sigma_{1,j}$: $X + z_1 = \sigma_{2,0}X + \sigma_{2,1}$

Therefore, $\sigma_{2,0}=1$ αnd $\sigma_{2,1}=z_1=\alpha^2$.

$$y_2 = \frac{(1)(1) + (1)(\alpha^2)}{\alpha^8(\alpha^8 + \alpha^2)} = \frac{1 + \alpha^2}{\alpha + \alpha^{10}} = \frac{\alpha^8}{\alpha^8} = \alpha^0 = 1$$

Therefore, $y_1=1$ and $y_2=1$.

Notice that we obtained the same error values y_i as in the first description of the direct method (sections 4.5.1 and 4.2).

Another possible solution for the y_i is to solve the y_i and $\sigma_{i,j}$ equations for T=3 and then let $z_3=0$ to receive the case of T=2. Doing so will result in the following:

$$y_1 = \frac{S_3 + z_2S_2}{z_1^2(z_1 + z_2)} = \frac{\alpha^5 + (\alpha^8)(1)}{\alpha^4(\alpha^2 + \alpha^8)} = \frac{\alpha^5 + \alpha^8}{\alpha^4} = \frac{\alpha^4}{\alpha^4} = \alpha^0 = 1$$

$$y_2 = \frac{S_3 + z_1S_2}{z_2^2(z_1 + z_2)} = \frac{\alpha^5 + (\alpha^2)(1)}{\alpha(\alpha^2 + \alpha^8)} = \frac{\alpha^5 + \alpha^2}{\alpha} = \frac{\alpha}{\alpha} = \alpha^0 = 1$$

Therefore, $y_1=1$ and $y_2=1$.

Still we obtain the same error values y_i (see sections 4.2, 4.5.1, 4.5.2.1, and the first part of this section).

4.6 DECODED CODE WORD C(X)'

Now since we have already calculated all our error locations x_i and error values y_i, we have just finished constructing our estimate of the error E(X)' which is simply the estimate of the noise.

$$E(X)' = \sum_{i=1}^{T} y_i x_i = y_1x_1 + y_2x_2 = (1)(X^2) + (1)(X^8) = X^2 + X^8 = X^8 + X^2$$

Notice that since $y_1=y_2=\ldots=y_T=1$ within the main (15,9) RS example for our particular R(X), E(X)' just so happened to be a binary polynomial. $y_i = 1,\alpha,\alpha^2,\ldots,\alpha^{n-1}$ and thus E(X)' is always a polynomial with code symbol coefficients.

The received vector is a function of C(X) and E(X); R(X) = C(X) + E(X). Since the decoder only has the received vector R(X) and since it can calculate our estimate of the error E(X)', the closest code word C(X)' can be determined as a function of R(X) and E(X)'.

$$C(X)' = R(X) - E(X)' = R(X) + E(X)'$$

Therefore, we find our closest code word by adding (equivalent to mod-2 subtracting) our estimate of the noise to the received block of data.

$$C(X)' = R(X) + E(X)'$$
$$= [X^8+\alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^3X^2+\alpha^8X+\alpha^{12}] + [X^8 + X^2]$$
$$= \underline{(1+1)X^8}+\alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\underline{(\alpha^3+1)X^2}+\alpha^8X+\alpha^{12}$$
$$= \alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$$

C(X)' is indeed the same as C(X) in this example!!!

$$C(X)' = \alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$$

From Section 4.2 the following code word was transmitted.

$C(X) = \alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$. Therefore, C(X)'=C(X) as expected!

Now strip the message field from the corrected data.

$$M(X)' = C_{n-1}'X^{k-1} + C_{n-2}'X^{k-2} + \ldots + C_{n-k+1}'X + C_{n-k}'$$
$$= \alpha^{11}X$$

The message M(X) transmitted systematically within the code word C(X) is $M(X) = \alpha^{11}X$. Therefore, M(X)'=M(X) as expected.

Now strip the check field from the corrected data.

$$CK(X)' = C_{n-k-1}'X^{n-k-1} + C_{n-k-2}'X^{n-k-2} + \ldots + C_1'X + C_0'$$
$$= \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$$

The check CK(X) transmitted systematically within the code word C(X) is $CK(X) = \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$. Therefore, CK(X)'=CK(X) as expected.

Notice that since there was definitely $T \leq t$ actual error symbols, our decoding process actually picked the same code word that was transmitted. The message M(X)' is then systematically extracted from the decoded code word C(X)'. The parity-check CK(X)' can also be systematically extracted if needed. In many applications the decoded parity-check CK(X)' is usually thrown away or is not even determined! Who needs it? Maybe someone might need the received (or played back) word R(X), but who needs CK(X)'? The C(X)' results demonstrate the correct decoding of a Reed-Solomon code. It works! It really works!

4.7 SUMMARY

In chapter 4 our RS code is demonstrated to be able to be decoded. This is because we are able to take our original message

symbols M(X), append parity-check symbols CK(X) to it, send it, corrupt it with errors E(X), receive it as R(X), and then decode R(X) back into our original M(X) if T≤t. We were able to demonstrate a simple example of a mk = 36 bit message (i.e., the binary sequence "000000000000000000000000000011100000"), encode the message by systematically adding redundancy to develop a mn = 60 bit code word (i.e., the binary sequence "000000000000000000000000000001110000001010111001110010101111111"), transmit (or record) it, allow errors to corrupt it, receive (or play back) it as a mn = 60 bit received word (i.e., the binary sequence "000000000000000000000000000001111100000010101110011100001011111"), and finally to decode the received word back to the exact original 36 bit message. This (15,9) RS example could correct a maximum of 3 code symbols out of 15 code symbols; this code can withstand a maximum symbol error rate SER_{max} (or simply block correction BC=t/n) of t/n = 20%. It is equivalent (but awkward) to indicate that this (15,9) RS example could correct up to a maximum of any where between 3 and 12 bits out of 60 bits depending where the bit errors occur. Usually this code is said to be able to withstand a SER = 20%.

If we feel good about the RS coding operations so far, then let us move onto chapter 5. Otherwise, we should reread the previous chapters. This coding stuff is fun!

CHAPTER 5
SYMBOL ERASING AND REED-SOLOMON CODING

We should now know the Reed-Solomon encoding and decoding operations which were presented in the previous chapters. Knowing this, let us continue and instead of assuming hard decision at the decoder, let us assume soft decision. If we have the soft decision capability, we can come up with an algorithm to pick a few of the worst quality symbols within each block length as the ones which might be in error. Since we have such flags in many systems (i.e., E_b levels, signal format violations, etc.) soft decision is quite possible. Also as n increases, it becomes easier to correctly determine at least some of these highly likely error symbols. By somewhat knowing some of the error locations through soft decision capabilities, the decoder can now work on not only $T \leq t$ errors, but also $T > t$ errors! The overall error correction capability increases!

A correctly designed decoder using soft decision (e.g., using the symbol erasing capability of RS codes) should yield better performance than running the data only through a RS decoder without soft decision.

THE DECODING PROCESS USING SYMBOL ERASING IS A THREE-STEP PROCESS:

1. Calculate the syndrome components and the modified syndrome components.

2. Calculate the error-locator word for the error locations.

3. Determine the decoded error pattern and correct the errors in the received word.

5.1 RS CODING USING SYMBOL ERASURE

If the demodulator within the coding system flags certain received code symbols R_i as unreliable, then these symbols are treated as erasure symbols; AN ERASURE SYMBOL IS DEFINED AS AN ERROR SYMBOL whose erasure location x_i" is known to a high degree of probability from the demodulator (and therefore the corresponding erasure value y_i" is known to a low degree of probability from the demodulator). If we accidentally flag an error-free symbol as being an erasure symbol, the soft decision process can still decode properly! We can pass any erasure symbol R_i" to the decoder and the soft decision process can also still decode properly! Symbol erasing is

NOT deleting symbols, but rather erasing them. Symbol erasing is erasing some received symbols NOT the entire received word. Symbol erasing can be, but is usually not, working within the gray zone. It is possible to increase the SNR a little by NOT necessarily resetting the erasure symbol R_i" to zero! THIS CHAPTER ASSUMES THAT THE DEMODULATOR PASSES WHATEVER ITS BEST ESTIMATE OF THE RECEIVED SYMBOL VALUES ARE TO THE RS DECODER.

The reason why the symbol erasing issue may be important is that the power of the error correcting code increases. The upper limit of T=2t error symbols can be reached if all 2t errors can be flagged as erasures. However, there are a few drawbacks. First of all, correcting erasures requires many times the computational capability. Secondly, in some systems the accuracy of the demodulator to correctly determine the erasure status may be too low for substantial increased system error correction; in fact, if it is poorly designed, it may even decrease the system's error correction capability.

A coding system which corrects erasures can correct up to t_E errors and t_E" erasures if $2t_E+t_E" \leq d-1$ where the distance $d \leq d_{min}$. For RS codes, we can correct all t_E errors and all t_E" erasures if $2t_E+t_E" \leq d_{min}-1$ or rather $2t_E+t_E" \leq 2t$.

For a (15,9) RS code:	t_E (errors)	t_E" (erasures)
	0	6,5,4,3,2,1, or 0
	1	4,3,2,1, or 0
	2	2,1, or 0
	3	0

To explain how the RS code can correct T>t error symbols using erasure, assume the demodulator flagged the X^7 and X^2 locations as being unreliable. Therefore, by the demodulator's particular algorithm and symbol quality data, it has decided that these were erasure symbols; the X^7 and X^2 locations are considered to be erasures. In our previous (15,9) RS example which has T=2 error symbols at the X^8 and the X^2 locations, the RS code, besides correcting the two errors, could have corrected an additional $t_E" \leq d-2t_E = 2$ erasures. Assume a similar case as in the previous (15,9) RS example, but with two erasures at the X^7 and the X^2 locations with two errors at the X^8 and the $X^1=X$ locations. Remember that ERASURES ARE ERRORS whose erasure locations are probably known for sure and whose erasure values are probably not known.

Let the same message and thus the same code word be sent as in chapters 1,2,3, and 4. The following is our (15,9) RS erasure example:

$M(X) = \alpha^{11}$

$C(X) = \alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$

Assume more than t errors were injected into the code word, i.e., assume T>t. A system without symbol erasing capability would not be able to correct the received word because T>t (except for a few rare complete decoding cases). Usually, there is only a high probability of error detection. Now assume the input to the demodulator to be some type of modulated "channel symbols" which represent the following:

$R(X)' = X^8+\underline{"?"X^7} + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\underline{"?"X^2}+\alpha^9X+\alpha^{12}$

very weak and noisy "channel symbols"

The output of the demodulator is equivalent to the input with some or all of the poor quality locations flagged as erasures. Also, the demodulator should pass its best guess of the value at each location because sometimes it may guess correctly. In other words, the following is given:

$R(X)'' = X^8+\underline{\alpha^7X^7} + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\underline{\alpha^3X^2}+\alpha^9X+\alpha^{12}$

ERASURES (unreliable code symbols)

After R(X)" is processed by the RS decoder with erasure capability, the following code word should result:

$C(X)'' = \alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12} = C(X)$

In summary, since C(X)" can equal C(X), it is possible to correct T>t error symbols if some of the poor quality locations can be determined to be erasures. In fact, if all the actual error locations are known, thus all the errors are erasures, the code can correct up to a maximum of T=2t error symbols. This particular example with four errors (two errors plus two erasures) will be completely decoded in sections 5.2 and 5.3; T=4>t=3 in our (15,9) RS example.

The following figure and table illustrate the architectural differences between a RS coding system with and without erasure; also see figures 3.1-1 and 4.1-1 and tables 3.1-1 and 4.1-1.

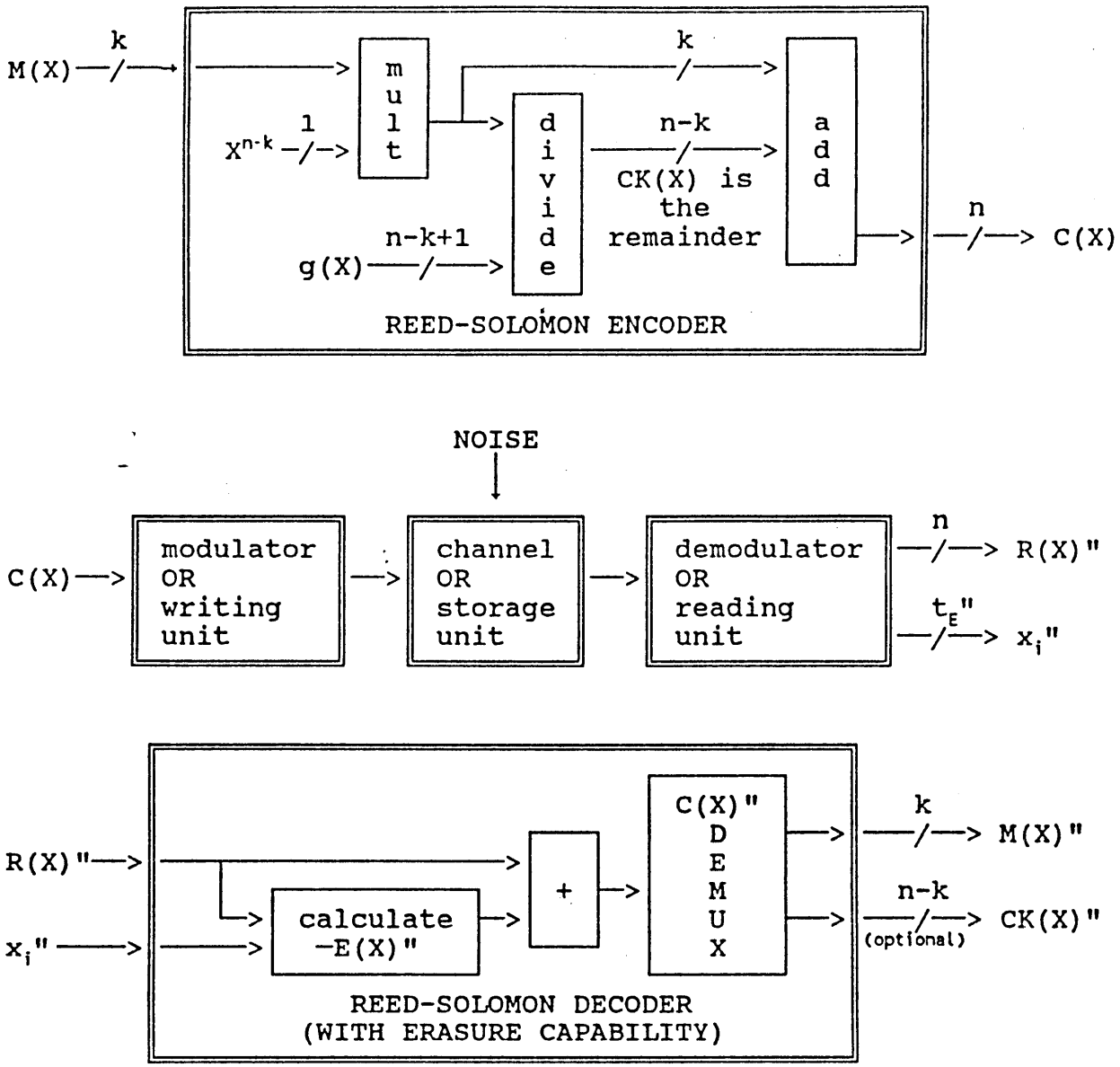


Figure 5.1-1. - RS coding block diagram with erasure capability.

TABLE 5.1-1. - RS SYMBOL ERASURE POLYNOMIAL DEFINITIONS

received word R(X)" in an erasure system consisting of received symbols R_i":

$$R(X)" = C(X) + E(X)$$
$$= [(C_{n-1}+E_{n-1})X^{n-1} + (C_{n-2}+E_{n-2})X^{n-2} + \ldots + (C_1+E_1)X + (C_0+E_0)]$$
$$= R_{n-1}"X^{n-1} + R_{n-2}"X^{n-2} + \ldots + R_1"X + R_0"$$

erasure positions x_i":

$x_i" = X^j$ for $i = 1,2,\ldots,t_E"$ AND for j defined from the demodulator where t_E = number of errors, $t_E"$ = number of erasures, and $2t_E+t_E" \leq 2t$.

decoded error pattern E(X)" in an erasure system consisting of error locations x_i (which can include erasure locations $x_i"$) and error values y_i (which can include erasure values $y_i"$):

$$E(X)" = y_1x_1 + y_2x_2 + \ldots + y_Tx_T$$

decoded code word C(X)" in an erasure system consisting of code word symbols C_i":

$$C(X)" = R(X)" - E(X)"$$
$$= R(X)" + E(X)"$$
$$= X^{n-k}M(X)" + CK(X)"$$
$$= X^{n-k}M(X)" + X^{n-k}M(X)" \bmod g(X)$$
$$= C_{n-1}"X^{n-1} + C_{n-2}"X^{n-2} + \ldots + C_1"X + C_0"$$

decoded message (or data or information) M(X)" in an erasure system consisting of message symbols M_i":

$$M(X)" = C_{n-1}"X^{k-1} + C_{n-2}"X^{k-2} + \ldots + C_{n-k+1}"X + C_{n-k}"$$
$$= M_{k-1}"X^{k-1} + M_{k-2}"X^{k-2} + \ldots + M_1"X + M_0"$$

decoded parity-check CK(X)" consisting of check symbols CK_i":

$$CK(X)" = C_{n-k-1}"X^{n-k-1} + C_{n-k-2}"X^{n-k-2} + \ldots + C_1"X + C_0"$$
$$= CK_{n-k-1}"X^{n-k-1} + CK_{n-k-2}"X^{n-k-2} + \ldots + CK_1"X + CK_0"$$

5.2 RS ENCODING USING SYMBOL ERASURE

The encoding procedure for RS codes using erasure is identical to RS codes without using erasure (see chapter 3). This is due to the fact that symbol erasure is soft decision, but only at the demodulator and decoder and not the modulator and encoder. Because of this and the discussion in section 5.1, the message and code word are $M(X) = \alpha^{11}X$ and $C(X) = \alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$.

5.3 RS DECODING USING SYMBOL ERASURE

In order to demonstrate the increase in error correction by using symbol erasure, the decoding operations are demonstrated for the (15,9) RS code example as discussed earlier in sections 5.1 and 5.2. In the following sections within 5.3, all the decoding operations for the received word $R(X)'' = X^8+\alpha^7X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^3X^2+\alpha^9X+\alpha^{12}$ will be shown. For learning purposes, R(X)" was made very similar to the R(X) for the nonerasure (15,9) RS example found in chapters 3 and 4 and appendix C; $R(X)_{earlier\_example} = X^8+\alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^3X^2+\alpha^8X+\alpha^{12}$.

THE DECODING PROCESS USING SYMBOL ERASURE IS A SIX-STAGE PROCESS:

1. Calculate the 2t syndrome components from the received word.

2. Calculate the $2t-t_E''$ modified syndrome components from the syndrome components and the erasure locations.

3. Calculate the t_E error-locator coefficients from the modified syndrome components.

4. Calculate the t_E error locations from the error-locator numbers which are from the error-locator word.

5. Calculate the $T=t_E+t_E''$ error values and erasure values from the error-locator numbers and the erasure-locator numbers.

6. Calculate the decoded code word from the received word, the error and erasure locations, and the error and erasure values.

5.3.1 Erasure Determination

Assume the received word has the errors and the erasures as presented in sections 5.1, 5.2, and 5.3. The summarized results are placed in table 5.3.1-1.

TABLE 5.3.1-1. - CODE WORD C(X) USED IN (15,9) RS ERASURE EXAMPLE

Location (X_i)	Code word symbol C_i	Received word symbol R_i	Received word symbol R_i"	Erasure capability (without —> with)
X^8	0	1	1	error —> error
X^7	α^{11}	?	?=α^7	error —> erasure
X^2	α^{14}	?	?=α^3	error —> erasure
X	α^8	α^9	α^9	error —> error

So therefore assume our new received word has two errors and two erasures.

$$R(X)'' = \underset{\substack{\text{ERROR}}}{X^8}+\underset{\substack{\text{ERASURE}}}{\alpha^7X^7} + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\underset{\substack{\text{ERASURE}}}{\alpha^3X^2}+\underset{\substack{\text{ERROR}}}{\alpha^9X}+\alpha^{12}$$

Because we are now handling two "errors" and two "erasures," the code is actually correcting $T = t_E+t_E''= 4$ error symbols greater than $t=3$ error symbols, but less than, or equal to, $2t=6$ error symbols! Do not confuse $T = t_E+t_E''$ and $2t_E+t_E'' \leq 2t$ with $T\leq 2t$; T can equal 2t when $t_E=0$.

5.3.2 Syndrome Components S_i

The first stage is to calculate the syndrome components S_i. For our example, FCR=1 and $\alpha^G=\alpha^1$. $R(X)'' = X^8+\alpha^7X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^3X^2+\alpha^9X+\alpha^{12}$.

$S_1 = R(\alpha)'' = \alpha^{13}$ $\quad S_3 = R(\alpha^3)'' = \alpha^{11}$ $\quad S_5 = R(\alpha^5)'' = \alpha^{14}$

$S_2 = R(\alpha^2)'' = \alpha^4$ $\quad S_4 = R(\alpha^4)'' = \alpha^{12}$ $\quad S_6 = R(\alpha^6)'' = \alpha^{14}$

Since not all the S_i are zero, then we know that $T=t_E+t_E''>0$.

5.3.3 Modified Syndrome Components S_i''

The second stage is to calculate the modified syndrome components S_i'' from the syndrome components S_i and the erasure-locator coefficients σ_j'':

$$S_i" = \sum_{j=0}^{t_E"} \sigma_j" S_{i-j} \qquad \text{for } i = t_E"+FCR, t_E"+FCR+1, \ldots, 2t+FCR-1$$

From the demodulator, $x_1"=X^2$ (or $z_1"=\alpha^2$ because $\alpha^G=\alpha^1$) and $x_2"=X^7$ (or $z_2"=\alpha^7$ because $\alpha^G=\alpha^1$), therefore, $t_E"=2$ erasures are given. Since we are going to use $t_E"=2$ erasures for this example, hopefully $t_E \leq 2$ errors; $t_E \leq (2t-t_E")/2$. If we later determine that $t_E>2$ errors, then we can decrease our $t_E"$; e.g., if $t_E=3$, then $t_E" \leq 2(t-t_E)$ which for this example would be $t_E"=0$.

It does not matter how the locations are assigned to the $x_i"$. However, I prefer lower locations to correspond to lower indices and higher locations to higher ones (i.e., $x_1"=X^2$ and $x_2"=X^7$ not $x_1"=X^7$ and $x_2"=X^2$). The reciprocal of the erasure-locator polynomial $\sigma_r(X)"$ is defined as having the erasure-locator numbers $z_i"$ as its roots:

$$\sigma_r(X)" = \prod_{i=1}^{t_E"} (X + z_i")$$

$$\begin{aligned} \sigma_r(X)" &= (X + z_1")(X + z_2") \\ &= (X + \alpha^2)(X + \alpha^7) \\ &= X^2 + \alpha^{12}X + \alpha^9 \end{aligned}$$

Therefore, $\sigma_r(X)" = X^2+\alpha^{12}X+\alpha^9$ and so $e_1"=\alpha^{12}$ and $e_2"=\alpha^9$ because $\sigma(X)" = \sigma_2"X^2+\sigma_1"X+1$. Since $\sigma_0"=\sigma_0$ is defined to be 1 and therefore, the $\sigma_i"$ is defined for $i=0,1,2,\ldots,t_E"$, we finally determine the modified syndrome components $S_i"$.

$$S_i" = \sum_{j=0}^{2} \sigma_j" S_{i-j} \qquad \text{for } i = 3,4,5,6$$

Substituting values:

$$\begin{aligned} S_3" &= S_3 + \sigma_1"S_2 + \sigma_2"S_1 = \alpha^{11} + \alpha^{12}\alpha^4 + \alpha^9\alpha^{13} = \alpha^{11} + \alpha + \alpha^7 = \alpha^{10} \\ S_4" &= S_4 + \sigma_1"S_3 + \sigma_2"S_2 = \alpha^{10} \\ S_5" &= S_5 + \sigma_1"S_4 + \sigma_2"S_3 = \alpha^8 \\ S_6" &= S_6 + \sigma_1"S_5 + \sigma_2"S_4 = \alpha^7 \end{aligned}$$

Therefore, $S_3" = \alpha^{10}$, $S_4" = \alpha^{10}$, $S_5" = \alpha^8$, and $S_6" = \alpha^7$.

5.3.4 Error-Locator Coefficients σ_i

The third stage is to calculate the error-locator polynomial $\sigma(X)$ from the modified syndrome components. For instructional purposes

we set up our example for t_E=2.

$$S_i'' = \sum_{j=0}^{t_E-1} S_{i+j-t_E}''\sigma_{t_E-j} \qquad \text{for } i = T+FCR, T+FCR+1, \ldots, 2t+FCR-1$$

$$S_i'' = \sum_{j=0}^{1} S_{i+j-2}''\sigma_{2-j} \qquad \text{for } i = 5,6$$

$$S_3''\sigma_2 + S_4''\sigma_1 = S_5''$$
$$S_4''\sigma_2 + S_5''\sigma_1 = S_6''$$

Substitution of values is shown below:

$$\alpha^{10}\sigma_2 + \alpha^{10}\sigma_1 = \alpha^8$$
$$\alpha^{10}\sigma_2 + \bar{\alpha}^{8}\sigma_1 = \alpha^7$$

Therefore, $\sigma_1=\alpha^{10}$ and $\sigma_2=\alpha^9$ and so $\sigma(X) = \alpha^9X^2+\alpha^{10}X+1$.

5.3.5 Error Locations x_i

The fourth stage is to calculate the error locations x_i using the roots of the reciprocal of the error-locator polynomial $\sigma_r(X)$. Determine the error-locator numbers z_i from $\sigma_r(X) = X^2+\alpha^{10}X+\alpha^9$.

$$\sigma_r(1) = \alpha^6$$
$$\sigma_r(\alpha) = 0$$
$$\sigma_r(\alpha^2) = \alpha^5$$
$$\sigma_r(\alpha^3) = \alpha^7$$
$$\sigma_r(\alpha^4) = \alpha^5$$
$$\sigma_r(\alpha^5) = \alpha^6$$
$$\sigma_r(\alpha^6) = \alpha^{10}$$
$$\sigma_r(\alpha^7) = \alpha^{10}$$
$$\sigma_r(\alpha^8) = 0$$

Therefore, $z_1=\alpha$ and $z_2=\alpha^8$. Since $\alpha^6=\alpha^1$ in our example, the corresponding error-locations x_1 and x_2 are simply X and X^8.

5.3.6 Error Values y_i

The fifth stage is to calculate the error values y_i. At this point there is no need to make a distinction between errors and erasures. Therefore, rearrange the indices such that the lower ones correspond to the lower locations. BEFORE: $T=t_E+t_E''=4$ errors and

erasures, $z_1"=\alpha^2$, $z_2"=\alpha^7$, $z_1=\alpha$, and $z_2=\alpha^8$ (and $x_1"=X^2$, $x_2"=X^7$, $x_1=X$, and $x_2=X^8$), and $y_1"$, $y_2"$, y_1, and y_2. AFTER: T=4 errors, $z_1=\alpha$, $z_2=\alpha^2$, $z_3=\alpha^7$, and $z_4=\alpha^8$ (and $x_1=X$, $x_2=X^2$, $x_3=X^7$, and $x_4=X^8$), and y_1, y_2, y_3, and y_4. Now solve for the error (including erasure) values.

$$S_i = \sum_{j=1}^{I} y_j z_j^i \qquad \text{for } i = FCR, FCR+1, \ldots, T+FCR-1 = 1, 2, \ldots, T$$

The following are the set of the independent NON-LINEAR simultaneous equations:

$$\begin{aligned}
y_1 z_1 + y_2 z_2 + y_3 z_3 + y_4 z_4 &= S_1 \\
y_1 z_1^2 + y_2 z_2^2 + y_3 z_3^2 + y_4 z_4^2 &= S_2 \\
y_1 z_1^3 + y_2 z_2^3 + y_3 z_3^3 + y_4 z_4^3 &= S_3 \\
y_1 z_1^4 + -y_2 z_2^4 + y_3 z_3^4 + y_4 z_4^4 &= S_4
\end{aligned}$$

Substituting values, the set becomes a set of independent LINEAR simultaneous equations:

$$\begin{aligned}
\alpha y_1 + \alpha^2 y_2 + \alpha^7 y_3 + \alpha^8 y_4 &= \alpha^{13} \\
\alpha^2 y_1 + \alpha^4 y_2 + \alpha^{14} y_3 + \alpha y_4 &= \alpha^4 \\
\alpha^3 y_1 + \alpha^6 y_2 + \alpha^6 y_3 + \alpha^9 y_4 &= \alpha^{11} \\
\alpha^4 y_1 + \alpha^8 y_2 + \alpha^{13} y_3 + \alpha^2 y_4 &= \alpha^{12}
\end{aligned}$$

Therefore, $y_1=\alpha^{12}$, $y_2=1$, $y_3=\alpha^8$, and $y_4=1$.

5.3.7 Decoded Code Word C(X)"

The sixth and final stage is to construct the decoded code word C(X)". The error locations x_2 and x_3 were determined in section 5.1 and again in section 5.3.1; $x_2=X^2$ and $x_3=X^7$. The error locations x_1 and x_4 were determined in section 5.3.5; $x_1=X$ and $x_4=X^8$. All the error values y_1, y_2, y_3, and y_4 were determined in section 5.3.6. Using erasure, the decoded error E(X)" is simply the x_i and the y_i as follows:

$$E(X)" = \sum_{i=1}^{I} y_i x_i$$

$$\begin{aligned}
E(X)" &= y_1 x_1 + y_2 x_2 + y_3 x_3 + y_4 x_4 \\
&= \alpha^{12} X + 1X^2 + \alpha^8 X^7 + 1X^8 \\
&= X^8 + \alpha^8 X^7 + X^2 + \alpha^{12} X
\end{aligned}$$

The decoded code word using erasure is:

$$\begin{aligned} C(X)'' &= R(X)' + E'(X)' \\ &= [X^8+\alpha^7X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^3X^2+\alpha^9X+\alpha^{12}] \\ &\quad + [X^8+\alpha^8X^7 + X^2+\alpha^{12}X] \\ &= \underline{(1+1)X^8}+\underline{(\alpha^7+\alpha^8)X^7} + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\underline{(\alpha^3+1)X^2}+\underline{(\alpha^9+\alpha^{12})X}+\alpha^{12} \end{aligned}$$

Therefore, $C(X)'' = \alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$. From section 3.3.2 or sections 5.1 or 5.2, $C(X) = \alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$. Therefore, $C(X)''=C(X)$ as expected!

Strip the message field from the corrected data. Therefore, $M(X)'' = C_{n-1}''X^{k-1}+C_{n-2}''X^{k-2}+\ldots+C_{n-k+1}''X+C_{n-k}'' = \alpha^{11}X$. From sections 3.3.2 or 5.1 or 5.2, $M(X) = \alpha^{11}X$. Therefore, $M(X)''=M(X)$ as expected.

Strip the parity-check field from the corrected data. Therefore, $CK(X)'' = C_{n-k-1}''X^{n-k-1}+C_{n-k-2}''X^{n-k-2}+\ldots+C_1''X+C_0'' = \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$. From sections 3.3.2 or 5.1, $CK(X) = \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$. Therefore, $CK(X)''=CK(X)$ as expected.

Therefore, we correctly decoded our (15,9) RS example with T>t error symbols using erasure capability! In fact, we pushed T to the maximum error correction capability of an erasure system for a particular value of t_E. If t_E were a smaller value, then we could have somehow picked more erasures (even if the demodulator was so called happy enough); this should also increase the SNR slightly. If t_E were a larger value, then we would have to pick fewer erasures.

5.3.8 Determining t_E in a Symbol Erasure System

For instructional and clarity purposes, the preceding sections ASSUMED $t_E=2$. In this section, we will again learn how the decoder can determine the value of t_E using MLD.

In our example worked in the preceding sections, the decoder was given $t_E=2$. In real life the decoder must determine t_E using MLD. It first assumes that $t_E=0$. If $t_E=0$ does not work, then it tries $t_E=1$. It keeps increasing t_E ($t_E \leq t$) until it can solve the equations to determine the value of t_E.

Let us now complete the work on the (15,9) RS erasure example. One of the ways to determine t_E is to first determine T. We could calculate the 2t syndrome components and then determine an error-locator polynomial using Berlekamp's iterative algorithm. For our example with $T \leq t$, we obtain $\sigma(X) = \alpha^{10}X^3+\alpha^7X^2+\alpha^6X+1$. This is a third degree polynomial and only has one root, i.e., $\sigma(\alpha^6)=0$ or

α^6 is the only root. Therefore, since there should be three roots and only one GF(16) root exists for this t=3 degree polynomial, T>t. Since T>t then $t_E+t_E"$>t and therefore t_E>t-$t_E"$ or t_E>1 (assuming all the erasures are non-error-free symbols). Then we can proceed and try t_E=2 and work the problem as shown in sections 5.3.4, 5.3.5, 5.3.6, and 5.3.7.

Another way to determine t_E is to first assume t_E=0. If t_E=0, then T=$t_E+t_E"$=2 locations, and thus $\sigma(X)$=0; we do not need to complete the second, third, and fourth stages. We need to determine the erasure values $y_1"$ and $y_2"$ which at this point we simply denote as the error values y_1 and y_2. We solve for the error values as shown in section 5.3.6 and this results in the following set of equations:

$$\begin{aligned} y_1z_1 + y_2z_2 &= S_1 \\ y_1z_1^2 + y_2z_2^2 &= S_2 \\ y_1z_1^3 + y_2z_2^3 &= S_3 \\ y_1z_1^4 + y_2z_2^4 &= S_4 \\ y_1z_1^5 + y_2z_2^5 &= S_5 \\ y_1z_1^6 + y_2z_2^6 &= S_6 \end{aligned}$$

We choose any T=2 equations to solve for the T=2 unknown y_i.

$$\begin{aligned} y_1z_1 + y_2z_2 &= S_1 \\ y_1z_1^2 + y_2z_2^2 &= S_2 \end{aligned}$$

Therefore, $y_1=\alpha^9$ and $y_2=\alpha^{12}$ with $z_1=z_1"=\alpha^2$ ($x_1=x_1"=X^2$), $z_2=z_2"=\alpha^7$ ($x_2=x_2"=X^7$), $S_1=\alpha^{13}$, and $S_2=\alpha^4$. We then form our decoded error pattern $E(X)" = y_1x_1 + y_2x_2 = \alpha^9X^2 + \alpha^{12}X^7$. The decoded code word $C(X)" = R(X)" + E(X)" = X^8+\alpha^2X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha X^2+\alpha^9X+\alpha^{12}$, but this is not a valid code word! It is not a valid code word because the syndrome of C(X)" is NOT zero, e.g., the third syndrome component of $C(X)" = C(\alpha^3)" \neq 0$. Therefore, t_E>0 errors occurred.

Next assume t_E=1. $S_i" = S_{i-1}"\sigma_1$ for i = 4,5,6. $\sigma_1=S_4"/S_3"$ which should be the same as $\sigma_1=S_5"/S_4"$ and $\sigma_1=S_6'/S_5"$, but is not! Therefore, t_E>1.

Now we try t_E=2 and work the problem as shown in sections 5.3.4, 5.3.5, 5.3.6, and 5.3.7.

We should note that if t_E>2 in our erasure example, then we would continue the MLD process for increasing values of t_E ($t_E \leq t$). In our example, if we increase t_E past t_E=2, then we would decrease $t_E"$ by two each time that we would increase t_E by one; $2t_E+t_E" \leq 2t$.

5.4 SUMMARY

In chapter 5 the Reed-Solomon code with symbol erasure capability was demonstrated to be working for $T = t_E + t_E'' > t$ error symbols. This is because we are able to take our original message M(X), append check information CK(X) systematically to it, send it, corrupt it with noise, receive it, send it through our demodulator with symbol erasure detection capability, and then decode the received vector R(X) back into our original M(X). Of course, symbol erasure systems can also correct $T \leq t$ error symbols!

APPENDIX A
RS HARDWARE ENCODING DESIGN

This appendix will step us carefully through a common SRC implementation to perform the RS encoding process. RS encoders are usually implemented using a SRC.

A.1 (15,9) RS SHIFT REGISTER ENCODER CIRCUIT

To develop a RS encoder, a SRC is usually used. Figure A.1-1 shows the SRC for our (15,9) RS example where $g(X) = X^6+\alpha^{10}X^5+\alpha^{14}X^4+\alpha^4X^3+\alpha^6X^2+\alpha^9X+\alpha^6$. Other RS codes can be implemented in the same manner. As in most, if not all, of encoder/decoder systems (codec systems), the encoder is the easy part to understand and to apply into hardware.

There are a few notes on the operation of the encoder circuit within figure A.1-1: When the switches are open, their outputs are all pulled to zero. The exception is when the switch outputs are the inputs to the output shift register; then, they are open circuited.

Table A.1-1 presents the procedure necessary to operate the (15,9) RS encoder which is shown in figure A.1-1. The figure after figure A.1-1, i.e., figure A.1-2, shows the blow up of the output shift register.

TABLE A.1-1. - (15,9) RS ENCODER PROCEDURE

Step 1.	Clear feedback shift register [$X^0=X^1=...=X^{2t-1}=0$].
Step 2.	Enable switch A and disable switch B.
Step 3.	Clock the information symbols into the circuit, most significant symbol first (clock k times).
Step 4.	Disable switch A and enable switch B.
Step 5.	Clock the check symbols out of the circuit (clock n-k additional times).
Step 6.	The resultant code word is in the output shift register.
Step 7.	GOTO either Step 1 or Step 2.

$g(X) = X^6+\alpha^{10}X^5+\alpha^{14}X^4+\alpha^4X^3+\alpha^6X^2+\alpha^9X+\alpha^6$

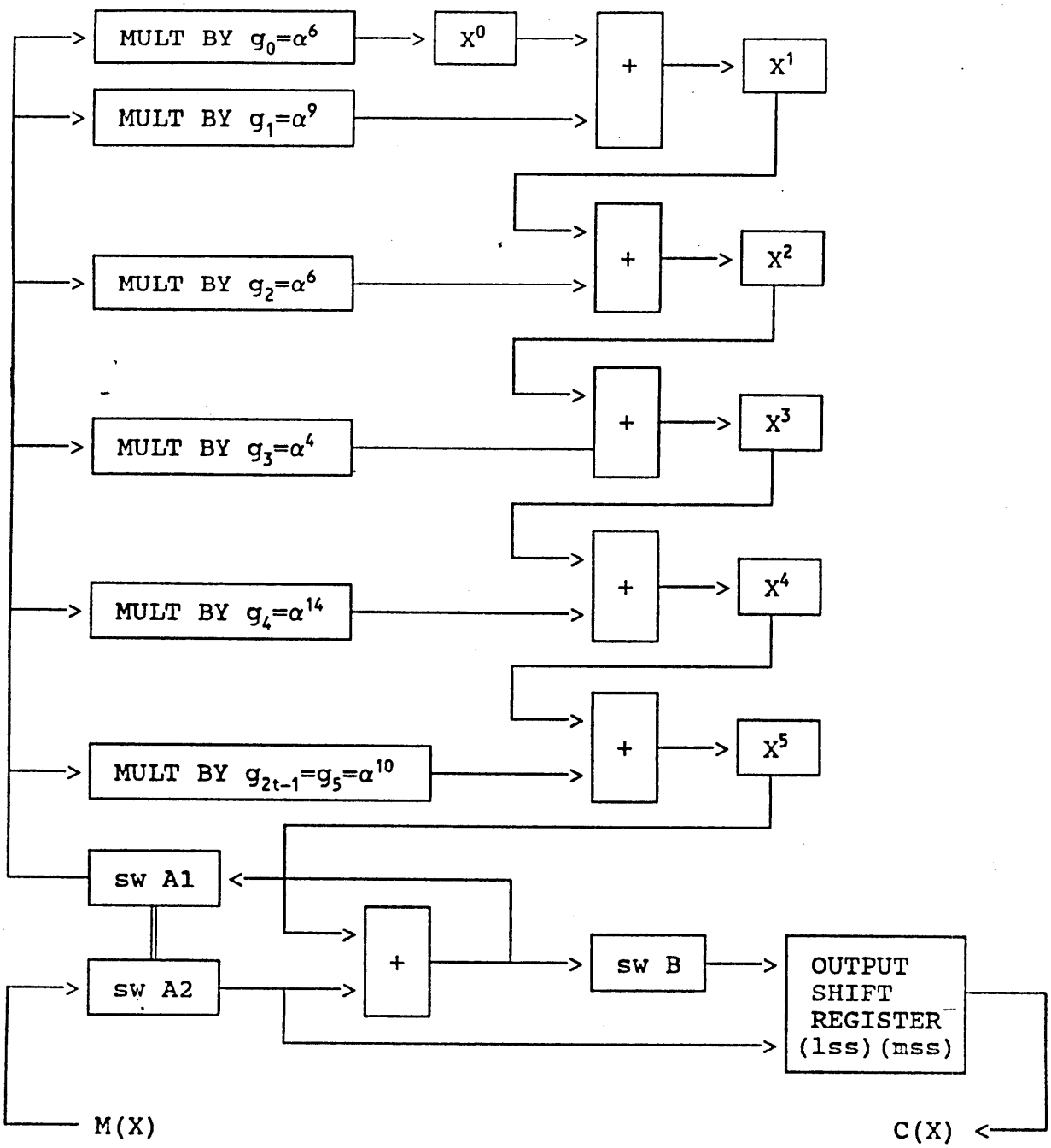


Figure A.1-1. - (15,9) RS encoder shift register circuit.

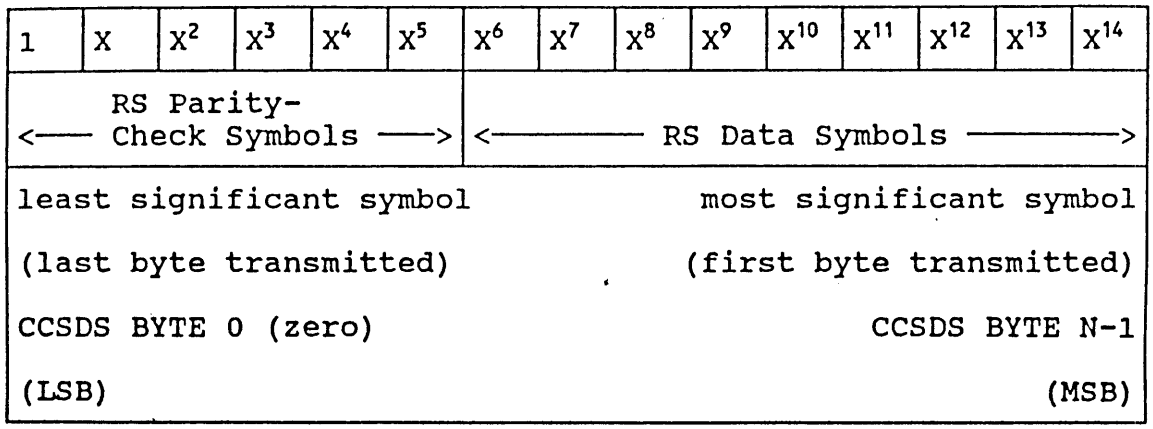


Figure A.1-2. - Blow up of the output shift register.

In an attempt to explain the different states of the hardware in detail, tables A.1-2 and A.1-3 are given with message $M(X) = \alpha^{11}X$.

TABLE A.1-2. - FEEDBACK SHIFT REGISTER STATES

Case	$M_i = \alpha^j$	X^i	Feedback Shift Register X^0	X^1	X^2	X^3	X^4	X^5
0	-- = --	--	0	0	0	0	0	0
1	$M_8 = 0$	X^8	0	0	0	0	0	0
2	$M_7 = 0$	X^7	0	0	0	0	0	0
3	$M_6 = 0$	X^6	0	0	0	0	0	0
4	$M_5 = 0$	X^5	0	0	0	0	0	0
5	$M_4 = 0$	X^4	0	0	0	0	0	0
6	$M_3 = 0$	X^3	0	0	0	0	0	0
7	$M_2 = 0$	X^2	0	0	0	0	0	0
8	$M_1 = \alpha^{11}$	X	α^2	α^5	α^2	1	α^{10}	α^6
9	$M_0 = 0$	1	α^{12}	α^8	α^{14}	α^4	α^{10}	α^8
10	-- = --	--	0	α^{12}	α^8	α^{14}	α^4	α^{10}
11	-- = --	--	0	0	α^{12}	α^8	α^{14}	α^4
12	-- = --	--	0	0	0	α^{12}	α^8	α^{14}
13	-- = --	--	0	0	0	0	α^{12}	α^8
14	-- = --	--	0	0	0	0	0	α^{12}
15	-- = --	--	0	0	0	0	0	0

TABLE A.1-3. - OUTPUT SHIFT REGISTER STATES

	<——— Output Shift Register ———>														
Case	1	X	X^2	X^3	X^4	X^5	X^6	X^7	X^8	X^9	X^{10}	X^{11}	X^{12}	X^{13}	X^{14}
0	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
1	0	--	--	--	--	--	--	--	--	--	--	--	--	--	--
2	0	0	--	--	--	--	--	--	--	--	--	--	--	--	--
3	0	0	0	--	--	--	--	--	--	--	--	--	--	--	--
4	0	0	0	0	--	--	--	--	--	--	--	--	--	--	--
5	0	0	0	0	0	--	--	--	--	--	--	--	--	--	--
6	0	0	0	0	0	0	--	--	--	--	--	--	--	--	--
7	0	0	0	0	0	0	0	--	--	--	--	--	--	--	--
8	α^{11}	0	0	0	0	0	0	0	--	--	--	--	--	--	--
9	0	α^{11}	0	0	0	0	0	0	0	--	--	--	--	--	--
10	α^8	0	α^{11}	0	0	0	0	0	0	0	--	--	--	--	--
11	α^{10}	α^8	0	α^{11}	0	0	0	0	0	0	0	--	--	--	--
12	α^4	α^{10}	α^8	0	α^{11}	0	0	0	0	0	0	0	--	--	--
13	α^{14}	α^4	α^{10}	α^8	0	α^{11}	0	0	0	0	0	0	0	--	--
14	α^8	α^{14}	α^4	α^{10}	α^8	0	α^{11}	0	0	0	0	0	0	0	--
15	α^{12}	α^8	α^{14}	α^4	α^{10}	α^8	0	α^{11}	0	0	0	0	0	0	0

It should be noted that the code word C(X) is the value in the output shift register during case n; in this example, n=15 so $C(X) = \alpha^{11}X^7+\alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$. This checks with the earlier calculation for C(X) within chapter 3.

To explain how tables A.1-2 and A.1-3 are developed the following equations and sample calculations are given. These are simply derived from figure A.1-1:

Case 0: $X^0 = X^1 = X^2 = X^3 = X^4 = X^5 = 0$

Case 1 through case k=9:

$X^0 = (X^5_{old} + M(X))\alpha^6$
$X^1 = X^0_{old} + (X^5_{old} + M(X))\alpha^9$
$X^2 = X^1_{old} + (X^5_{old} + M(X))\alpha^6$
$X^3 = X^2_{old} + (X^5_{old} + M(X))\alpha^4$
$X^4 = X^3_{old} + (X^5_{old} + M(X))\alpha^{14}$
$X^5 = X^4_{old} + (X^5_{old} + M(X))\alpha^{10}$

Case k+1=10 through case n=15:

$X^0 = 0$
$X^1 = X^0_{old}$
$X^2 = X^1_{old}$
$X^3 = X^2_{old}$
$X^4 = X^3_{old}$
$X^5 = X^4_{old}$

Working through some of the calculations:

Case 0:
$X^0 = X^1 = X^2 = X^3 = X^4 = X^5 = 0$

Case 8:

$$
\begin{aligned}
X^0 &= (X^5_{old}+M(X))\alpha^6 = (0+\alpha^{11})\alpha^6 = (\alpha^{11})\alpha^6 = \alpha^2 \\
X^1 &= X^0_{old} + (X^5_{old}+M(X))\alpha^9 = (\alpha^{11})\alpha^9 = \alpha^5 \\
X^2 &= X^1_{old} + (X^5_{old}+M(X))\alpha^6 = (\alpha^{11})\alpha^6 = \alpha^2 \\
X^3 &= X^2_{old} + (X^5_{old}+M(X))\alpha^4 = (\alpha^{11})\alpha^4 = 1 \\
X^4 &= X^3_{old} + (X^5_{old}+M(X))\alpha^{14} = (\alpha^{11})\alpha^{14} = \alpha^{10} \\
X^5 &= X^4_{old} + (X^5_{old}+M(X))\alpha^{10} = (\alpha^{11})\alpha^{10} = \alpha^6
\end{aligned}
$$

Case 9:

$$
\begin{aligned}
X^0 &= (X^5_{old}+M(X))\alpha^6 = (\alpha^6 + 0)\alpha^6 = (\alpha^6)\alpha^6 = \alpha^{12} \\
X^1 &= X^0_{old} + (X^5_{old}+M(X))\alpha^9 = \alpha^2 + (\alpha^6)\alpha^9 = \alpha^8 \\
X^2 &= X^1_{old} + (X^5_{old}+M(X))\alpha^6 = \alpha^5 + (\alpha^6)\alpha^6 = \alpha^{14} \\
X^3 &= X^2_{old} + (X^5_{old}+M(X))\alpha^4 = \alpha^2 + (\alpha^6)\alpha^4 = \alpha^4 \\
X^4 &= X^3_{old} + (X^5_{old}+M(X))\alpha^{14} = 1 + (\alpha^6)\alpha^{14} = \alpha^{10} \\
X^5 &= X^4_{old} + (X^5_{old}+M(X))\alpha^{10} = \alpha^{10} + (\alpha^6)\alpha^{10} = \alpha^8
\end{aligned}
$$

Therefore, $CK(X) = \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$.

The parity-check symbols are available in the feedback shift register after the message symbols M(X) have been clocked into the circuit k times. Cases k+1 through n simply shifts these check symbols out of the feedback register and into the output shift register which already contains M(X) at that time.

A.2 OTHER RS SHIFT REGISTER ENCODER CIRCUITS

$CK(X) = X^{n-k}M(X) \bmod g(X)$. The hardware can take care of the time delay by just piecing the data fields together at the proper time. So our input can be either $X^{n-k}M(X)$ or just M(X) to determine the following hardware decoder. The following SRC is of the same Galois configuration as figure A.1-1.

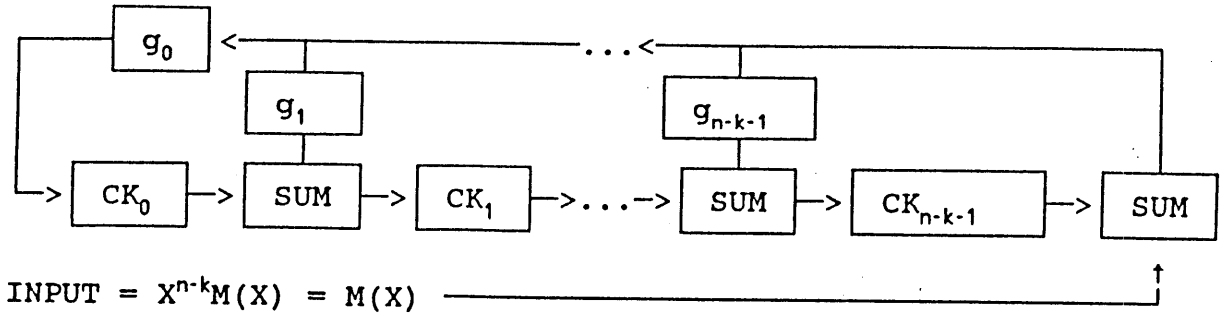


Figure A.2-1. - Encoder SRC using CK(X) = M(X) mod g(X).

CK(X) = $X^{n-k}h(X)M(X)$ mod (X^n+1) because h(X) = (X^n+1) / g(X). The following circuit is using this parity-check polynomial h(X) instead of the generator polynomial g(X); h(X) is related to g(X). The circuit is initially loaded with the M_i. Notice that this Fibonacci configuration uses the reciprocal of h(X) to dictate the sequence of the coefficient connections.

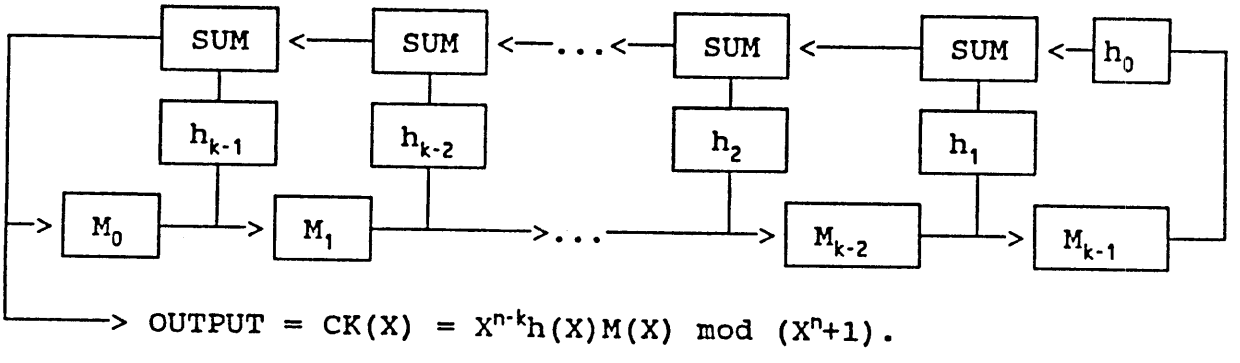


Figure A.2-2. - Encoder SRC using CK(X) = $X^{n-k}h(X)M(X)$ mod (X^n+1).

Besides the Galois and Fibonacci configurations, there are others which require substantially less hardware. One of these configurations which is more closely matched to the hardware is the Bit-Serial Reed-Solomon Encoder by Elwyn R. Berlekamp. The Galois and Fibonacci configurations are more straightforward from a mathematical and more easily understood viewpoint. However, the Bit-Serial encoders are more efficient from an implementation viewpoint. If there is further interest in Bit-Serial encoders, please refer to the references section.

APPENDIX B
RS HARDWARE DECODING DESIGN

If one only views the coding system as a mapping and demapping procedure and tries to design a look-up-table hardware circuit, one is in for a big surprise. This type of design is impractical because it requires a ridiculous amount of memory; Table B-1 testifies to this fact. We need a better hardware design than this family of designs!

TABLE B-1. - MEMORY TABLE OF A LOOK-UP-TABLE HARDWARE CIRCUIT

Code:	RS encoder	RS decoder
(n,k)	2^{mk} X m(n-k) bits	2^{mn} X mk bits
(3,1)	4 X 4 bits	64 X 2 bits
(7,3)	512 X 12 bits	$>10^{6}$ X 9 bits
(7,5)	$>10^{4}$ X 6 bits	$>10^{6}$ X 15 bits
(15,9)	$>10^{10}$ X 24 bits	$>10^{18}$ X 36 bits
(255,223)	$>10^{69}$ X 256 bits	$>10^{79}$ X 1,784 bits

The decoding hardware could simply be a combinational logic circuit as in figure B-1. The S_i are an EXCLUSIVE-OR (XOR or SUM or Σ) function of R(X), the x_i and the y_i are an AND function of the S_i, the E(X)' is simply the x_i and the y_i with zero padding, and then finally C(X)' = R(X)+E(X)'. But for large codes with a lot of error correction capability, this circuit is also not practical.

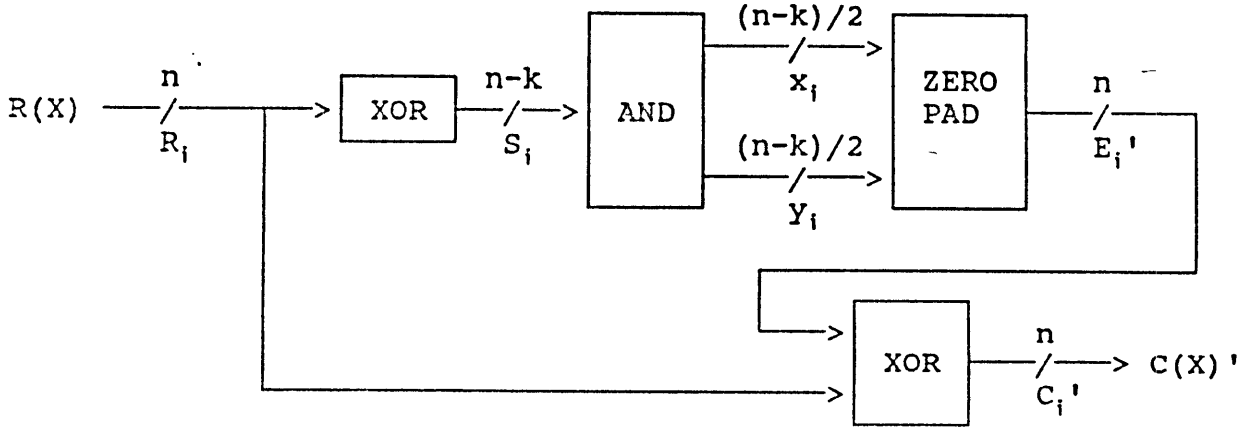


Figure B-1. - Combinational RS decoding circuit.

RS decoders are very complex compared to RS encoders. RS decoder circuits usually employ dedicated processors using parallel processing techniques. Low data rate decoders, especially post processing designs, can be done using sequential techniques. High data rate, real-time RS decoders usually require parallel processing designs. Since most decoding applications are ones with continuous inputs, often pipelining (a form of parallel processing) is easily applied to the sequential stages of RS decoding. If the stages themselves can be partitioned into more of a parallel processing architecture, then even higher data rates along with lower decoder processing delay can be obtained without interleaving. The following sections concentrate on some general design considerations for dedicated and pipelined architectures.

B.1 BCH SHIFT REGISTER SYNDROME CIRCUITS

The SRC to calculate the syndrome $s(X) = s_{2t-1}X^{2t-1} + \ldots + s_1X + s_0$ can be as simple as one of the previous encoder circuits with the input being R(X) instead of M(X). The syndrome components S_i are then found as $S_i = s(\alpha^i)$ by additional circuitry.

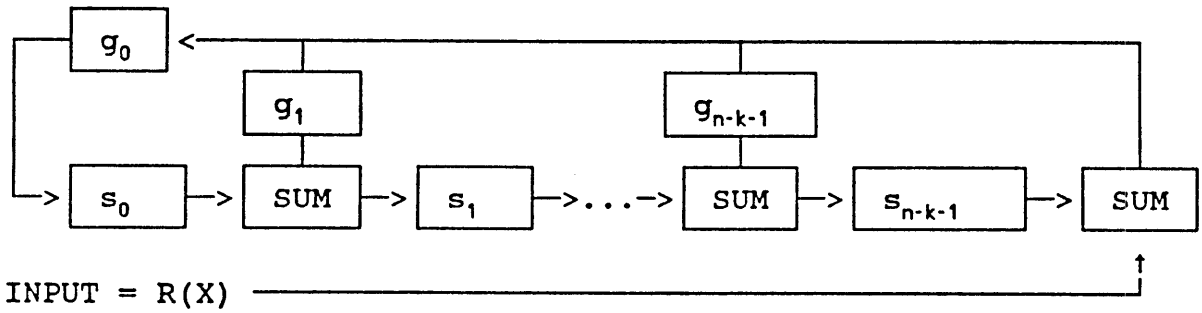


Figure B.1-1. - Syndrome SRC using s(X) = R(X) mod g(X).

We can also send the R(X) into the front of the circuit, but with $s(X)_{register} = s(X)^{[(n-k)shift]}$.

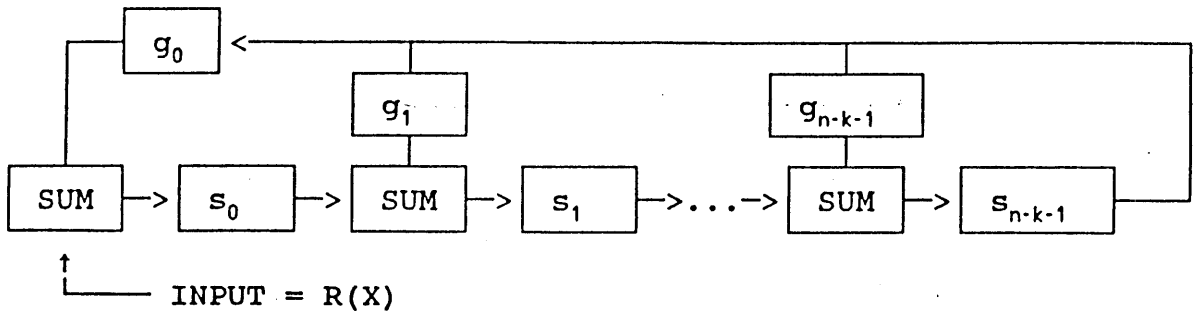


Figure B.1-2. - Another syndrome SRC using s(X) = R(X) mod g(X).

S_i can also be found directly after R(X) has been completely clocked through the following SRC. This SRC also demonstrates that dividing a polynomial by $(X+\alpha^i)$ is the same as evaluating it at α^i; $S_i=R(\alpha^i)$.

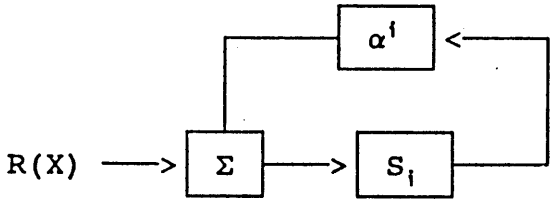


Figure B.1-3. - Syndrome SRC using $S_i = R(\alpha^i)$.

B.2 GENERAL HARDWARE CIRCUITS FOR $\alpha^i \cdot \alpha^j = \alpha^k$

Instead of calculating the remaining decoding steps using only a microcomputer and software, one should consider designing the decoder using more pure hardware, i.e., ROM's, SRC's, XOR gates, AND gates, programmable gate arrays, microslice processors, etc.

$\alpha^i + \alpha^j = \alpha^k$: We already know from sections 1.3.1 and 1.4.5 that addition can be performed using m XOR gates on a tuple-by-tuple basis.

$\alpha^{constant}\alpha^j = \alpha^k$: We can multiply by a single constant $\alpha^{constant}$ by using m XOR trees. These trees are also designed on a tuple-by-tuple basis using such design tools as Karnaugh maps.

$\alpha^i\alpha^j = \alpha^k$: We can multiply any two field symbols together by multiplying and then reducing both the α^i and α^j polynomials'

representations together using algebra. Then, we can build the hardware using just XOR and AND gates for each binary tuple. That is, let $\alpha^i = i_3\alpha^3 + i_2\alpha^2 + i_1\alpha + i_0$ for i_j = either 0 or 1 and j = 0,1,2,...,m-1. Let α^j be likewise defined. Then, the following relationship occurs using the F(X) and α(X) that we used in our (15,9) RS example.

$$\alpha^i\alpha^j = \alpha^{i+j} =$$
$$[(i_0j_3+i_1j_2+i_3j_1+i_3j_0)+i_3j_3]\alpha^3$$
$$+ [(i_0j_2+i_1j_1+i_2j_0)+i_3j_3+(i_3j_2+i_2j_3)]\alpha^2$$
$$+ [(i_0j_1+i_1j_0)+(i_3j_2+i_2j_3)+(i_1j_3+i_2j_2+i_3j_1)]\alpha$$
$$+ [(i_0j_0)+(i_1j_3+i_2j_2+i_3j_1)]$$

$\alpha^i\alpha^j = \alpha^K$: We can multiply any two field symbols α^i and α^j together by using two m-by-m ROM tables to find the logarithms of α^i and α^j which are i and j. Then, we can use an end-around carry adder to ADD and reduce i and j modulo n. After this, use another m-by-m ROM table to take the antilog of K=i+j to determine $\alpha^K = \alpha^{i+j}$. In this circuit neither input can be $0=\alpha^{-\infty}$; however, this can be designed around.

$\alpha^i\alpha^j = \alpha^K$: Multiplying α^i by α^j can also be performed in a manner similar to dividing one symbol using its polynomial representation by the other symbol's polynomial representation. This circuit is similar to the encoding circuits, but use F(X) to determine the taps of the SRC. Load the SRC with α^j, clock i times to multiply by α^i ($i \geq 1$), and the result is in the SRC. Each clock of this SRC is equivalent to counting through the field, i.e., $1,\alpha,\alpha^2,\ldots,\alpha^{n-1}$, $1,\alpha,\alpha^2,\ldots,\alpha^i$. Another SRC which counts by alpha (equivalently multiplies by alpha) is a m-bit parallel clocked shift register with its output multiplied by alpha; use any previous circuit to do the multiplication. Then, after the multiplication, send its output to the shift register's input. The result α^K could then be stored in the SRC if desired.

$\alpha^j / \alpha^i = \alpha^K$: Division between two field symbols α^i and α^j can be done by taking the denominator α^i and sending it through a 2^m-by-m ROM. This determines its inverse α^{-i}. Then, $\alpha^j / \alpha^i = \alpha^K$ can be found by multiplying α^{-i} and α^j together using any valid multiplication design. The denominator input cannot be zero.

$\alpha^j / \alpha^i = \alpha^K$: Division can also be done by subtracting logarithms. We can divide any two field symbols α^i and α^j (except $\alpha^i=0$) together by using two m-by-m ROM tables to find the logarithms of α^i and α^j which are i and j. Then, we can use an end-around carry adder to SUBTRACT and reduce i and j modulo n. After this, use another m-by-m ROM table to take the antilog of K=j-i to determine $\alpha^K = \alpha^{j-i}$.

B.3 DIRECT METHOD HARDWARE FOR $\sigma(X)$

The direct method requires fewer calculations than the iterative method for T = 0,1,2,3, and maybe 4 error symbols. So we may want to have our hardware solve for $\sigma(X)$ using the direct method for when a few error symbols are thought to have occurred. When a lot of errors are thought to have occurred we should possibly use some other method applied into hardware. Also a hardware system might run different algorithms simultaneously and the first one done passes its $\sigma(X)$ solution to the next decoding stage; this does not seem synchronous and efficient.

B.3.1 Previously Calculated Determinants

In section 4.3.2, we start out trying to solve a t-by-t matrix. If DET $|S\sigma|_t = 0$, then we keep trying the next smaller matrix until DET $|S\sigma|_i$ for i<t is not zero; the largest non-zero determinant determines how many errors is thought to have taken place (i.e., determines T). It should be noted that once the determinant of a t-by-t matrix has been calculated, it is possible for all the remaining determinants for the i-by-i matrix, i<t, to also have been partly calculated. Thus, these results should be placed in a temporary memory location and the hardware may be able to save time or amount of hardware not recalculating terms of the smaller determinants. Using the familiar (15,9) RS example (section 4.3.2), let T=2. Then, instead of choosing the first two equations as in section 4.3.2, we will choose either the middle two or the last two equations from the following:

$$S_1\sigma_2 + S_2\sigma_1 = S_3$$
$$S_2\sigma_2 + S_3\sigma_1 = S_4$$
$$S_3\sigma_2 + S_4\sigma_1 = S_5$$
$$S_4\sigma_2 + S_5\sigma_1 = S_6$$

Wσ get either set of these equations:

$$S_2\sigma_2 + S_3\sigma_1 = S_4 \qquad\qquad S_3\sigma_2 + S_4\sigma_1 = S_5$$
$$S_3\sigma_2 + S_4\sigma_1 = S_5 \qquad\qquad S_4\sigma_2 + S_5\sigma_1 = S_6$$

The corresponding matrices are $|S\sigma|_{2,3}$ and $|S\sigma|_{3,4}$.

$$|S\sigma|_{2,3} = \begin{vmatrix} S_2 & S_3 \\ S_3 & S_4 \end{vmatrix} \qquad\qquad |S\sigma|_{3,4} = \begin{vmatrix} S_3 & S_4 \\ S_4 & S_5 \end{vmatrix}$$

The determinants of these matrices have already been calculated and stored into memory at the time when the determinant of the t-by-t matrix was calculated. Let us show this. When T=t=3,

we get some equations.

$$S_1\sigma_3 + S_2\sigma_2 + S_3\sigma_1 = S_4$$
$$S_2\sigma_3 + S_3\sigma_2 + S_4\sigma_1 = S_5$$
$$S_3\sigma_3 + S_4\sigma_2 + S_5\sigma_1 = S_6$$

$$\text{Let } |S\sigma|_t = \begin{vmatrix} S_1 & S_2 & S_3 \\ S_2 & S_3 & S_4 \\ S_3 & S_4 & S_5 \end{vmatrix}$$

The corresponding determinant is DET $|S\sigma|_t$.

$$\text{DET } |S\sigma|_t = \text{DET} \begin{vmatrix} S_1 & S_2 & S_3 \\ S_2 & S_3 & S_4 \\ S_3 & S_4 & S_5 \end{vmatrix}$$

Let us calculate this by arbitrarily taking the determinant across the first row.

$$\text{DET } |S\sigma|_t = S_1 \text{ DET} \begin{vmatrix} S_3 & S_4 \\ S_4 & S_5 \end{vmatrix} + S_2 \text{ DET} \begin{vmatrix} S_2 & S_4 \\ S_3 & S_5 \end{vmatrix} + S_3 \text{ DET} \begin{vmatrix} S_2 & S_3 \\ S_3 & S_4 \end{vmatrix}$$

$$\text{DET } |S\sigma|_t = S_1 \text{ DET } |S\sigma|_{3,4} + S_2 \text{ DET} \begin{vmatrix} S_2 & S_4 \\ S_3 & S_5 \end{vmatrix} + S_3 \text{ DET } |S\sigma|_{2,3}$$

Notice that DET $|S\sigma|_{3,4}$ and DET $|S\sigma|_{2,3}$ was calculated when DET $|S\sigma|_t$ was calculated. Therefore, it is shown that some determinants of a smaller matrix can be calculated if the determinant of its larger matrix has been calculated.

There exists one way to calculate DET $|S\sigma|_t$ to be able to include all the four equations (i.e., the S_3,S_4,S_5, and S_6 equations in the beginning of this section), but not all the combinations of them. Let us show this by continuing this example. Calculate DET $|S\sigma|_t$ by starting at the second row instead of the first.

$$\text{DET } |S\sigma|_t = S_2 \text{ DET} \begin{vmatrix} S_2 & S_3 \\ S_4 & S_5 \end{vmatrix} + S_3 \text{ DET} \begin{vmatrix} S_1 & S_3 \\ S_3 & S_5 \end{vmatrix} + S_4 \text{ DET} \begin{vmatrix} S_1 & S_2 \\ S_3 & S_4 \end{vmatrix}$$

$$\text{DET } |S\sigma|_t = S_2 \text{ DET } |S\sigma|_{2,4} + S_3 \text{ DET} \begin{vmatrix} S_1 & S_3 \\ S_3 & S_5 \end{vmatrix} + S_4 \text{ DET } |S\sigma|_{1,3}$$

Therefore, to save time and/or other resources by not recalculating all the determinants' terms, select the particular equations which correspond to how the DET $|S\sigma|_t$ is calculated. Designing how this is calculated might come in handy when designing the hardware. If it's easy to implement, try not to recalculate something unless you need to.

B.3.2 Reduced $\sigma(X)$ Computations

There is a shortened method to solve the set of equations for $\sigma(X)$. Let us use the direct method in section 4.3.2, but reduce the number of multiplies, divides, and adds. Follow the following example which is similar to the (15,9) RS example with T=t=3.

$$|S\sigma|_t = \begin{vmatrix} S_1 & S_2 & S_3 \\ S_2 & S_3 & S_4 \\ S_3 & S_4 & S_5 \end{vmatrix}$$

$$\begin{aligned} \text{DET } |S\sigma|_t &= S_1[S_3S_5+(S_4)^2]+S_2[S_2S_5+S_3S_4]+S_3[S_2S_4+(S_3)^2] \\ &= S_1S_3S_5+S_1(S_4)^2+(S_2)^2S_5+S_2S_3S_4+S_2S_3S_4+(S_3)^3 \end{aligned}$$

And now notice that we can save 4 multiplies and 2 adds by noticing that $\alpha^i+\alpha^i=0$ in modulo-2 math.

$$\text{DET } |S\sigma|_t = S_1S_3S_5+S_1(S_4)^2+(S_2)^2S_5+(S_3)^3$$

$$\sigma_1 = (\text{DET } |S\sigma|_t)^{-1} \text{ DET } \begin{vmatrix} S_1 & S_2 & S_4 \\ S_2 & S_3 & S_5 \\ S_3 & S_4 & S_6 \end{vmatrix}$$

$$\sigma_1 = (\text{DET } |S\sigma|_t)^{-1}[S_1S_3S_6+S_1S_4S_5+(S_2)^2S_6+S_2S_3S_5+S_2(S_4)^2+(S_3)^2S_4]$$

Well, no terms dropped out here. Similarly, σ_2 is found.

$$\sigma_2 = (\text{DET } |S\sigma|_t)^{-1}[S_1(S_5)^2+S_1S_4S_6+S_2S_4S_5+S_3(S_4)^2+S_2S_3S_6+(S_3)^2S_5]$$

Well, no terms dropped out here either. Similarly, but with two terms dropping out, σ_3 is found.

$$\begin{aligned} \sigma_3 &= (\text{DET } |S\sigma|_t)^{-1}[S_3S_4S_5+(S_4)^3+S_2(S_5)^2+S_2S_4S_6+S_3S_4S_5+(S_3)^2S_6] \\ &= (\text{DET } |S\sigma|_t)^{-1}[(S_4)^3+S_2(S_5)^2+S_2S_4S_6+(S_3)^2S_6] \end{aligned}$$

This circuit requires 40 multiplies, 3 divisions, and 16 adds to solve for $\sigma(x)$ or $\sigma_r(X)$ when T=t in a (15,9) RS code using Cramer's rule. If we care to do more algebra, we can find that we can reduce the number of multiplies, divides, and adds even further.

B.4 HIGH DATA RATE RS DECODING DESIGN

There are real-time processing decoders and there are post processing decoders. Post processing RS decoders are ones which possess an increasing decoding delay as the number of errors to correct increases. Typical post processing RS decoders are software programs executed on most any type of computer.

Post processing hardware are rarely designed for real-time systems. This is because these systems introduce buffer overflow conditions (or possible loss of data) and adds to the overall decoding delay and complexity of the system. Real-time systems almost always use real-time decoders. Real-time decoders have a fixed decoding delay which is usually small and is independent of the number of errors to correct.

Real-time RS decoders usually take advantage of the pipelined nature of the decoding stages so that high data rates can often be accommodated without being forced to use interleaving techniques. If we do not take advantage of these major partitions, we must wait longer before we can start decoding another received word; if there are long time periods between each received word, then the overall data rate decreases. Pipelining techniques are often used to design high data rate RS decoders which are often found in real-time systems. Pipelining is partitioning a sequential process into many smaller processes such that dedicated circuits efficiently and simultaneously work these smaller processes in a sequential manner. High data rate designs are similar to figure B.4-2 while low data rate ones are similar to figure B.4-1.

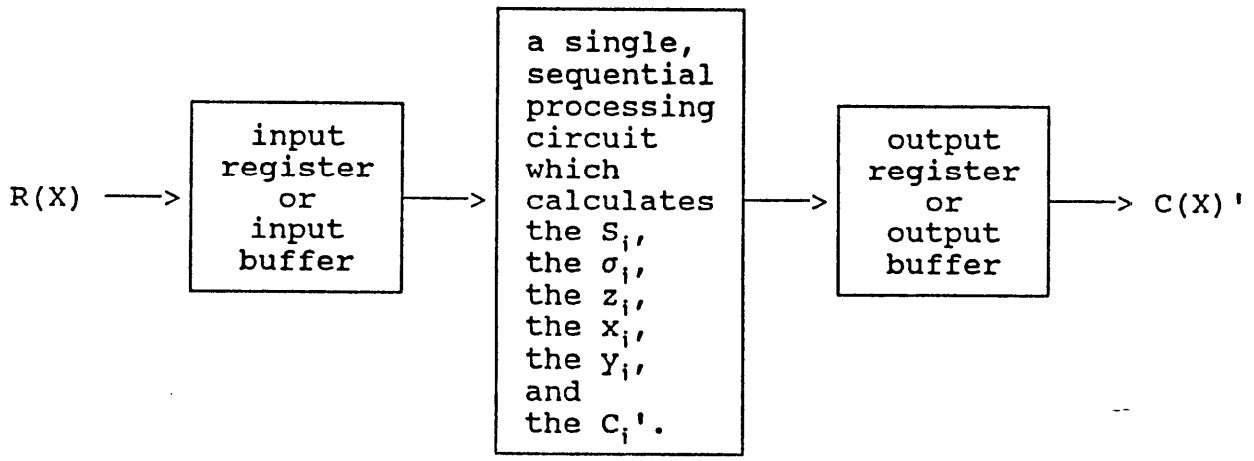


Figure B.4-1. - Low data rate RS single processor decoder.

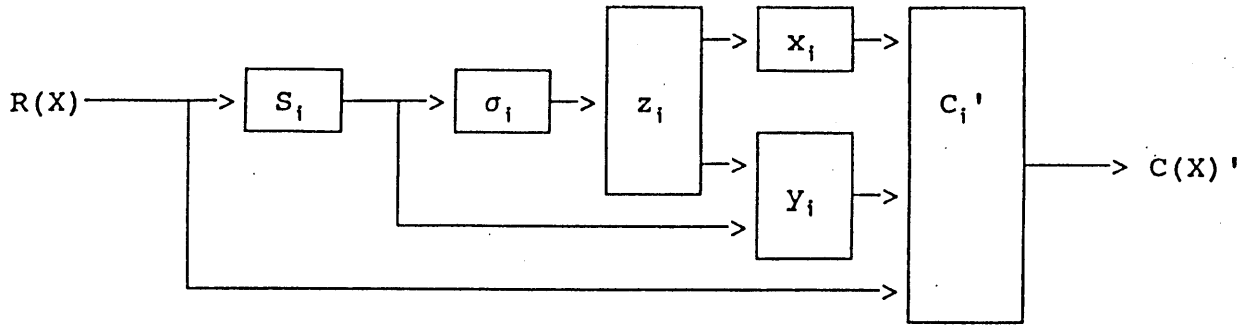


Figure B.4-2. - High data rate RS pipelined decoder.

If we have a low data rate RS decoder, then we might design one encompassing processing circuit. This circuit would be responsible for calculating the S_i, σ_i, z_i, x_i, y_i, and C_i' for R(X) before it would accept another R(X) to be decoded. This single circuit might be a general purpose microprocessor or a microslice processor or comprised of many commercially available discrete parts or maybe even some type of computer executing some type of software. Figure B.4-1 demonstrates the single processor design. For a real-time system, the input register must be at least as large as the certain constant number of clock cycles required to calculate the S_i, σ_i, z_i, x_i, y_i, and C_i' for the largest number of errors to decode. Low data rate, real-time decoders also have larger input registers than their high data rate, pipeline counterparts. For a real-time system, the output register will contain the decoded code word C(X)'. Due to the decoder delay, the registered output is synchronously bursty, i.e., there are long time periods between decoded code words. If it is desired to have a continuous output data rate, then we can smooth the output either by designing a more complex output register or by synonymously using a small first in, first out (FIFO) memory circuit.

If we have a high data rate RS decoder, then we would probably design a processing circuit with multiple processors. Each processor would only work on a separate portion of the overall process. Pipelining speeds up the process along with smaller input registers and decreasing decoder delay. One processor would be efficiently designed to specifically calculate the S_i, another for the σ_i, and others for the z_i, x_i, y_i, and C_i'. This pipeline process allows, for example, the new S_i to be calculated simultaneously while the σ_i from the old S_i are being calculated. These multiple processors might be SRC's, dedicated hardware, or commercially available processors. Figure B.4-2 demonstrates this multiple processing (really parallel processing) circuit. For a real-time system, a pipeline design can be designed without an

output register; if the parity-check symbols are not output [i.e., only the message symbols (and possibly a decoder status word) are output] and a continuous output data rate is not required, then an output register is not needed to smooth the output. Real-time processing RS decoders usually require registers (i.e., small time delays before some pipelined circuits) while post processing RS decoders usually require buffers and/or files.

A typical pipelined, high data rate decoder often uses SRC's to implement each stage. The SRC to calculate the S_i can be almost the same as the encoder circuit (see section B.1). The SRC to calculate the σ_i is often patterned after Berlekamp's iterative algorithm and Euclidean greatest common divisor algorithm. The σ_i determination is the most complicated and usually requires the most processing time. Therefore, it is the data rate limiter of most systems. The z_i values can be calculated quickly using a SRC and using the Chien search with a parallel format implementation. If the σ_i SRC requires more processing time than a z_i SRC implemented in a sequential format, then a sequential z_i SRC (rather than one in a parallel format) might be used to save size, weight, and power. The x_i circuit can be simplified slightly by letting the primitive element α^G used in the code word generator be $\alpha^G=\alpha^1$. The y_i circuit should be designed to require less processing time than the limiting process (which is often the σ_i circuit). The final SRC is the C_i' circuit which is nothing more than a time delayed version of the received symbols R_i shifted out after some of the R_i have been added to the y_i at the x_i.

The partitioning between the stages is illustrated in figure B.4-2. For some of the stages, it is possible to start the process of calculating its set of values when the previous stage has not yet completely calculated all of its values. For example, it is possible for the C_i' circuit to start to output the decoded code word symbols C_i' at the time when the most significant error location x_T and the associated error value y_T had been calculated AND still maintained a continuous output. A correctly just-in-time architecture should add to the degree of pipelined (or parallel) processing.

Besides the previous discussions of stage partitioning, partitioning of the stages themselves are possible. Hypersystolic array designs are examples of this. These hypersystolic array designs partition the processing of each stage into many cells (or computational units) which add to the degree of parallel processing; therefore, even higher data rates along with less decoder delay results. See "Hypersystolic Reed-Solomon Decoder Final Report" within the references section.

For a high data rate (real-time) system, a pipelined design that is

globally synchronous may be preferred. For an ultra high data rate (real-time) system, a design with even more parallel processing that is globally asynchronous, but locally synchronous, e.g., the hypersystolic designs, may be preferred. Low data rate systems might be designed using a SRC, a general purpose processor, or by running some high level (or low level) computer language program on any available computer. High data rate systems usually demand some type of parallel processing implemented directly into a SRC.

APPENDIX C
MATRICES AND RS CODING

People sometimes prefer to work with matrices to reduce the algebraic load. Using matrices can come in handy when developing software and possibly hardware. I sometimes prefer matrices because it's sometimes easier to see how the code is working.

C.1 RS ENCODING USING MATRICES

Start off with the generator matrix g. g is a k-by-n matrix $g_{k\text{-by-}n}$. In RS coding it must be constructed directly from the generator polynomial $g(X) = X^6+\alpha^{10}X^5+\alpha^{14}X^4+\alpha^4X^3+\alpha^6X^2+\alpha^9X+\alpha^6$. To simplify the notation, let $-\infty = 0$, $0 = 1$, $1 = \alpha$, $2 = \alpha^2$, ..., $n-1 = \alpha^{n-1}$. Therefore, g(X) = [6 9 6 4 14 10 0]. Notice that since we are working with matrices, I decided to use the mathematical convention of writing in increasing order of magnitude, i.e., $g(X) = [g_0\ g_1\ \ldots\ g_{n-k}]$ and not $g(X) = [g_{n-k}\ g_{n-k-1}\ \ldots\ g_0]$. The non-systematic generator matrix $g_{non\text{-}sys,k\text{-}by\text{-}n} = g_{non\text{-}sys}$ is obtained from the generator g(X) = [6 9 6 4 14 10 0].

$$g_{non\text{-}sys} = \begin{vmatrix} 6 & 9 & 6 & 4 & 14 & 10 & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & 6 & 9 & 6 & 4 & 14 & 10 & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & 6 & 9 & 6 & 4 & 14 & 10 & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & 6 & 9 & 6 & 4 & 14 & 10 & 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & 6 & 9 & 6 & 4 & 14 & 10 & 0 & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & 6 & 9 & 6 & 4 & 14 & 10 & 0 & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 6 & 9 & 6 & 4 & 14 & 10 & 0 & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 6 & 9 & 6 & 4 & 14 & 10 & 0 & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 6 & 9 & 6 & 4 & 14 & 10 & 0 \end{vmatrix}$$

The systematic generator matrix g_{sys} is directly constructed from the non-systematic generator matrix $g_{non\text{-}sys}$ by standard matrix row operations; one row multiplied by α^i and then added to another row and the result replacing the row added to. To acquire the systematic form, we want a k-by-(n-k) Parity matrix $P_{k\text{-by-}(n\text{-}k)}$ joined with the k-by-k identity matrix I_k, i.e., $g_{sys,k\text{-by-}k} = [P_{k\text{-by-}(n\text{-}k)}\ I_k]$.

In other words, transform the $g_{non\text{-}sys}$ into the g_{sys} shown below.

$$g_{sys} = \begin{vmatrix} 6 & 9 & 6 & 4 & 14 & 10 & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ 1 & 12 & 3 & 8 & 14 & 12 & -\infty & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ 3 & 11 & 10 & 9 & 7 & 1 & -\infty & -\infty & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ 7 & 12 & 8 & 0 & 7 & 8 & -\infty & -\infty & -\infty & 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\ 14 & 12 & 5 & 0 & 9 & 4 & -\infty & -\infty & -\infty & -\infty & 0 & -\infty & -\infty & -\infty & -\infty \\ 4 & 14 & 12 & 1 & 9 & 9 & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & -\infty & -\infty & -\infty \\ 7 & 6 & 4 & 14 & 11 & 4 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & -\infty & -\infty \\ 2 & 13 & 0 & 3 & 4 & 10 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & 8 & 1 & 4 & 3 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 \end{vmatrix}$$

This is clearly of the form $g_{sys,k\text{-}by\text{-}k} = [P_{k\text{-}by\text{-}(n\text{-}k)}\ I_k]$ where $P_{k\text{-}by\text{-}(n\text{-}k)}$ is:

$$P_{k\text{-}by\text{-}(n\text{-}k)} = \begin{vmatrix} 6 & 9 & 6 & 4 & 14 & 10 \\ 1 & 12 & 3 & 8 & 14 & 12 \\ 3 & 11 & 10 & 9 & 7 & 1 \\ 7 & 12 & 8 & 0 & 7 & 8 \\ 14 & 12 & 5 & 0 & 9 & 4 \\ 4 & 14 & 12 & 1 & 9 & 9 \\ 7 & 6 & 4 & 14 & 11 & 4 \\ 2 & 13 & 0 & 3 & 4 & 10 \\ -\infty & -\infty & 8 & 1 & 4 & 3 \end{vmatrix}$$

and where the identity matrix I_k is:

$$I_k = \begin{vmatrix} 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & 0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & 0 & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & 0 & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & 0 \end{vmatrix}$$

Performing matrix row operations within a finite field is the same as we usually perform matrix row operations within a infinite field. However, we use the addition and multiplication methods for finite fields, not infinite fields. Let us work an example of calculating the first two rows of the g_{sys} presented previously. We first start at the first row of $g_{non\text{-}sys}$, denoted $g_{non\text{-}sys,row0}$. From the previous $g_{non\text{-}sys}$ we obtain the following:

$$g_{non\text{-}sys,row0} = [\ 6\ \ 9\ \ 6\ \ 4\ 14\ 10\ \ 0\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ ]$$

Therefore, the coefficients of "row0" are:

$g_{non\text{-}sys,row0,0} = 6$
$g_{non\text{-}sys,row0,1} = 9$
$g_{non\text{-}sys,row0,2} = 6$
etc.
$g_{non\text{-}sys,row0,n-k-1} = g_{non\text{-}sys,row0,5} = 10$
$g_{non\text{-}sys,row0,n-k} = g_{non\text{-}sys,row0,6} = 0$
$g_{non\text{-}sys,row0,n-k+1} = g_{non\text{-}sys,row0,7} = -\infty$
etc.
$g_{non\text{-}sys,row0,n-1} = g_{non\text{-}sys,row0,14} = -\infty$.

Now the first row, denoted row0, is the same for both the systematic and non-systematic generators because $g_{non\text{-}sys,row0,n-k} = g_{non\text{-}sys,row0,6} = \alpha^0 = 1$ and $g_{non\text{-}sys,row0,i} = \alpha^{-\infty} = 0$ for $i = n-k+1, n-k+2, \ldots, n-1 = 7,8,\ldots,14$. Therefore, $g_{sys,row0} = g_{non\text{-}sys,row0} =$ [6 9 6 4 14 10 0 $-\infty$ $-\infty$ $-\infty$ $-\infty$ $-\infty$ $-\infty$ $-\infty$ $-\infty$].

To find the second row of g_{sys}, denoted $g_{sys,row1}$, we do standard matrix row operations.

$$
\begin{aligned}
g_{sys,row1} &= g_{non\text{-}sys,row0,n-k-1} g_{non\text{-}sys,row0} + g_{non\text{-}sys,row1} \\
&= 10[\ 6\ 9\ 6\ 4\ 14\ 10\ 0\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ ] \\
&\quad + g_{non\text{-}sys,row1} \\
&= [\ 16\ 19\ 16\ 14\ 24\ 20\ 10\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ ] \\
&\quad + g_{non\text{-}sys,row1} \\
&= [\ 1\ 4\ 1\ 14\ 9\ 5\ 10\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ ] \\
&\quad + g_{non\text{-}sys,row1} \\
&= [\ 1\ 4\ 1\ 14\ 9\ 5\ 10\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ ] \\
&\quad + [\ -\infty\ 6\ 9\ 6\ 4\ 14\ 10\ 0\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ ] \\
&= [\ (1+-\infty)\ (4+6)\ (1+9)\ (14+6)\ (9+4)\ (5+14)\ (10+10)\ (-\infty+0) \\
&\quad (-\infty+-\infty)\ (-\infty+-\infty)\ (-\infty+-\infty)\ (-\infty+-\infty)\ (-\infty+-\infty)\ (-\infty+-\infty)\ (-\infty+-\infty)\ ] \\
&= [\ (1)\ (12)\ (3)\ (8)\ (14)\ (12)\ (-\infty)\ (0)\ (-\infty)\ (-\infty)\ (-\infty) \\
&\quad (-\infty)\ (-\infty)\ (-\infty)\ (-\infty)\ ] \\
&= [1\ 12\ 3\ 8\ 14\ 12\ -\infty\ 0\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty]
\end{aligned}
$$

The result is the i^{th} row of g_{sys}, denoted $g_{sys,rowi}$, when $g_{sys,rowi,n-k+i} = 0$ and $g_{sys,rowi,j} = -\infty$ for $j = n-k, n-k+1, \ldots, n-k+i-1$ and for $j = n-k+i+1, n-k+i+2, \ldots, n-1$. The result that we have just obtained, is the second row denoted $g_{sys,row1}$. This is because $g_{sys,row1,7} = \alpha^0 = 1$ and $g_{sys,row1,j} = \alpha^{-\infty} = 0$ for $j = 6$ and for $j = 8,9,\ldots,14$. It should be noted that more and more iterations of the previous procedure are needed to obtain $g_{sys,rowi}$ for i increasing.

The parity-check polynomial $h(X) = X^n+1 \ / \ g(X)$ has a corresponding parity-check matrix $h_{(n-k)\text{-}by\text{-}n}$. h can be either systematic or non-systematic, but must be in accordance with its partner g; just as h(X) is related to g(X), h must somehow also be related to g.

$h_{non\text{-}sys}$ can be found from $h(X) = [h_0\ h_1\ \ldots\ h_{last\text{-}tuple}]$ in the same manner that g can be found from $g(X) = [g_0\ g_1\ \ldots\ g_{2t}]$. Once either g or h is found the other is specified; $g_{k\text{-}by\text{-}k} = [P_{k\text{-}by\text{-}(n\text{-}k)}\ I_k]$ and $h_{(n\text{-}k)\text{-}by\text{-}n} = [I_{n\text{-}k}\ P_{k\text{-}by\text{-}(n\text{-}k)}{}^T]$ where $P_{k\text{-}by\text{-}(n\text{-}k)}{}^T$ is the transpose of $P_{k\text{-}by\text{-}(n\text{-}k)}$.

The message M can be encoded directly into the code word C using g; $C_{1\text{-}by\text{-}n} = M_{1\text{-}by\text{-}k}g_{k\text{-}by\text{-}(n\text{-}k)}$. Use the (15,9) RS example to demonstrate this. From chapter 4, M = [-∞ 11 -∞ -∞ -∞ -∞ -∞ -∞ -∞].

$$C_{sys} = Mg_{sys} = [-\infty\ 11\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty]g_{sys}$$

$g_{sys} = g_{systematic}$ was previously calculated within this appendix. Once this matrix calculation is completed, then C_{sys} is generated.

$$C_{sys} = [12\ 8\ 14\ 4\ 10\ 8\ -\infty\ 11\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty]$$

This result checks with the results obtained in chapter 4; $C(X) = \alpha^{11}X^7 + \alpha^8X^5+\alpha^{10}X^4+\alpha^4X^3+\alpha^{14}X^2+\alpha^8X+\alpha^{12}$.

C.2 RS DECODING USING MATRICES

The syndrome s can be calculated directly from the received word R using h^T.

$$s_{1\text{-}by\text{-}(n\text{-}k)} = R_{1\text{-}by\text{-}n}h_{n\text{-}by\text{-}(n\text{-}k)}{}^T$$

To find the errors and their values notice that $R_{1\text{-}by\text{-}n} = C_{1\text{-}by\text{-}n} + E_{1\text{-}by\text{-}n}$ where E is the error word. Now $s = Rh^T = [C+E]h^T = Ch^T+Eh^T$. It is a fact that $s=Ch^T=0$ because h is the parity-check of C. Therefore, $s = Ch^T+Eh^T = Eh^T$. THEREFORE, THE SYNDROME IS A FUNCTION ONLY OF THE ERROR PATTERN (OR WORD) AND NOT THE RECEIVED WORD!

Wait a second, $s=Rh^T=Eh^T$; but this does not mean R=E. So, in our calculations, we determine the value of the syndrome s by $s=Rh^T$ and then find E with the fewest non-zero m-tuples such that $s=Eh^T$. This fewest number of errors idea should be familiar to us by now; it is MLD. The problem is to come up with an efficient algorithm which determines this E. We could start with no errors (T=0) and if $s=Rh^T=0$, then no errors occurred.

But what if one error symbol occurred? Then, $s=Rh^T=[s_{FCR}\ s_{FCR+1}\ \ldots\ s_{FCR+n-k-1}] \neq 0$ where s_i are the coefficients of s(X). So we need to start calculating $s=Eh^T$ for each n^2 single, non-zero error symbol possibilities of E until we get this particular value of s. Notice the worst case condition is

performing the calculations for all the n^2 possibilities or storing all n^2 possibilities in a table.

But what if two error symbols occurred? Then if $S_i \neq 0$, the n^2 single error symbol possibilities in $s=Eh^T$ would not equal $s=Rh^T$. So then we try all the $(n^4-n^3)/2$ possibilities approximately n^4 double, non-zero error symbol possibilities. One and only one solution of E exists with the fewest number of error symbols. If there is no solution, then we need to keep calculating $s=Eh^T$ for more and more errors ($T \leq t$) until a solution is reached. You can imagine the number of calculations which are needed to be performed if we have a lot of errors. That is why iterative algorithms are very popular. The number of possible combinations of T, non-zero error symbols is:

$$\binom{n}{T}(P^m-1)^T = (n!/(T!(n-T)!))(P^m-1)^T$$

Even super computing usually cannot determine all of these possible combinations to store into huge tables which are impractical. Continuously calculating $s=Eh^T$ for arbitrary error patterns until a solution is obtained is also usually impractical. However, we can check our results from the previous chapters. Let us calculate $s=Rh^T$ which should equal $s=Eh^T$; denote $s=Rh^T$ as $s_R=Rh^T$ and $s=Eh^T$ as $s_E=Eh^T$. The received word from chapter 4 is R = [12 8 3 4 10 8 $-\infty$ 11 0 $-\infty$ $-\infty$ $-\infty$ $-\infty$ $-\infty$ $-\infty$].

$$s_R = Rh^T$$
$$= [12\ 8\ 3\ 4\ 10\ 8\ -\infty\ 11\ 0\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty]h^T$$

Let us find h^T. $h = h_{sys} = [I_{n-k}\ P_{k\text{-by-}(n-k)}{}^T]$

$$h_{sys} = \begin{vmatrix} 0 & -\infty & -\infty & -\infty & -\infty & -\infty & 6 & 1 & 3 & 7 & 14 & 4 & 7 & 2 & -\infty \\ -\infty & 0 & -\infty & -\infty & -\infty & -\infty & 9 & 12 & 11 & 12 & 12 & 14 & 6 & 13 & -\infty \\ -\infty & -\infty & 0 & -\infty & -\infty & -\infty & 6 & 3 & 10 & 8 & 5 & 12 & 4 & 0 & 8 \\ -\infty & -\infty & -\infty & 0 & -\infty & -\infty & 4 & 8 & 9 & 0 & 0 & 1 & 14 & 3 & 1 \\ -\infty & -\infty & -\infty & -\infty & 0 & -\infty & 14 & 14 & 7 & 7 & 9 & 9 & 11 & 4 & 4 \\ -\infty & -\infty & -\infty & -\infty & -\infty & 0 & 10 & 12 & 1 & 8 & 4 & 9 & 4 & 10 & 3 \end{vmatrix}$$

And therefore, $h^T = h_{sys}{}^T$. It should be noted that the minimum distance d_{min} is the smallest number of columns of h (or rows of h^T) that sum to zero; this is a linear block code corollary and thus also works for RS codes. Notice that in this example, $d_{min} = 2t+1$ should be seven symbols. There are only n-k=6 rows of h_{sys} and because of the identity matrix, no six or fewer unique rows add to zero. Therefore, this only shows us that $d_{min} > 6$ and it is; $d_{min}=7$. This fact should serve as a good check for a valid h (and thus g). Getting back to this example, $h^T = h_{sys}{}^T$.

$$h^T = \begin{vmatrix} 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & 0 & -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & 0 & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & 0 & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & 0 \\ 6 & 9 & 6 & 4 & 14 & 10 \\ 1 & 12 & 3 & 8 & 14 & 12 \\ 3 & 11 & 10 & 9 & 7 & 1 \\ 7 & 12 & 8 & 0 & 7 & 8 \\ 14 & 12 & 5 & 0 & 9 & 4 \\ 4 & 14 & 12 & 1 & 9 & 9 \\ 7 & 6 & 4 & 14 & 11 & 4 \\ 2 & 13 & 0 & 3 & 4 & 10 \\ -\infty & -\infty & 8 & 1 & 4 & 3 \end{vmatrix}$$

So get-back to s_R

$$\begin{aligned} s_R &= Rh^T \\ &= [12\ 8\ 3\ 4\ 10\ 8\ -\infty\ 11\ 0\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty]h^T \\ &= [3\ 11\ 5\ 9\ 7\ 1] \\ &= [s_0\ s_1\ s_2\ s_3\ s_4\ s_5] \end{aligned}$$

Therefore, $s_0=\alpha^3$, $s_1=\alpha^{11}$, $s_2=\alpha^5$, $s_3=\alpha^9$, $s_4=\alpha^7$, and $s_5=\alpha$; $s(X) = R(X) \bmod g(X) = s_{2t-1}X^{2t-1}+\ldots+s_1X+s_0$. Now let us check our error pattern decoded in chapter 4. The error word from chapter 4 is $E = [-\infty\ -\infty\ 0\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ 0\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty]$. Let us calculate $s_E = Eh^T$ and make sure $s_R=s_E$.

$$\begin{aligned} s_E &= Eh^T \\ &= [-\infty\ -\infty\ 0\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ 0\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty\ -\infty]h^T \\ &= [3\ 11\ 5\ 9\ 7\ 1] \end{aligned}$$

Now does $s_E = s_R = [s_0\ s_1\ s_2\ s_3\ s_4\ s_5] = [3\ 11\ 5\ 9\ 7\ 1]$? Yes, they are both the same!!!

Great! Syndrome calculations can also be done using matrices; sometimes matrices are easier to work with. But to be more sure of having calculated the g and h matrices correctly, let us calculate the syndrome components S_i and verify them; $S_i = s(\alpha^i)$.

$$s(X) = \sum_{i=0}^{2t-1} s_i X^i$$

$$\begin{aligned} s(X) &= s_0 + s_1X + s_2X^2 + \ldots + s_{2t-1}X^{2t-1} \\ &= s_0 + s_1X + s_2X^2 + s_3X^3 + s_4X^4 + s_5X^5 \\ &= \alpha^3 + \alpha^{11}X + \alpha^5X^2 + \alpha^9X^3 + \alpha^7X^4 + \alpha X^5 \end{aligned}$$

$S_1 = s(\alpha) = 3+11\cdot1+5\cdot2+9\cdot3+7\cdot4+1\cdot5 = 0$ Therefore, $S_1=\alpha^0=1$.

$S_2 = s(\alpha^2) = 3+11\cdot2+5\cdot4+9\cdot6+7\cdot8+1\cdot10 = 0$ Therefore, $S_2=\alpha^0=1$.

$S_3 = s(\alpha^3) = 3+11\cdot3+5\cdot6+9\cdot9+7\cdot12+1\cdot15 = 5$ Therefore, $S_3=\alpha^5$.

$S_4 = s(\alpha^4) = 0$ Therefore, $S_4=\alpha^0=1$.

$S_5 = s(\alpha^5) = -\infty$ Therefore, $S_5=\alpha^{-\infty}=0$.

$S_6 = s(\alpha^6) = 10$ Therefore, $S_6=\alpha^{10}$.

These results agree with the ones in chapter 4. Also, the S_i can be represented as $S_i = [S_{FCR} \; S_{FCR+1} \; \cdots \; S_{2t+FCR-1}]$; e.g., $S_i = [S_1 \; S_2 \; S_3 \; S_4 \; S_5 \; S_6] = [1 \; 1 \; a^5 \cdot 1 \; 0 \; a^{10}] = [0 \; 0 \; 5 \; 0 \; -\infty \; 10]$.

Now decode the message!

$C_{sys}' = R + E$

$= [12 \; 8 \; 3 \; 4 \; 10 \; 8 \; -\infty \; 11 \; 0 \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty]$

$+[-\infty \; -\infty \; 0 \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty \; 0 \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty]$

$= [12 \; 8 \; (3+0) \; 4 \; 10 \; 8 \; -\infty \; 11 \; (0+0) \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty]$

$= [12 \; 8 \; 14 \; 4 \; 10 \; 8 \; -\infty \; 11 \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty]$

$= [CK_{1\text{-by-}(n-k)}' \; M_{1\text{-by-}k}']$

C_{sys}' should equal C_{sys}. In fact, it does if $T \leq t$. Anyway, our estimate of the message M' is extracted from $C' = [12 \; 8 \; 14 \; 4 \; 10 \; 8 \; -\infty \; 11 \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty]$. From chapter 4, $M = [-\infty \; 11 \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty]$ which agrees with the decoded message $M' = [-\infty \; 11 \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty \; -\infty]$. Again from chapter 4, $CK = [12 \; 8 \; 14 \; 4 \; 10 \; 8]$ which also agrees with the decoded parity-check $CK' = [12 \; 8 \; 14 \; 4 \; 10 \; 8]$. Therefore, we can do these operations in matrix form if we desire to.

APPENDIX D
A GENERAL MATHEMATICAL OVERVIEW OF RS CODING

This appendix assumes some understanding of the terminology and relates to primitive (n,k) RS codes over $GF(2^m)$ in a non-erasure system.

The general form of the code word generator g(X) is:

$$g(X) = \prod_{i=FCR}^{2t+FCR-1} (X+(\alpha^G)^i)$$

The roots of the code word generator g(X) are consecutive powers of any primitive element α^G of $GF(2^m)$ which can be different than the primitive element $\alpha(X)=\alpha$ used in generating the field with the field generator F(X). The first consecutive root (FCR) is an integer. A code word C(X) is comprised of a message word M(X) annexed with a parity-check word CK(X). If the message word code symbols M_i (of the form α^j) are unaltered and are appropriately placed inside a C(X), then the C(X) is said to be systematic. To be non-systematic, a code word is often of the form $C(X)_{non\text{-}systematic} = M(X)g(X)$. To be systematic, a code word is often of the form $C(X)_{systematic} = X^{n-k}M(X) + CK(X) = (X^{n-k}M(X)) + (X^{n-k}M(X)) \bmod g(X)$. We transmit or record code words.

We receive or play back received words. We know that a received word R(X) may not be a C(X), i.e., we know errors within R(X) are possible. We need to determine which symbols, if any, within R(X) are in error, i.e., we need to determine the error-locations x_i of the form X^j (or the error-location numbers z_i of the form α^j). But wait a minute. RS codes have code symbols from $GF(q)=GF(P^m)$ not GF(P); RS codes are q-ary BCH codes and are not P-ary [e.g., 2-ary (or simply binary)] BCH codes. Therefore, we also need to determine the error values y_i of the form α^j.

We know something about the coding system; we know g(X), m, n, and R(X). We assume that the number of error symbols T is less than or equal to the error correction capability t. The purpose of error correction decoding is to find and correct the errors. Assume an error-locator polynomial $\sigma(X)$ as a function of the error-location numbers z_i; $\sigma(X)$ is a function of the error-locations x_i because the x_i are a function of the z_i and the code word generator's primitive element α^G. How do we get $\sigma(X)$ from R(X)? We know if α^G is the code word generator's primitive element and that α^G may not necessarily be the special case of $\alpha^G=\alpha(X)=\alpha$, then we should denote $S_i = R(\alpha^I) = s(\alpha^I)$ for $\alpha^I = (\alpha^j)^i = \alpha^{ji}$, for i=FCR,FCR+1,...,2t+FCR-1,

and for $s(X) = R(X) \bmod g(X)$ or $s(X)$ from $s = Rh^T$. If FCR just so happens to be 1 and α^G just so happens to be $\alpha^1=\alpha$, then $S_i = R(\alpha^i) = s(\alpha^i)$ for $i=1,2,\ldots,2t$ and the x_i are simply the z_i, but in the X^{some_power} form as opposed to the α^{some_power} form. IN SUMMARY, THE FIRST DECODING STEP IS TO CALCULATE THE 2t SYNDROME COMPONENTS.

We also know that $S_i = R(\alpha^i) = C(\alpha^i) + E(\alpha^i) = E(\alpha^i)$; the syndrome components S_i are a function of the error pattern. Therefore, the next step in determining how $\sigma(X)$ is calculated from $R(X)$ is to link the syndrome components S_i to the error-locator polynomial $\sigma(X)$.

Assume an error-location polynomial $\sigma(X)$ as a function of the error-location numbers z_i which are in turn a function of the error-locations x_i:

$$\sigma(X) = (1+z_1X)(1+z_2X)\ldots(1+z_TX) = 1 + \sigma_1X + \ldots + \sigma_TX^T$$

Then the reciprocal of $\sigma(X)$ is:

$$\sigma_r(X) = (X+z_1)(X+z_2)\ldots(X+z_T) = X^T + \sigma_1X^{T-1} + \ldots + \sigma_{T-1}X + \sigma_T$$

Therefore:

$$X^T + \sigma_1X^{T-1} + \ldots + \sigma_{T-1}X + \sigma_T = (X+z_1)(X+z_2)\ldots(X+z_T)$$

Notice that $\sigma_r(X)=0$ for $X=z_1,z_2,\ldots,z_T$.

We need $y_jz_j^i$ on the left side of the previous equation. Why or how was $y_jz_j^i$ chosen? Well, we know $S_i=E(\alpha^i)$ for $i=FCR,FCR+1,\ldots,2t+FCR-1$ and for α^G being a primitive element of the GF(16). If $T \le t$ error symbols have occurred, we know that the actual error pattern $E(X) = y_1X^{j1} + y_2X^{j2} + \ldots + y_TX^{jT}$ for j_K, $K=1,2,\ldots T$. Therefore, since $S_i=E(\alpha^i)$ for $i=FCR,FCR+1,\ldots,2t+FCR-1$, we obtain $S_i = y_1z_1^i + y_2z_2^i + \ldots + y_Tz_T^i$ for $i=FCR,FCR+1,\ldots,2t+FCR-1$. The error-location numbers z_i are of the form $z_i = (\alpha^G)^{jK}$ [where j_K is from E(X)]; if α^G just so happened to be $\alpha^1=\alpha$, then $z_i = \alpha^{jK}$ [where j_K is from E(X)] and $x_i = X^{jK}$ [where j_K is from E(X)]. $S_i = y_1z_1^i + y_2z_2^i + \ldots + y_Tz_T^i$ for $i=FCR,FCR+1,\ldots,2t+FCR-1$ ARE KNOWN AS THE WEIGHTED POWER-SUM SYMMETRIC FUNCTIONS. Since the S_i are of the NON-LINEAR form $y_1z_1^i + y_2z_2^i + \ldots + y_Tz_T^i$, we need the NON-LINEAR form $y_jz_j^i$.

Getting back to finding the link between $\sigma(X)$ and the syndrome, multiply the previous $\sigma(X)$ equation by $y_jz_j^i$ on both sides. The result is the following equivalent equation:

$$y_jz_j^i(X^T + \sigma_1X^{T-1} + \ldots + \sigma_{T-1}X + \sigma_T) = y_jz_j^i((X+z_1)(X+z_2)\ldots(X+z_T))$$

Now substitute $X=z_j$ into the previous equation and remember that $\sigma_r(X)=0$ for $X = z_1, z_2, \ldots, z_{T-1}$, or z_T.

$$y_j z_j^i (z_j^T + \sigma_1 z_j^{T-1} + \ldots + \sigma_{T-1} z_j + \sigma_T) = 0 \qquad \text{for } j=1,2,\ldots,T$$

Simplify the previous equation for $j=1,2,\ldots,T$ [because $\sigma_r(z_j)=0$ for $j=1,2,\ldots,T$] to result in:

$$y_j z_j^{i+T} + y_j z_j^{i+T-1}\sigma_1 + \ldots + y_j z_j^{i+1}\sigma_{T-1} + y_j z_j^i \sigma_T = 0$$

Rearrange the terms of the previous NON-LINEAR equation to obtain the following:

$$y_j z_j^i \sigma_T + y_j z_j^{i+1}\sigma_{T-1} + \ldots + y_j z_j^{i+T-1}\sigma_1 + y_j z_j^{i+T} = 0$$

The previous equation is true for $j=1,2,\ldots,T-1$, or T. Since all of the 2t syndrome components S_i was shown to be of the form $S_i = y_1 z_1^i + y_2 z_2^i + \ldots + y_T z_T^i$ for $i=FCR,FCR+1,\ldots,2t+FCR-1$, then the following LINEAR equation for $i=FCR,FCR+1,\ldots,T+FCR-1$ results:

$$S_i\sigma_T + S_{i+1}\sigma_{T-1} + \ldots + S_{i+T-1}\sigma_1 + S_{i+T} = 0$$

The number of independent LINEAR equations for the previous equation is T and the number of unknowns is also T; therefore, the σ_i can be determined from the previous equation.

When T<t errors occur, we obtain additional equations used to solve for fewer unknowns. This is because we have 2t syndrome components (not 2T syndrome components) available. Usually the link is expressed as: $S_i\sigma_T + S_{i+1}\sigma_{T-1} + \ldots + S_{i+T-1}\sigma_1 + S_{i+T} = 0$ for $i=FCR,FCR+1,\ldots,2t-T+FCR-1$. The link can also be synonymously expressed as the following set of equations:

$$\begin{aligned}
&S_{FCR}\sigma_T + S_{FCR+1}\sigma_{T-1} + \ldots + S_{T+FCR-1}\sigma_1 + S_{T+FCR} = 0\\
&S_{FCR+1}\sigma_T + S_{FCR+2}\sigma_{T-1} + \ldots + S_{T+FCR}\sigma_1 + S_{T+FCR+1} = 0\\
&\text{etc.}\\
&S_{2t-T+FCR-1}\sigma_T + S_{2t-T+FCR}\sigma_{T-1} + \ldots + S_{2t+FCR-2}\sigma_1 + S_{2t+FCR-1} = 0
\end{aligned}$$

Sometimes the link is also expressed as $S_i = S_{i-T}\sigma_T + S_{i-T+1}\sigma_{T-1} + \ldots + S_{i-1}\sigma_1$ for $i = T+FCR,T+FCR+1,\ldots,2t+FCR-1$.

$S_i\sigma_T + S_{i+1}\sigma_{T-1} + \ldots + S_{i+T-1}\sigma_1 + S_{i+T} = 0$ for $i=FCR,FCR+1,\ldots,2t-T+FCR-1$ IS THE LINK THAT WE ARE SEARCHING FOR; this links the known $S_i = S_1, S_2, \ldots, S_{2t}$ to the unknown $\sigma(X) = 1 + \sigma_1 X + \ldots + \sigma_T X^T$.

We know $T \leq t$, but we do not yet know the actual value of T; there may be many possible solutions of the previous set of LINEAR equations for $T \leq t$. Using maximum likelihood decoding, we will

choose the value for T to be the least value out of its many possible values; e.g., if T=t or T=t-1 would be able to solve $S_i\sigma_T + S_{i+1}\sigma_{T-1} + \ldots + S_{i+T-1}\sigma_1 + S_{i+T} = 0$ for i=FCR,FCR+1, ...,T+FCR-1 and T=1,2,...,t-2 would not be able to solve it, then we would simply say that T=t-1 errors has occurred. IN SUMMARY, THE SECOND DECODING STEP IS TO CALCULATE THE ERROR-LOCATOR POLYNOMIAL FROM THE 2t SYNDROME COMPONENTS.

The next step in the decoding process is to correct the errors in the received word R(X). This final decoding step is performed by calculating the error-locations x_i, the error values y_i, and finally correcting any correctable error, if any, in R(X).

The inverse of the roots of $\sigma(X)$ [or simply the roots of $\sigma_r(X)$] are the error-location numbers z_i! The error-locations x_i are related to the-error-location numbers z_i and are in the X^j form and not the α^j form. $x_i = X\text{^}[(\log_\alpha z_i)/G]$, e.g., if GF(16), $z_1=\alpha^6$, and $\alpha^G=\alpha^2$, then $x_1 = X\text{^}[(\log_\alpha \alpha^6)/2] = X\text{^}(6/2) = X\text{^}(3) = X^3$. If α^G just so happened to be $\alpha^1=\alpha$, then $x_i = X\text{^}(\log_\alpha z_i)$, e.g., if GF(16) and $z_1=\alpha^3$, then $x_1 = X\text{^}(\log_\alpha \alpha^3) = X\text{^}(3) = X^3$.

After enough error-location numbers z_i (also denoted as error-location numbers z_l) have been calculated, then we can start calculating the error values y_i of the form α^j. We know $S_i=S_1,S_2,\ldots,S_{2t}$ and we know $z_i=z_1,z_2,\ldots,z_T$ and we know $S_i = z_1^i y_1 + z_2^i y_2 + \ldots + z_T^i y_T$ for i=FCR,FCR+1,...,2t+FCR-1. Since $T \leq t$, we have enough LINEAR equations to solve for the y_i.

Therefore, since we found the x_i and the y_i, the decoded error E(X)' is of the form:

$$E(X)' = y_1x_1 + y_2x_2 + \ldots + y_Tx_T$$

Therefore, the decoded code word C(X)' is:

$$C(X)' = R(X) - E(X)' = R(X) + E(X)'$$

IN SUMMARY, THE THIRD AND FINAL DECODING STEP IS TO CORRECT THE ERRORS IN THE RECEIVED WORD.

In summary, Reed-Solomon coding is

I. S_i from R(X) or from s(X) [s(X) from R(X) and g(X)]

II. σ_i from S_i

III. C(X)' from $R(X) + E(X)' = R(X) + y_1x_1 + y_2x_2 + \ldots + y_Tx_T$

a. x_i from z_i (z_i from σ_i)

b. y_i from S_i and z_i.

REFERENCES

1. Advanced Hardware Architectures (AHA), Inc., "AHA4010 High Speed Reed Solomon Encoder/Decoder T=1-10," Idaho : AHA, September 1988.

2. E.R. Berlekamp, Algebraic Coding Theory, California : Aegean Park Press, 1984.

3. E.R. Berlekamp, "Bit-Serial Reed-Solomon Encoders," IEEE Transactions on Information Theory, Volume IT-28, Number 6, pages 869-874, November 1982.

4. Cyclotomics (a Kodak company) for the Defense Advanced Research Projects Agency (DARPA), "Hypersystolic Reed-Solomon Decoder Final Report," Volume I, California : Cyclotomics, March 1988.

5. S. Lin, An Introduction to Error-Correcting Codes, New Jersey : Prentice-Hall, 1970.

6. S. Lin and D.J. Costello, Error Control Coding: Fundamentals and Applications, New Jersey : Prentice-Hall, 1983.

7. Midwest Research Institute (MRI) for The National Academy of Public Administration, Economic Impact and Technological Progress of NASA Research and Development Expenditures: Volume III, Technology Case Studies: Digital Communications, Civil Aeronautics Performance and Efficiency, and Future Technology Areas, pages I-1 to D-36, Missouri : MRI, September 1988.

8. H. Taub and D.L. Schilling, Principles of Communication Systems, New York : McGraw-Hill, 1986.

9. E.J. Weldon, partial handouts from the seminar "Error-Correcting Codes & Reed-Solomon Codes," pre-1987.

10. J.S.L. Wong, T.K. Truong, B. Benjauthirt, B.D.L. Mulhall, and I.S. Reed, "Review of Finite Fields: Applications to Discrete Fourier Transforms and Reed-Solomon Coding," Jet Propulsion Laboratory (JPL) publication 77-23, July 1977.

11. R.E. Ziemer and W.H. Tranter, Principles of Communications: Systems, Modulation, and Noise, Massachusetts : Houghton Mifflin, 1985.

ADDITIONAL RECOMMENDED READING

1. T.C. Bartee, Data Communications, Networks, and Systems, Chapter 10 by J.P. Odenwalder, Indiana : Howard W. Sams & Company, 1985.

2. E.R. Berlekamp, Algebraic Coding Theory, New York : McGraw-Hill, 1968.

3. E.R. Berlekamp, "The Technology of Error-Correcting Codes," Proceedings of the IEEE, Volume 68, Number 5, pages 564-593, May 1980.

4. E.R. Berlekamp, R.E. Peile, and S.P. Pope, "The Application of Error Control to Communications," IEEE Communications Magazine, Volume 25, Number 4, pages 44-57, April 1987.

5. G.C. Clark and J.B. Cain, Error-Correction Coding for Digital Communications, New York : Plenum Press, 1981.

6. N. Glover and T. Dudley, Practical Error Correction Design for Engineers, Colorado : Data Systems Technology (DST), 1988.

7. A.M. Michelson and A.H. Levesque, Error-Control Techniques for Digital Communication, New York : John Wiley & Sons, 1985.

8. F.J. McWilliams and N.J.A. Sloane, The Theory of Error-Correcting Codes, Amersterdam : North Holland, 1977.

9. W.W. Petterson, Error-Correcting Codes, New York : The MIT Press, 1961.

10. W.W. Petterson and E.J. Weldon, Error-Correcting Codes, New York : The MIT Press, 1972.

11. M.K. Simon, J.K. Omura, R.A. Scholtz, B.K. Levitt, Spread Spectrum Communications, Volumes I,II, and III, Maryland : Computer Science Press, 1985.