Technical Report

Investigation of the Use of Erasures in a Concatenated Coding Scheme

Submitted to:

NASA Lewis Research Center
21000 Brookpark Road
Cleveland, Ohio 44135

Submitted by:
Dr. S. C. Kwatra, Principal Investigator
Philip J. Marriott, Graduate Research Assistant

Department of Electrical Engineering
College of Engineering
University of Toledo
Toledo, Ohio 43606

Report No. DTVI-53
June 1997

Technical Report

Investigation of the Use of Erasures in a Concatenated Coding Scheme

Submitted to:

NASA Lewis Research Center
21000 Brookpark Road
Cleveland, Ohio 44135

Submitted by:
Dr. S. C. Kwatra, Principal Investigator
Philip J. Marriott, Graduate Research Assistant

Department of Electrical Engineering
College of Engineering
University of Toledo
Toledo, Ohio 43606

Report No. DTVI-53
June 1997

This report contains part of the work performed under NASA grant NAG3-1718 during the period September 1994 to June 1997. The research was performed as part of the Master's thesis requirement of Mr. Philip J. Marriott.

S. C. Kwatra

Principal Investigator

An Abstract of

Investigation of the use of erasures in a concatenated coding scheme

by

Philip J. Marriott

Submitted in partial fulfillment of the requirements for the Master of Science degree in

Electrical Engineering

University of Toledo

June 1997

A new method for declaring erasures in a concatenated coding scheme is investigated. This method is used with the rate 1/2 K = 7 convolutional code and the (255, 223) Reed Solomon code. Errors and erasures Reed Solomon decoding is used. The erasure method proposed use a soft output Viterbi algorithm and information provided by decoded Reed Solomon codewords in a deinterleaving frame. The results show that a gain of 0.3 dB is possible using a minimum amount of decoding trials.

ACKNOWLEDGMENTS

I would like to express my appreciation and indebtedness to my advisor Dr. Subhash Kwatra, for his guidance, support, and extreme patience throughout my masters program and research. I would also like to thank Dr. Junghwan Kim and Mr. R. E. Jones of NASA Lewis Research Center for serving on my committee. Pat Konwinski was also very helpful during the time I spent at this University. My thanks also go out to the colleagues in the Communications Lab, especially Tingfang Ji, Ping An, and Superna Metha for the many helpful suggestions while conducting my research.

Finally, I would like to thank my family and friends for their emotional support and constant encouragement. This research could not have been completed without it.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Concatenated coding systems are often used for forward error correction to obtain large coding gains when transmitting information over unreliable channels. One of the most popular concatenated coding systems is illustrated in Figure 1.1. This concatenated system uses a convolutional code with Viterbi decoding for the inner code, and a Reed Solomon code as the outer code. This is effective for a number of reasons. Convolutional codes provide sufficient random error correction, but tend to generate burst errors for low signal to noise ratio at the decoder output. Reed Solomon (RS) codes have significant burst error correcting capacity, but do not handle random errors very well. In this concatenated system, the inner convolutional code is used to correct the random errors, and although the Viterbi decoder will produce short burst errors at its output, the outer Reed Solomon code will be able to correct these bursts. The effects of these burst errors can further be reduced by using an interleaver between the inner and outer decoders. In addition, the Viterbi decoder can further improve the performance by accepting soft decisions from the receiver.

The use of erasures is one way to increase the performance of Reed Solomon codes. Erasure decoding can be thought of as the simplest form of soft decision. An erasure indicates the reception of a signal whose corresponding symbol value is in doubt.

In some cases it is better to erase the symbol than to force a decision that may be incorrect. Erasing a position gives information to the decoder as to the location of a possible error. A block code with minimum distance d_{min} can correct v errors and ρ erasures as long as the inequality $2 \cdot v + \rho \leq d_{min}$ is satisfied. Therefore, it is possible for a t-error correcting RS code to correct more than t errors if errors are transformed into erasures.

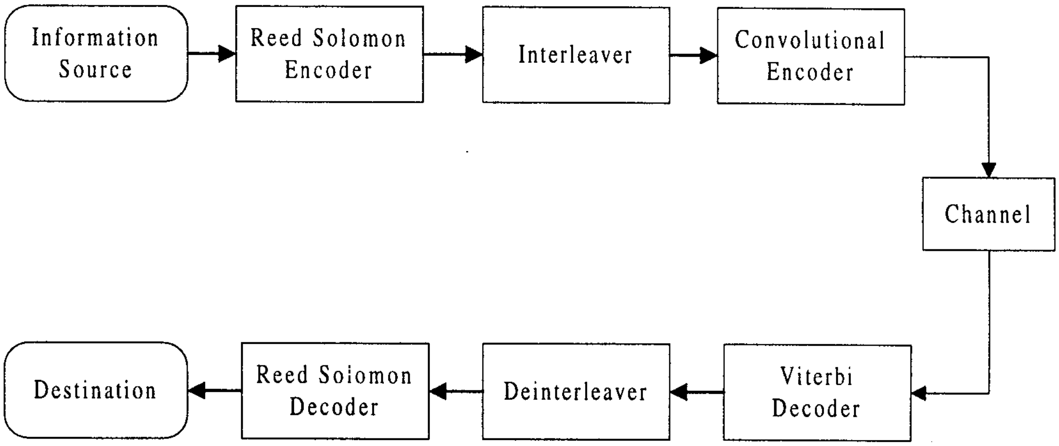


Figure 1.1 A concatenated coding system using inner convolutional code with Viterbi decoding and an outer Reed Solomon code

In the concatenated system in Figure 1.1, the Viterbi decoder produces hard outputs for input to the Reed Solomon decoder. The full capability of the concatenated system is not fully realized because no reliability information is exchanged between the inner and outer decoders. If the Viterbi decoder could be modified to generate reliability information about its output, this information could be used to declare erasures at the input to the Reed Solomon decoder, thus improving the performance. One method that can be used to accomplish this is the Soft Output Viterbi Algorithm (SOVA). The method proposed by Hagenauer and Hoeher [4] uses information provided by the path metrics in the Viterbi decoder to determine a reliability value associated with each outgoing bit.

One application where this gain could be potentially useful is in NASA deep space missions. The transmission of data over large distances, combined with limited transmission power, results in low signal to noise ratio at the receiving end. This, coupled with the fact that the data being transmitted is in the form of compressed images where the required probability of error is 10^{-5}, leads to the need for a powerful coding system [16]. The NASA standard for deep space communications is the (255, 223) 16 error correcting RS code as the outer code, and the rate 1/2 convolutional code with constraint length K = 7. Interleaver depths of I = 2 to 8 have been used. The use of a SOVA and an errors and erasures RS decoder can provide additional gains with no need to modify the transmitting end. This enables erasure decoding to be used in existing missions. This is particularly helpful for missions where unforeseen problems occur. The Galelaio mission where the main antenna failed is one such instance. Every tenth of a decibel gain that can be obtained in this instance is extremely helpful [15].

One method used to improve the NASA standard for deep space communications through the use of erasures has been investigated by Paaske [7]. This method uses the deinterleaver to provide information concerning the probable locations of errors in non-decoded Reed Solomon codewords in an interleaving frame. In an deinterleaving frame there are I Reed Solomon codewords, where I is the interleaving depth. If after attempting to decode the frame, some of the RS codewords fail to decode, redecoding is used. Erasures are declared using information provided by the error positions in the successfully decoded RS codewords. Because the Viterbi decoder produces burst of errors at it's output, and the data is fed into the deinterleaver by row and output by column to the Reed Solomon decoder, the bursts occur at the same symbols in neighboring Reed Solomon words in the deinterleaver frame. If some but not all of the Reed Solomon words in the deinterleaving frame have been successfully decoded, the positions of the errors in the decoded words are known. The knowledge of the error positions can be used to declare erasures in the same positions in neighboring, yet to be decoded Reed Solomon codewords.

1.1 Proposed research

The purpose of this report is to investigate the performance of the use of a Soft Output Viterbi Algorithm used in a concatenated coding scheme with an errors and erasures RS decoder. The reliability information provided by the SOVA will be converted into Reed Solomon symbol erasures for the RS decoder. A table of least reliable symbols will be compiled for each RS codeword, and systematically erased. In addition, another method loosely based upon Paaske's method will be investigated. This method combines the SOVA output with a deinterleaver. The table of least reliable symbols can be modified using additional information provided by the deinterleaver. If after the first decoding of a deinterleaving frame, there are less than I successful decoding of RS codewords, redecoding is attempted. It turns out that not only does the SOVA output produce burst errors, but the reliabilities for these error symbols are identical. This information is used to modify the table of smallest reliabilities. The performance of these codes will be obtained through the use of a computer simulation written in C computer language. The convolutional code developed for use in this simulation is capable of handling any code rate and constraint length. The Reed Solomon code, likewise, can handle any symbol size and number of symbol errors corrected. The Reed Solomon decoder is an errors and erasures decoder. Although the codes developed are capable of handling any size code, the NASA standard coding system will be investigated with various interleaving depths. The simulation will be performed over a AWGN channel using BPSK modulation and Raised Cosine FIR filters. The coding systems will also be simulated over an ideal BPSK channel.

The structure of this report is as follows. Chapter 2 contains all of the background information. Chapter 3 will contain the details of the computer simulation. Chapter 4 will contain the strategy for declaring RS symbol erasures from the reliability information generated by the SOVA, and the strategy for using the SOVA with the deinterleaver for redecoding. Chapter 4 will also present the results of the simulation for both methods investigated. Chapter 5 will contain conclusions, and ideas for possible future research. The simulation flow charts are found in Appendix A and the C language source code used to perform the simulations can be found in Appendix B.

Chapter 2

Background

Before discussing the two methods for erasure declaration presented in this report, it is helpful to become familiar with some of the basic concepts of error control codes. This chapter will contain all of the background necessary to understand the various elements used in the concatenated system. Encoding and decoding of Reed Solomon and convolutional codes will be reviewed. In addition, the method used for errors and erasures decoding in the Reed Solomon decoder will be discussed, as well as the method used for generating the soft outputs in the Viterbi decoder. Block interleaving will be briefly discussed, in addition to the redecoding method proposed by Paaske.

2.1 Reed Solomon codes

Bose-Chadhuri-Hocquenghem (BCH) codes are a powerful class of cyclic codes which outperform all other block codes with the same block length and code length [9]. These codes are a generalization of Hamming codes to allow multiple error correction. Reed Solomon (RS) codes are special subclass of BCH codes which utilize non-binary symbols. The non-binary symbols used in RS codes are formed using finite field arithmetic. Finite fields are sometimes called Galois fields and are denoted by GF(p), where p is the number of elements in the field, and is a prime number.

2.1.1 Galois fields

A field is a set of elements in which we can do addition, subtraction, multiplication, and division without leaving the set. Subtraction and division are defined by the additive inverse and the multiplicative inverse. Addition and multiplication must also satisfy the commutative, associative, and distributive laws. A field with a finite number of elements is called a finite field. For example, GF(7) = {0, 1, 2, 3, 4, 5, 6} is a field under modulo 7 addition and multiplication. Reed Solomon codes are codes with symbols from the field $GF(2^m)$, where $GF(2^m) = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{2^m-2}\}$. The field $GF(2^m)$ is an extension of the ground field GF(2), and the elements in this field can be represented by an ordered sequence of m components, $(a_0, a_1, a_2, \ldots, a_{m-1})$, or an m-tuple. Each of the components are from the ground field GF(2). The 2^m elements of the field $GF(2^m)$ are defined by an irreducible polynomial, or a primitive polynomial P(x). Each element will satisfy the condition $P(\alpha) = 0$. The primitive polynomials that define the elements for m = 3 to 9 are as in Table 2.1. For example, the elements in $GF(2^3)$ are defined using the primitive polynomial $1 + x + x^3$. The elements either are the zero element '0', the identity element '1', or some power of the base element α. The element α^3 is derived from the primitive polynomial and the relationship $P(\alpha) = 0$.

$$P(\alpha) = \alpha^3 + \alpha + 1 = 0$$

$$\text{or} \quad \alpha^3 = \alpha + 1.$$

All other elements are simply generated by multiplication by α. The table repeats after α^{m-2} (i.e. $\alpha^6 \cdot \alpha = 1$, $\alpha^6 \cdot \alpha^2 = \alpha$ etc.). The elements for $GF(2^3)$ are as follows.

$$
\begin{aligned}
0 &= 0 \\
1 &= 1 \\
\alpha &= 1\cdot\alpha = \alpha \\
\alpha^2 &= \alpha\cdot\alpha = \alpha^2 \\
\alpha^3 &= 1 + \alpha \\
\alpha^4 &= \alpha\cdot\alpha^3 = \alpha\cdot(\alpha+1) = \alpha+\alpha^2 \\
\alpha^5 &= \alpha\cdot\alpha^4 = \alpha\cdot(\alpha+\alpha^2) = 1+\alpha+\alpha^2 \\
\alpha^6 &= \alpha\cdot\alpha^5 = \alpha\cdot(1+\alpha+\alpha^2) = 1+\alpha^2 \\
&\vdots
\end{aligned}
$$

Table 2.1: List of primitive polynomials for m = 3 to 9

m	P(X)
3	$1 + x + x^3$
4	$1 + x + x^4$
5	$1 + x^2 + x^5$
6	$1 + x + x^6$
7	$1 + x^3 + x^7$
8	$1 + x^2 + x^3 + x^4 + x^8$
9	$1 + x^4 + x^9$

It is useful to represent these elements in a number of ways. The polynomial representation is given by $a_0 + a_1\alpha + a_2\alpha^2 + \ldots + a_{m-1}\alpha^{m-1}$ and the m-tuple representation is given by $(a_0, a_1, a_2, \ldots, a_{m-1})$. The elements and the various representations for GF(8) are given in Table 2.2.

Table 2.2 Three representations for the elements of GF(8) generated by $1 + x + x^3$

Power Representation	Polynomial Representation	3-tuple Representation
0	0	(0 0 0)
1	1	(1 0 0)
α	α	(0 1 0)
α^2	α^2	(0 0 1)
α^3	$1 + \alpha$	(1 1 0)
α^4	$\alpha + \alpha^2$	(0 1 1)
α^5	$1 + \alpha + \alpha^2$	(1 1 1)
α^6	$1 + \alpha^2$	(1 0 1)

Multiplication and addition follow the rules of finite field algebra. Multiplication of two elements is accomplished by adding the powers of the two elements modulo 2^m-1. For example, in GF(8), $\alpha^4 \cdot \alpha^5 = \alpha^{(4+5) \bmod 7} = \alpha^2$. For addition of two elements in a field, it is useful to use the m-tuple representation of an element. Consider a = $(a_0, a_1, a_2, \ldots, a_{m-1})$ and b = $(b_0, b_1, b_2, \ldots, b_{m-1})$. The addition of a and b is simply the addition of each component in the m-tuple representation, namely$(a_0 + b_0, a_1 + b_1, a_2 + b_2, \ldots, a_{m-1} + b_{m-1})$. Because each component of the m-tuple is from GF(2), binary addition is used.

2.1.2 Generating Reed Solomon codes

A t-error correcting Reed Solomon code with symbols from GF(2^m) has the following parameters:

Block Length	$n = 2^m - 1$
Number of information symbols	$k = n - 2t$
Minimum Distance	$d_{min} = 2t + 1$

The generator polynomial of a t-error correcting Reed Solomon code is:

$$g(x) = (x + \alpha) \cdot (x + \alpha^2) \cdots (x + \alpha^{2t})$$

where g(x) has all of its roots and coefficients from GF(2^m). The code generated from g(x) is a (n, n - 2t) cyclic code. The code words are generated by:

$$\bar{c} = \bar{u} \cdot G$$

Where G is the generator matrix in systematic form. Let us design a t = 2 error correcting Reed Solomon code using symbols from GF(2^3) = GF(8). We know that:

Block length	$n = 2^3 - 1$	$n = 7$
Information symbols	$k = n - 2t$	$k = 3$

The generator polynomial for this (7, 3) Reed Solomon code is given by:

$$g(x) = (x + \alpha) \cdot (x + \alpha^2) \cdot (x + \alpha^3) \cdot (x + \alpha^4)$$

or

$$g(x) = \alpha^3 + \alpha \cdot x + x^2 + \alpha^3 \cdot x^3 + x^4$$
$$= [\ \alpha^3\ \alpha\ 1\ \ \alpha^3\ 1]$$

The generator matrix in non-systematic form is:

$$G(X) = \begin{bmatrix} \alpha^3 & \alpha & 1 & \alpha^3 & 1 & 0 & 0 \\ 0 & \alpha^3 & \alpha & 1 & \alpha^3 & 1 & 0 \\ 0 & 0 & \alpha^3 & \alpha & 1 & \alpha^3 & 1 \end{bmatrix}$$

To get the matrix into systematic form, we must convert the last three columns into an identity matrix. This is accomplished by row operations. The result after doing so is:

$$G(X) = \begin{bmatrix} \alpha^3 & \alpha & 1 & \alpha^3 & 1 & 0 & 0 \\ \alpha^6 & \alpha^6 & 1 & \alpha^2 & 0 & 1 & 0 \\ \alpha^5 & \alpha^4 & 1 & \alpha^4 & 0 & 0 & 1 \end{bmatrix}$$

The information bits to be transmitted are u = [010 011 110]. From Table 2.2 we know that these bits correspond to the symbols $u = [\alpha\ \alpha^5\ \alpha^3]$ in GF(8). Using $\bar{c} = \bar{u} \cdot \bar{G}$, we obtain the code vector

$$c = [\alpha\ \alpha^5\ \alpha^3] \cdot \begin{bmatrix} \alpha^3 & \alpha & 1 & \alpha^3 & 1 & 0 & 0 \\ \alpha^6 & \alpha^6 & 1 & \alpha^2 & 0 & 1 & 0 \\ \alpha^5 & \alpha^4 & 1 & \alpha^4 & 0 & 0 & 1 \end{bmatrix}$$

$$c = [\ \alpha\ \ \alpha^3\ \ \alpha^4\ \ \alpha^4\ \ \alpha\ \ \alpha^5\ \ \alpha^3\ ]$$

or

$$c = [010\ \ 110\ \ 011\ \ 011\ \ 010\ \ 011\ \ 110]$$

The encoding of Reed Solomon codes can be also accomplished using a shift register circuit. For a t-error correcting RS code, the generator polynomial is given by:

$$g(x) = (x + \alpha) \cdot (x + \alpha^2) \cdots (x + \alpha^{2t})$$

$$= g_0 + g_1 x + g_2 x^2 + \ldots + g_{2t-1} x^{2t-1} + x^{2t}$$

where g(x) has all of its roots and has coefficients from GF(2^m). The generator polynomial g(x) has been chosen so that it and codewords generated by it have zeros for $2 \cdot t$ consecutive powers of α

$$g(\alpha^j) = 0 \quad \text{for } j = 1, 2, \ldots 2 \cdot t$$

The code generated from g(x) is a (n, n - 2t) cyclic code. The encoding of a non-binary cyclic code is similar to the encoding of a binary cyclic code. Let

$$u(x) = u_0 + u_1 x + u_2 x^2 + \cdots + u_{2t-1} x^{2t-1}$$

be the message to be encoded. In systematic form, the 2t parity check symbols are the coefficients of the remainder $b(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{2t-1} x^{2t-1}$ which is obtained by dividing the message polynomial u(x) by the generator polynomial g(x). In hardware, this is accomplished by using the shift register circuit of Figure 2.1. The encoder circuit works as follows. The k information symbols are first loaded into the circuit. At the same time, the k information symbols are transferred directly to the output.

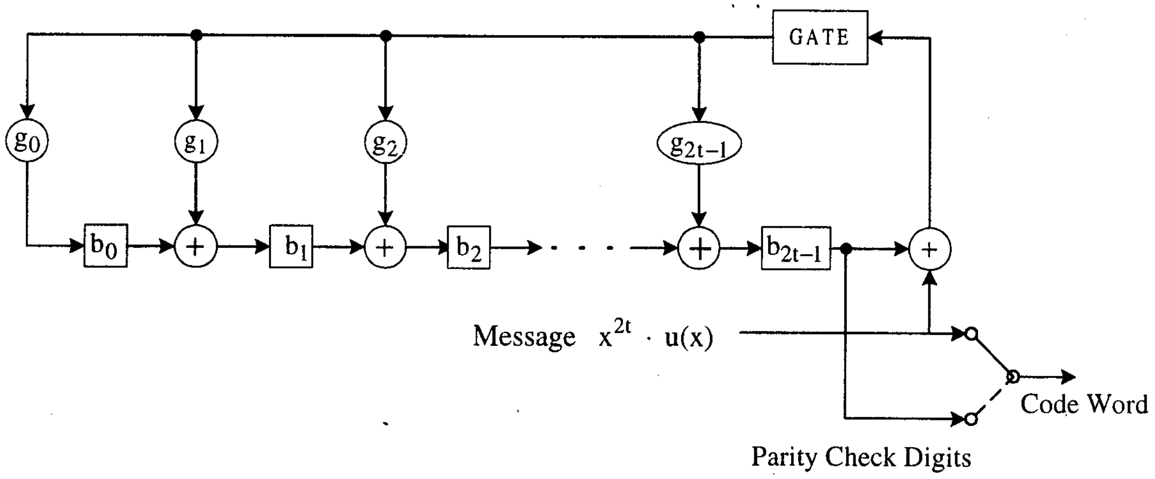


Figure 2.1 Encoding circuit for t error correcting RS code

When all of the information symbols have been read in, the 2t parity symbols are present in the 2t registers denoted $b_0, b_1, \ldots, b_{2t-1}$, and are then transferred to the output, thus completing the systematic code word. This process can best be illustrated with an example. Consider the t = 2 error correcting (7, 3) Reed Solomon code. The generator polynomial is

$$\begin{aligned} g(x) &= (x+\alpha)\cdot(x+\alpha^2)\cdot(x+\alpha^3)\cdot(x+\alpha^4) \\ &= \alpha^3 + \alpha\cdot x + x^2 + \alpha^3\cdot x^3 + x^4 \end{aligned}$$

and it's corresponding encoding circuit is given in Figure 2.2. The encoding of the information symbols $u = [\alpha\ \alpha^5\ \alpha^3\ ]$ is given in Table 2.3. The information symbols are fed directly to the output to the encoding circuit. At t = 1, the first information symbol is fed into the encoding circuit, and the register contents are modified. Information symbols are fed into the encoder until t = 3. At this time, the 4 parity symbols are present in the registers b_0, b_1, b_2, b_3, and are sent to the output of the

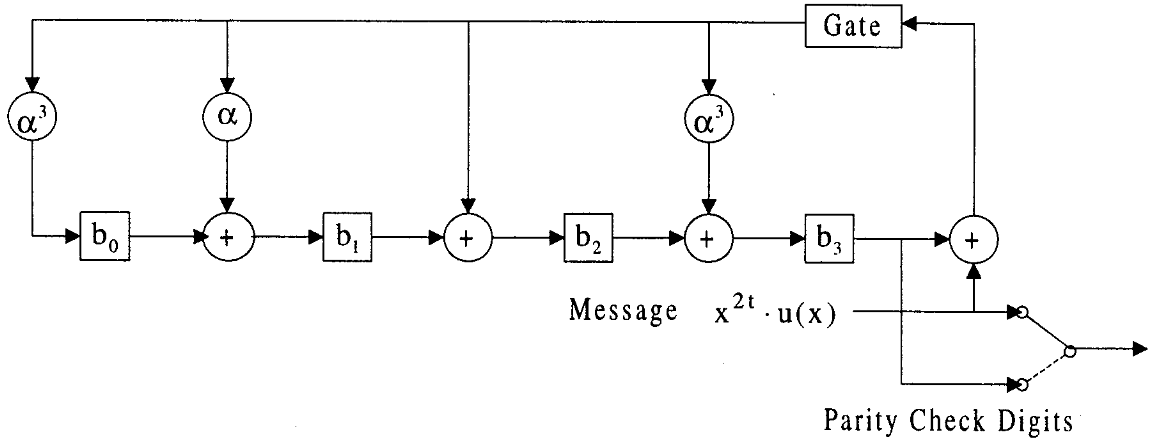


Figure 2.2 Encoding circuit for t = 2 error correcting (7, 3) RS code

encoding circuit. The encoded vector is equal to $c = [b_0\ b_1\ b_2\ b_3\ u_2\ u_1\ u_0]$ or $c = [\ \alpha \quad \alpha^3 \quad \alpha^4 \quad \alpha^4 \quad \alpha \quad \alpha^5 \quad \alpha^3\ ]$.

Table 2.3 Shift register contents for the encoding of $u = [\alpha\ \alpha^5\ \alpha^3\ ]$

t	Input Symbols	Gate	b_0	b_1	b_2	b_3
0	-	-	0	0	0	0
1	$u_0 = \alpha^3$	α^3	α^6	α^4	α^3	α^6
2	$u_1 = \alpha^5$	$\alpha^5 + \alpha^6 = \alpha$	α^4	1	α^2	α^6
3	$u_2 = \alpha$	$\alpha + \alpha^6 = \alpha^5$	α	α^3	α^4	α^4

2.1.3 Decoding of Reed Solomon Codes

Let $r(x) = r_0 + r_1x + \ldots + r_{n-1}x^{n-1}$ be a received polynomial which is equal to a codeword $c(x) = c_0 + c_1x + \ldots + c_{n-1}x^{n-1}$ corrupted by an error pattern $e(x) = e_0 + e_1x + \ldots + e_{n-1}x^{n-1}$.

$$r(x) = c(x) + e(x)$$

The syndrome of the received polynomial is obtained by evaluating r(x) at the $2 \cdot t$ zeros.

$$S_j = r(\alpha^j) = c(\alpha^j) + e(\alpha^j) \quad j = 1, 2, \ldots, 2 \cdot t$$

Any codeword c(x) will have zeros for these $2 \cdot t$ powers of α, and thus, have a syndrome equal to zero. Therefore, the syndrome of the received word is equivalent to the syndrome of the error pattern.

$$S_j = r(\alpha^j) = e(\alpha^j) = \sum_{k=0}^{n-1} e_k \cdot (\alpha^j)^k \quad j = 1, 2, \ldots, 2 \cdot t \tag{2.1}$$

If there are v errors in positions $i_1, i_2, \ldots, i_v$, (2.1) can be expressed as

$$\begin{aligned} S_j &= \sum_{l=1}^{v} e_{i_l} (\alpha^j)^{i_l} \\ &= e_{i_1}\alpha^{i_1} + e_{i_2}\alpha^{i_2} + \ldots + e_{i_v}\alpha^{i_v} \quad j = 1, 2, \ldots, 2 \cdot t \end{aligned} \tag{2.2}$$

To reduce the notational complexity of (2.2), the error locations will be defined as $X_l = \alpha^{i_l}$, and the error magnitudes as $Y_l = e_{i_l}$ where $l = 1, 2, \ldots, v$. (2.2) then becomes

$$\begin{aligned} S_1 &= Y_1X_1 + Y_2X_2 + \ldots + Y_vX_v \\ S_2 &= Y_1X_1^2 + Y_2X_2^2 + \ldots + Y_vX_v^2 \\ S_3 &= Y_1X_1^3 + Y_2X_2^3 + \ldots + Y_vX_v^3 \\ &\vdots \\ S_{2t} &= Y_1X_1^{2t} + Y_2X_2^{2t} + \ldots + Y_vX_v^{2t} \end{aligned} \tag{2.3}$$

The error locator polynomial $\Lambda(x)$ is defined as

$$\begin{aligned} \Lambda(x) &= (1 - xX_1)(1 - xX_2) \cdots (1 - xX_v) \\ &= 1 + \Lambda_1 x + \Lambda_2 x^2 + \ldots + \Lambda_{v-1}x^{v-1} + \Lambda_v x^v \end{aligned} \tag{2.4}$$

where the roots of $\Lambda(x)$ are the error locations $X_1, X_2, \ldots, X_v$. The coefficients of the error location polynomial Λ_l $l = 0, 1, \ldots, v$ are related to the error locations by the following equations

$$\begin{aligned} \Lambda_0 &= 1 \\ \Lambda_1 &= X_1 + X_2 + \ldots + X_{v-1} + X_v \\ \Lambda_2 &= X_1X_2 + X_1X_3 + \ldots + X_{v-2}X_v + X_{v-1}X_v \\ \Lambda_3 &= X_1X_2X_3 + X_1X_2X_4 + \ldots + X_{v-3}X_{v-1}X_v + X_{v-2}X_{v-1}X_v \\ &\vdots \\ \Lambda_v &= X_1X_2X_3 \cdots X_{v-1}X_v \end{aligned} \tag{2.5}$$

(2.3) and (2.5) are related by Newton's identities [11]

$$\begin{aligned}
S_1 + \Lambda_1 &= 0 \\
S_2 + \Lambda_1 S_1 + 2\Lambda_2 &= 0 \\
S_3 + \Lambda_1 S_2 + \Lambda_2 S_1 + 3\Lambda_3 &= 0 \\
&\vdots \\
S_v + \Lambda_1 S_{v-1} + \Lambda_2 S_{v-2} + \ldots + \Lambda_{v-1} S_1 + v\Lambda_v &= 0 \\
S_{v+1} + \Lambda_1 S_v + \Lambda_2 S_{v-1} + \ldots + \Lambda_v S_1 &= 0 \\
&\vdots \\
S_{2t} + \Lambda_1 S_{2t-1} + \Lambda_2 S_{2t-2} + \ldots + \Lambda_v S_{2t-v} &= 0
\end{aligned} \tag{2.6}$$

(2.6) can be solved directly to obtain the coefficients of the error locator polynomial, but such methods require a number of computations proportional to t^3 [2]. This makes a direct solution of (2.6) not practical, especially for RS codes that need to correct a large amount of errors. Berlekamp's algorithm is much more computationally efficient method of correcting RS codes. The complexity increases linearly with t, so codes correcting large numbers of errors can be implemented [11]. Berlekamp's algorithm first finds a minimum degree polynomial $\Lambda^{(1)}(x)$ whose coefficients satisfy Newton's first identity. This polynomial is tested whether the second Newton identity is also satisfied. If it does, then $\Lambda^{(2)}(x) = \Lambda^{(1)}(x)$. If not, then a correction term, or discrepancy is added to $\Lambda^{(1)}(x)$ to form $\Lambda^{(2)}(x)$ such that $\Lambda^{(2)}(x)$ satisfies the first two Newton's identities. Next $\Lambda^{(2)}(x)$ is tested whether it satisfies the third Newton's identity, etc. This process continues until $\Lambda^{(2t)}(x)$ is obtained. Then $\Lambda(x) = \Lambda^{(2t)}(x)$. If there are less than t errors, $\Lambda(x)$ produces the error pattern.

Massey's shift register based interpretation of Berlekamp's algorithm is known as the Berlekamp-Massey algorithm [2, 11]. The Newton's Identities in (2.6) can be expressed in an alternate form

$$S_j = -\sum_{i=1}^{v} \Lambda_i S_{j-i} \quad j = v+1,\ v+2,\ \ldots,\ 2v. \tag{2.7}$$

Massey [17] recognized that (2.7) can be represented physically using a linear feedback shift register (LFSR) as shown in Figure 2.3.

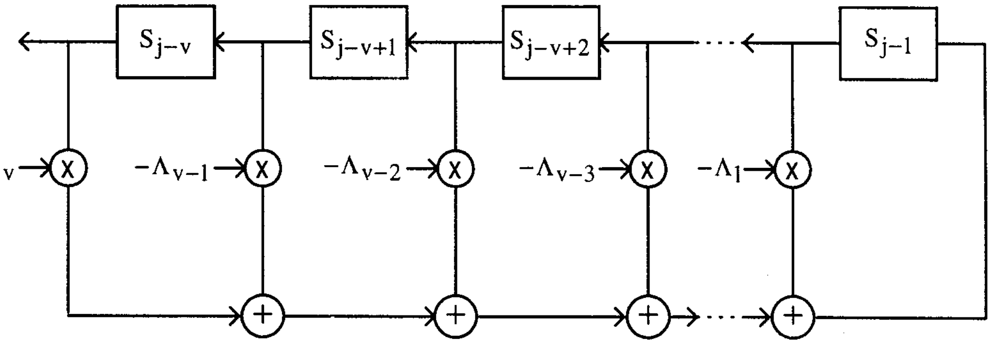


Figure 2.3 LFSR interpretation of (2.7)

The output of the LFSR will be the 2t syndromes $S_1, S_2, \ldots, S_{2t}$, and the register taps are the coefficients of the error correction polynomial $\Lambda(x)$. The LFSR can be designed to generate the known sequence of syndromes such that $\Lambda(x)$ is of the smallest degree. The procedure for finding the taps of the LFSR is similar to Berlekamp's algorithm. First a connection polynomial $T(x) = 1 + \Lambda_1 x + \ldots + \Lambda_{L-1}x^{L-1} + \Lambda_L x^L$ is formed whose coefficients are the taps of a length L LFSR. The Berlekamp-Massey algorithm first finds T(x) of length L = 1 such that the first output of the LFSR is the first syndrome

S_1. The second output of the LFSR is compared to the second syndrome, and if the two are not equal then the connection polynomial is modified using a discrepancy term. If the two are equal the taps remain the same. The third output of the LFSR is compared to the third syndrome, and if they are not equal, the taps of the connection polynomial are modified. This process continues for 2t iterations. At the end of the 2t iterations, the taps of the LFSR specify the coefficients of the error correction polynomial $\Lambda(x)$. The details of the algorithm are presented below.

2.1.3.1 Berlekamp-Massey Algorithm [2]

1. Compute the syndrome of the received codeword $S_j = r(\alpha^j)$.
2. Initialize the following variables

   Error locator polynomial $\Lambda(x) = 1$

   Index $r = 0$

   Temporary storage $B(x) = 1$

   Shift register length $L = 0$
3. Set $r = r + 1$.
4. Compute the r th discrepancy, which is the error in the next syndrome

$$\Delta_r = \sum_{j=0}^{L} \Lambda_j S_{r-j}$$

5. If $\Delta_r = 0$, set $B(x) = x \cdot B(x)$ and go to step 11.
6. Compute the new connection polynomial

$$T(x) = \Lambda(x) - \Delta_r \cdot x \cdot B(x)$$

7. If $2 \cdot L > r - 1$, set $B(x) = x \cdot B(x)$ and go to step 10.

8. Store old shift register after normalizing

$$B(x) = \Delta_r^{-1}\Lambda(x)$$

9. Update shift register length $L = r - L$.

10. Update the shift register

$$\Lambda(x) = T(x)$$

11. If $r < 2 \cdot t$, go to step 3.

12. If $\deg \Lambda(x) \neq L$, there are more than t errors. Stop.

13. Determine the roots of $\Lambda(x)$. The inverses of these roots are the error locations $X_1, X_2, \ldots, X_v$.

14. Determine the corresponding error values $Y_1, Y_2, \ldots, Y_v$.

The simplest method to find the roots of $\Lambda(x)$ in step 13 is by using a process known as a Chien Search. This is a trial and error approach which computes $\Lambda(\alpha^j)$ for j = 0, 1, .. 2^m-2. If $\Lambda(\alpha^j) = 0$, then α^j is a root of $\Lambda(x)$. The error magnitudes can be calculated by using the Forney algorithm[3, 11]. First, compute a syndrome polynomial S(x) from the $2 \cdot t$ syndromes.

$$S(x) = 1 + \sum_{j=1}^{2 \cdot t} S_j x^j \tag{2.7}$$

The error evaluator polynomial $\Omega(x)$ can be computed by the product of the syndrome polynomial S(x) and the error locator polynomial $\Lambda(x)$.

$$\Omega(x) = S(x) \cdot \Lambda(x) \bmod x^{2 \cdot t+1} \tag{2.8}$$

Next, compute the derivative of the error locator polynomial $\Lambda'(x)$.

$$\Lambda'(x) = \sum_{i=1}^{v} X_i \prod_{j \neq i} (1 - xX_j) \qquad (2.9)$$

The error magnitudes $Y_1, Y_2, \ldots, Y_v$ can then be calculated using the Forney Algorithm.

$$Y_l = \frac{X_l^{-1}\Omega(X_l^{-1})}{\Lambda'(X_l^{-1})} \quad l = 1, 2, \ldots, v \qquad (2.10)$$

For example, let $r(x) = \alpha + \alpha^2 x + \alpha^4 x^2 + \alpha^4 x^3 + \alpha^6 x^4 + \alpha^5 x^5 + \alpha^3 x^6$ be a code word corrupted by an error pattern e(x). The first step in decoding is to compute the syndrome of the received polynomial.

$S_1 = r(\alpha) = \alpha + \alpha^2\alpha + \alpha^4\alpha^2 + \alpha^4\alpha^3 + \alpha^6\alpha^4 + \alpha^5\alpha^5 + \alpha^3\alpha^6$
$S_1 = 1$

$S_2 = r(\alpha^2) = \alpha + \alpha^2\alpha^2 + \alpha^4\alpha^4 + \alpha^4\alpha^6 + \alpha^6\alpha^8 + \alpha^5\alpha^{10} + \alpha^3\alpha^{12}$
$S_2 = \alpha^2$

$S_3 = r(\alpha^3) = \alpha + \alpha^2\alpha^3 + \alpha^4\alpha^6 + \alpha^4\alpha^9 + \alpha^6\alpha^{12} + \alpha^5\alpha^{15} + \alpha^3\alpha^{18}$
$S_3 = 1$

$S_4 = r(\alpha^4) = \alpha + \alpha^2\alpha^4 + \alpha^4\alpha^8 + \alpha^4\alpha^{12} + \alpha^6\alpha^{16} + \alpha^5\alpha^{20} + \alpha^3\alpha^{24}$
$S_4 = \alpha^6$

The syndrome polynomial is

$$S(x) = 1 + \sum_{j=1}^{2\cdot t} S_j x^j = 1 + x + \alpha^2 x^2 + x^3 + \alpha^6 x^4 . \qquad (2.11)$$

Next, the Berlekamp-Massey algorithm is used to find the error locator polynomial $\Lambda(x)$. The results of the computations for each iteration of the algorithm are given in Table 2.4.

Table 2.4 Results of the computations for each iteration of the Berlekamp-Massey algorithm

r	Δ_r	T(x)	B(x)	Λ(x)	L
0	-	-	1	1	0
1	1	$1 + x$	1	$1 + x$	1
2	α^6	$1 + \alpha^2 x$	x	$1 + \alpha^2 x$	1
3	α^5	$1 + \alpha^2 x + \alpha^5 x^2$	$\alpha^2 + \alpha^4 x$	$1 + \alpha^2 x + \alpha^5 x^2$	2
4	0	$1 + \alpha^2 x + \alpha^5 x^2$	$\alpha^2 + \alpha^4 x$	$1 + \alpha^2 x + \alpha^5 x^2$	2

The error locator polynomial is found to be $\Lambda(x) = 1 + \alpha^2 x + \alpha^5 x^2$. The roots of $\Lambda(x)$ are α^6 and α^3, and the inverses of these roots give the error locations $X_1 = \alpha$ and $X_2 = \alpha^4$. The error evaluator polynomial is

$$\Omega(x) = S(x)\Lambda(x) = (1 + x + \alpha^2 x^2 + x^3 + \alpha^6 x^4)(1 + \alpha^2 x + \alpha^5 x^2)$$
$$= 1 + \alpha^6 x + \alpha^5 x^2$$

and the derivative of the error location polynomial is $\Lambda'(x) = \alpha^2$. The error values can be calculated using (2.10)

$$Y_1 = \frac{\alpha(1 + \alpha^6\alpha^6 + \alpha^5\alpha^{12})}{\alpha^2} = \frac{\alpha(1 + \alpha^5 + \alpha^3)}{\alpha^2} = \alpha^5$$

$$Y_2 = \frac{\alpha^4(1 + \alpha^6\alpha^3 + \alpha^5\alpha^6)}{\alpha^2} = \frac{\alpha^4(1 + \alpha^2 + \alpha^4)}{\alpha^2} = \alpha^5$$

The error polynomial is $e(x) = \alpha^5 x + \alpha^5 x^4$, and the corrected polynomial is

$$r(x) + e(x) = (\alpha + \alpha^2 x + \alpha^4 x^2 + \alpha^4 x^3 + \alpha^6 x^4 + \alpha^5 x^5 + \alpha^3 x^6) + (\alpha^5 x + \alpha^5 x^4)$$

$$= \alpha + \alpha^3 x + \alpha^4 x^2 + \alpha^4 x^3 + \alpha x^4 + \alpha^5 x^5 + \alpha^3 x^6$$

2.1.3.2 Errors and erasures RS decoder

In order to correct both errors and erasures, certain modifications to the Berlekamp-Massey algorithm will need to be made. Suppose that a received polynomial r(x) contains v errors in locations $i_1, i_2, \ldots, i_v$ and ρ erasures in locations $j_1, j_2, \ldots, j_\rho$. An errors and erasures Reed Solomon code can correct v errors and ρ erasures as long as $2 \cdot \rho + v \le d_{min}$ where d_{min} is the minimum distance of the code. The error locations are given by $X_k = \alpha^{i_k}$ $k = 1, 2, \ldots, v$ and the erasure locations are given by $U_l = \alpha^{j_l}$ $l = 1, 2, \ldots, \rho$. The erased positions are known at the beginning of the decoding operation, and are filled with zeros before the decoding begins.

2.1.3.3 Berlekamp-Massey algorithm for errors and erasures [2]

1. Substitute zeros into the erased positions in the received word.
2. Compute the syndrome of the received codeword $S_j = r(\alpha^j)$.
3. Initialize the following variables

   Errors and erasures locator polynomial $\Lambda(x) = 1$

   Index $r = 0$

   Temporary Storage $B(x) = 1$

Shift register length $L = 0$

4. Set $r = r + 1$.

5. If $r > \rho$ go to step 10

6. $\Lambda(x) = \Lambda(x) \cdot (1 - U_r \cdot x)$

7. $B(x) = \Lambda(x)$

8. $L = L + 1$

9. Go to step 4

10. Compute the r th discrepancy, which is the error in the next syndrome

$$\Delta_r = \sum_{j=0}^{L} \Lambda_j S_{r-j}$$

11. If $\Delta_r = 0$, set $B(x) = x \cdot B(x)$ and go to step 11.

12. Compute the new connection polynomial $T(x) = \Lambda(x) - \Delta_r \cdot x \cdot B(x)$

13. If $2 \cdot L > r + \rho - 1$, set $B(x) = x \cdot B(x)$ and go to step 16.

14. Store old shift register after normalizing

$$B(x) = \Delta_r^{-1} \Lambda(x)$$

15. Update shift register length $L = r - L + \rho$.

16. Update the shift register

$$\Lambda(x) = T(x)$$

17. If $r < 2 \cdot t$, go to step 4.

18. If $\deg \Lambda(x) \neq L$, $2\rho + v > d_{\min}$. Stop.

19. Determine the roots of $\Lambda(x)$. The inverses of these roots give the error locations $X_1, X_2, \ldots, X_v$ and the erasure locations $U_1, U_2, \ldots, U_\rho$.

20. Compute the error evaluator polynomial

$$\Omega(x) = S(x) \cdot \Lambda(x) \mod x^{2 \cdot t+1}$$

21. Use the Forney Algorithm to compute the error values

$$Y_{i_k} = \frac{X_k \Omega(X_k^{-1})}{\Lambda'(X_k^{-1})} \qquad i = 1, 2, \ldots, v \tag{2.13}$$

and the erasure values

$$Y_{j_l} = \frac{U_l \Omega(U_l^{-1})}{\Lambda'(U_l^{-1})} \qquad j = 1, 2, \ldots, \rho \tag{2.14}$$

For example, let $r(x) = \alpha + \alpha^3 x + f x^2 + \alpha^4 x^3 + \alpha^6 x^4 + f x^5 + \alpha^3 x^6$ be a code word corrupted by an unknown error pattern e(x) and a erasure pattern $f(x) = f x^2 + f x^5$ with known positions and unknown values denoted by ‘f’. The first step in decoding is to insert zeros into the erased positions and compute the syndrome of the received polynomial.

$$S_1 = r(\alpha) = \alpha + \alpha^3\alpha + \alpha^4\alpha^3 + \alpha^6\alpha^4 + \alpha^3\alpha^6$$
$$S_1 = \alpha$$

$$S_2 = r(\alpha^2) = \alpha + \alpha^3\alpha^2 + \alpha^4\alpha^6 + \alpha^6\alpha^8 + \alpha^3\alpha^{12}$$
$$S_2 = \alpha^6$$

$$S_3 = r(\alpha^3) = \alpha + \alpha^3\alpha^3 + \alpha^4\alpha^9 + \alpha^6\alpha^{12} + \alpha^3\alpha^{18}$$
$$S_3 = \alpha^6$$

$$S_4 = r(\alpha^4) = \alpha + \alpha^3\alpha^4 + \alpha^4\alpha^{12} + \alpha^6\alpha^{16} + \alpha^3\alpha^{24}$$
$$S_4 = 0$$

The syndrome polynomial is

$$S(x) = 1 + \sum_{j=1}^{2\cdot t} S_j x^j = 1 + \alpha x + \alpha^6 x^2 + \alpha^6 x^3 .$$

Next, the modified Berlekamp-Massey algorithm is used to find the error locator polynomial $\Lambda(x)$. The contents of the variables for each iteration of the algorithm are given in Table 2.5.

Table 2.5 Results of the computations for each iteration of the errors and erasures Berlekamp-Massey algorithm

r	Δ_r	T(x)	B(x)	$\Lambda(x)$	L
0	-	-	1	1	0
1	-	-	$1 + \alpha^2 x$	$1 + \alpha^2 x$	1
2	-	-	$1 + \alpha^3 x + x^2$	$1 + \alpha^3 x + x^2$	2
3	α^3	$1 + \alpha^2 x^2 + \alpha^3 x^3$	$\alpha^4 + x + \alpha^4 x^2$	$1 + \alpha^2 x^2 + \alpha^3 x^3$	3
4	α^2	$1 + \alpha^6 x + \alpha^4 x^3$	$\alpha^4 x + x^2 + \alpha^4 x^3$	$1 + \alpha^6 x + \alpha^4 x^3$	3

The error locator polynomial is found to be $\Lambda(x) = 1 + \alpha^6 x + \alpha^4 x^3$. The roots of $\Lambda(x)$ are α^2, α^3, and α^5. The inverses of these roots give the error location $X_1 = \alpha^4$ and the erasure locations $U_1 = \alpha^2$ and $U_2 = \alpha^5$, which were known at the beginning of the decoding operation. The error connection polynomial is

$$\Omega(x) = S(x)\Lambda(x) = (1 + \alpha x + \alpha^6 x^2 + \alpha^6 x^3)(1 + \alpha^6 x + \alpha^4 x^3)$$

$$= 1 + \alpha^5 x + \alpha^2 x^2 + \alpha^2 x^3$$

and the derivative of the error location polynomial is $\Lambda'(x) = \alpha^6 + \alpha^4 x^2$. The error value can be calculated using (2.13)

$$Y_1 = \frac{\alpha^4(1 + \alpha^5\alpha^3 + \alpha^2\alpha^6 + \alpha^2 x^9)}{\alpha^6 + \alpha^4\alpha^6} = \frac{\alpha^4(1 + \alpha + \alpha + \alpha^4)}{\alpha^6 + \alpha^3} = \alpha^5$$

and the erasure values by using (2.14)

$$Y_1 = \frac{\alpha^2(1 + \alpha^5\alpha^5 + \alpha^2\alpha^{10} + \alpha^2 x^{15})}{\alpha^6 + \alpha^4\alpha^{10}} = \frac{\alpha^2(1 + \alpha^3 + \alpha^5 + \alpha^3)}{\alpha^6 + 1} = \alpha^4$$

$$Y_1 = \frac{\alpha^5(1 + \alpha^5\alpha^2 + \alpha^2\alpha^4 + \alpha^2 x^6)}{\alpha^6 + \alpha^4\alpha^4} = \frac{\alpha^5(1 + 1 + \alpha^6 + \alpha)}{\alpha^6 + \alpha} = \alpha^5$$

The error and erasure polynomials are $e(x) = \alpha^5 x^3$ and $f(x) = \alpha^4 x^2 + \alpha^5 x^5$ and the corrected polynomial is

$$r(x) + e(x) + f(x) = (\alpha + \alpha^3 x + \alpha^4 x^3 + \alpha^6 x^4 + \alpha^3 x^6) + (\alpha^5 x^3) + (\alpha^4 x^2 + \alpha^5 x^5)$$

$$= \alpha + \alpha^3 x + \alpha^4 x^2 + \alpha^4 x^3 + \alpha^6 x^4 + \alpha^5 x^5 + \alpha^3 x^6$$

2.2 Block Interleaving

Interleaving is commonly used to break up correlated errors into random errors by rearranging the symbols. This is done because most block and convolutional codes are optimal for random errors. Interleaving causes correlated errors to be spread out over time, and then the coding system can handle the errors as if they were random. There are two major types of interleaving, block and convolutional. Block interleavers are used in conjunction with the concatenated systems in Figure 1.1, and will be the only method discussed here. Because the interleaver is to be used in conjunction with the symbol based Reed Solomon encoder and decoder, the interleaving will be done on a symbol level, rather than on a bit level. Block interleavers can be implemented using an MxN matrix. The symbols are fed into the matrix by column, and fed out by rows. At the deinterleaving stage, the symbols are fed into the matrix by row, and output by column. Consider this simple example. The sequence {0, 1, 2, 3, ..., 11} will be fed into a 3x4 block interleaver by column

$$\begin{matrix} 0 & 3 & 6 & 9 \\ 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \end{matrix}$$

The interleaver then outputs the data by row. The output sequence is {0, 3, 6, 9, 1, 4, 7, 10, 2, 5, 8, 11}. The deinterleaving is accomplished by entering the sequence by row, and outputting by column. The deinterleaved sequence is {0, 1, 2, 3, ..., 11}.

2.2.1 Redecoding of deinterleaving frame using erasure

Paaske [7] has developed a strategy to declare Reed Solomon symbol erasures using information provided by the deinterleaver. In order to understand the erasure declaring procedure discussed later in the report, and for comparison purposes, an overview of Paaske's method is presented below.

Because the Viterbi decoder produces burst errors at it's output, and the deinterleaver spreads these bursts over several codewords, it is highly likely that the burst errors will occur at identical positions in neighboring Reed Solomon codewords (RSW) in each deinterleaving frame. The output of the Viterbi decoder is fed into the deinterleaver by row, and then the columns are fed to the RS decoder as in Figure 2.4. Each column of the deinterleaver makes up a Reed Solomon codeword. Assume after errors only decoding of each of the RSW, that some of the codewords in the deinterleaver frame were decoded correctly (≤ 16 errors in the codeword) and some were undecodable (> 16 errors in the codeword). Because a RSW with v errors and ρ erasures can be corrected if $2 \cdot v + \rho \leq d_{min}$, declaring erasures and redecoding the deinterleaving frame may provide improvement. This improvement is highly dependent upon if the declared erasure hits an error. Erasures that hit errors will be called good erasures (GE) and ones that do not hit an error will be called bad erasures (BE).

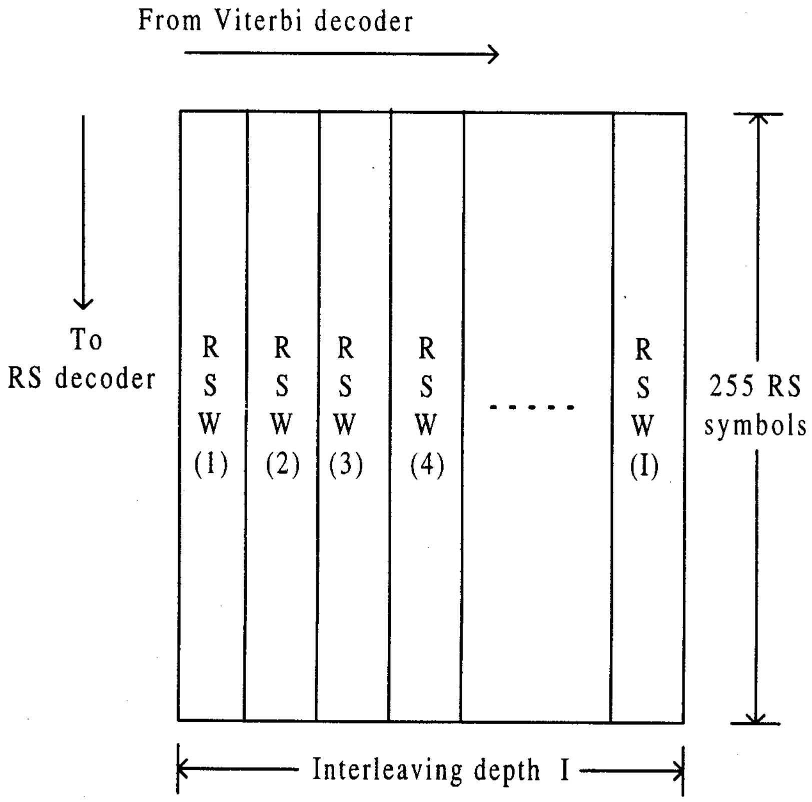


Figure 2.4 Typical deinterleaving frame

Paaske's method for declaring erasures in the de-interleaving frame makes use of the bursty nature of the Viterbi decoder output. Because the data is fed into the deinterleaver by row, the bursts will span over many RSW as in Figure 2.5. Let RSW(i) denote the i th codeword in an interleaving frame. It should be noted that a burst of length l at symbol k starting at RSW(i_1) will affect symbol k in RSW(i_1 +j) for $i_1 + j \leq I$ and symbol k + 1 in RSW(i_1 +j-I) for $i_1 + j > I$. If codeword RSW(i) has been correctly decoded, the positions where errors have occurred will be known. It is highly probable that the same symbols in the neighboring codewords will also be errors. Paaske developed 4 erasure declaring procedures (EP1-EP4), three of which declare erasures in the non-decoded RSW using information provided by the decoded RSW in the deinterleaving frame. A brief description of the procedures is presented below.

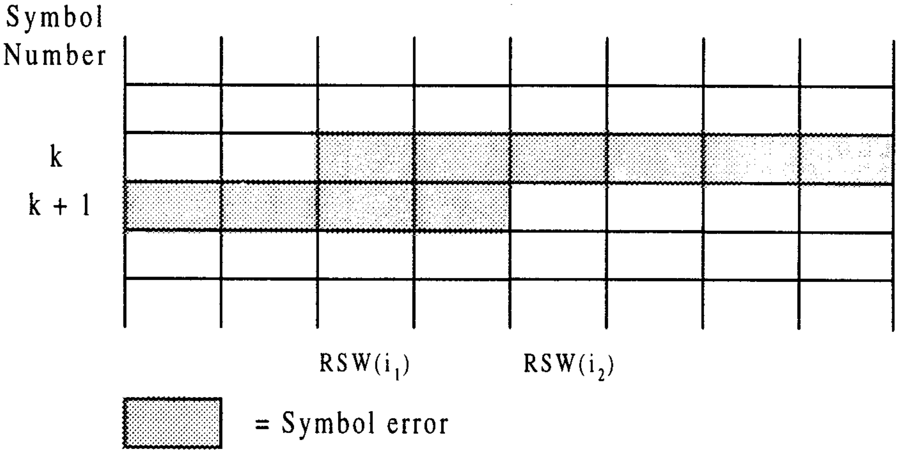


Figure 2.5 Typical burst error in a deinterleaving frame

EP1: Assume that two RSW have been decoded, and both contain errors in position k. Let RSW(i_1) and RSW(i_2) be two correctly decoded RSW with an error in position k in both as shown in Figure 2.6. Also assume that RSW(i_1+1) to RSW(i_2-1) have not been decoded. It is highly probable that position k is an error in these undecoded words, and is erased. These erasures are classified as double sided erasures (DSE) and the probability that a DSE is a GE is 0.96 [7]. An example of a DSE is shown in Figure 2.6.

EP2: Assume that a codeword RSW(i) has been successfully decoded and that RSW(i - 1) and RSW(i + 1) have not been decoded. For all error positions in RSW(i) erase the same positions in RSW(i - 1) and RSW(i + 1). These are called single sided erasures (SSE) and are GE with a probability of 0.60 [7]. An example of SSE declaration is given in Figure 2.7 where RSW(i) is the decoded word with errors in positions k_1, k_2, k_3, and k_4. Note that the SSE declared in RSW(i + 1) at position k_2 and RSW(i - 1) at position k_3 do not hit symbol errors, and are therefore BE.

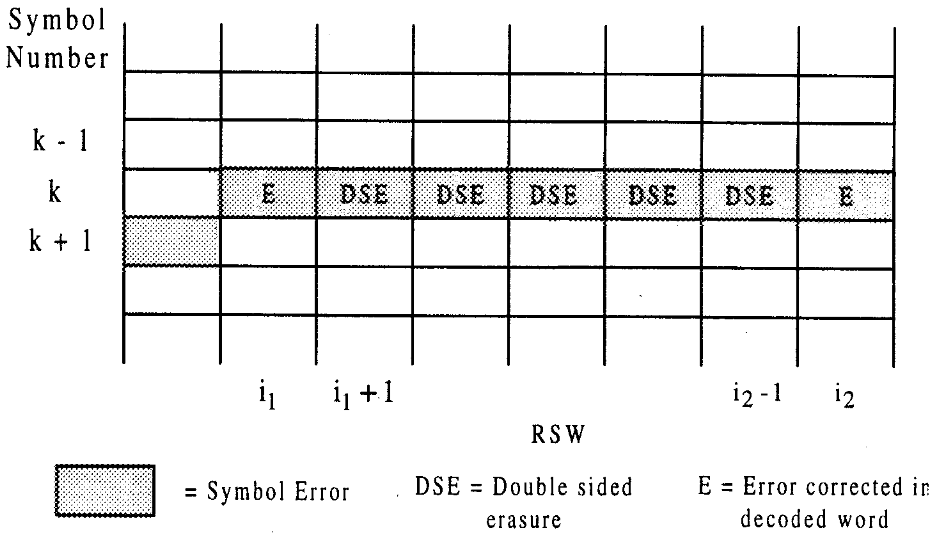


Figure 2.6 An example of a double sided erasure declaration

EP3: Assume that RSW(i) has been decoded and contains e errors, and RSW(i - 1) and RSW(i + 1) have not been decoded. Also assume that s_1 DSE can be obtained if EP1 is used. The s_1 DSE are combined with s_2 SSE chosen from the e - s_1 possible SSE. The optimal choice for the number of SSE s_2 is treated in [7].

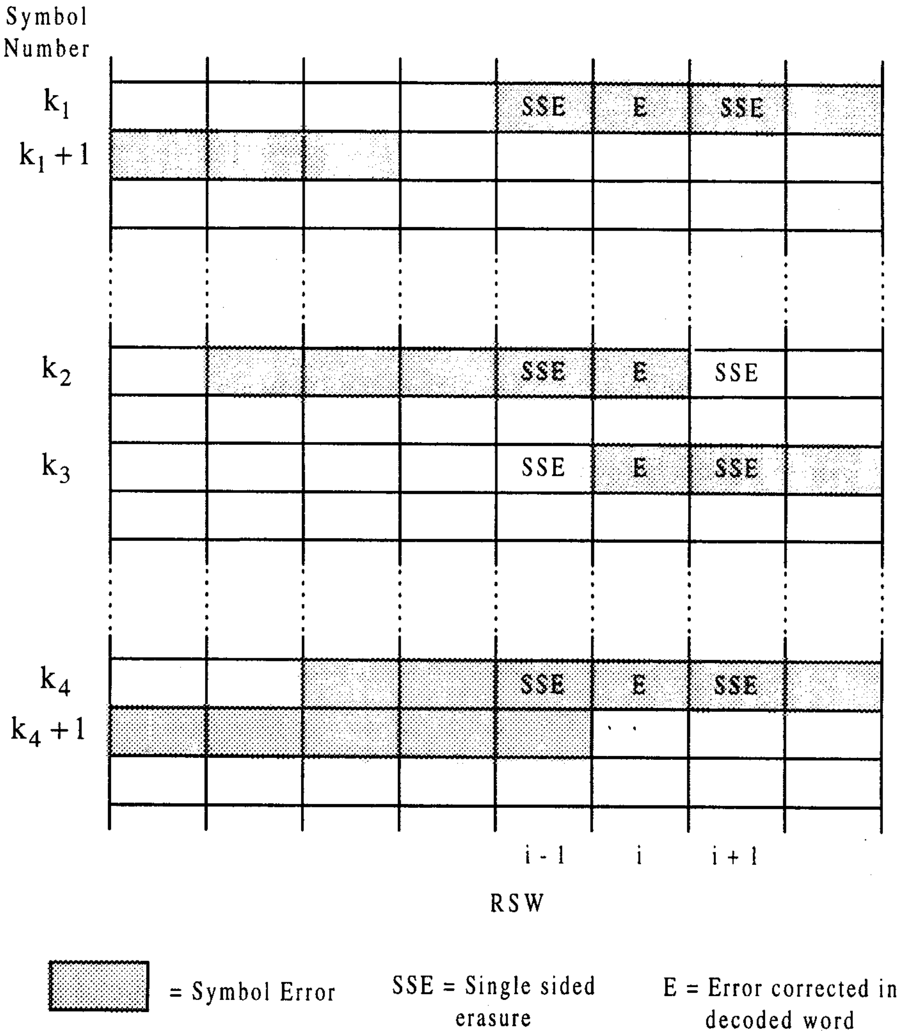


Figure 2.7 An example of single sided erasure declaration

EP4: This procedure assumes that one of the non-decoded RSW has 17 errors. Two symbols are selected and make erasures. If there are 17 errors, the probability that an erasure is a good erasure is 1/15.

The erasure declaring procedure proposed by Paaske [7] involves the following steps:

1) Try decoding each RSW in the deinterleaving frame using errors only decoding.

2) Set i_d equal to the number of successfully decoded RSW.

3) If $i_d = I$, go to step 9.

4) If $i_d = 0$, go to step 8.

5) Attempt to decode each non-decoded RSW using EP1.

6) Attempt to decode each non-decoded RSW using EP2.

7) Attempt to decode each non-decoded RSW using EP3.

8) Attempt to decode each non-decoded RSW using EP4.

9) Stop.

For steps 5) through 8), decoding is attempted on the first non-decoded RSW using the EP specified in the step. If this decoding attempt is successful, then proceed to step 2). If not successful, then try the next non-decoded RSW using the EP specified in the step. This continues until either one of the non-decoded RSW is successfully decoded, or all non-decoded RSW have been tried and none are successful. If all non-decoded RSW have been attempted using the given EP, and there are no successfully decoded RSW, then proceed to the next step. Erasure procedures EP3 and EP4 involve selecting erasures in a systematic way. This step is repeated on each codeword until either a successful decoding of the codeword, or a maximum number of trials T_{max} has been attempted. In the simulations conducted by Paaske, $T_{max} = 500$ trials.

2.3 Convolutional codes

Convolutional codes are fundamentally different than block codes. Block codes divide the information sequence into segments of length k, and map these k bits onto a codeword of length n. Convolutional codes on the other hand convert the entire data stream into one code word, regardless of the length of the information sequence. A (n, k, m) convolutional encoder has k inputs and n outputs, where $k < n$ and both k and n are small integers. The memory order m should be made large in order to achieve a high degree of error correcting capability [6].

2.3.1 Convolutional encoder

Convolutional codes are implemented using a linear feed forward shift register circuit. A typical encoding circuit is given in Figure 2.8, and will be used as a model for discussing convolutional encoders and the Viterbi decoder. The information sequence $u = (u_0, u_1, u_2, \ldots)$ is fed into encoder circuit k bits at a time. The memory elements are tapped and the bits contained in memory are added together using modulo-2 adders to obtain a pair of output data streams $v^{(0)} = (v_0^{(0)}, v_1^{(0)}, v_2^{(0)}, \ldots)$ and $v^{(1)} = (v_0^{(1)}, v_1^{(1)}, v_2^{(1)}, \ldots)$. These output sequences are combined to create the final codeword $v = (v_0^{(0)}, v_0^{(1)}, v_1^{(0)} v_1^{(1)}, v_2^{(0)}, v_2^{(1)}, \ldots)$.

The constraint length K is defined as the maximum number of output bits that can be affected by any input bit. Since each information bit stays in the encoder for m + 1 time units, and during each time interval the encoder produces n output bits, the constraint length is defined as $K = n \cdot (m+1)$.

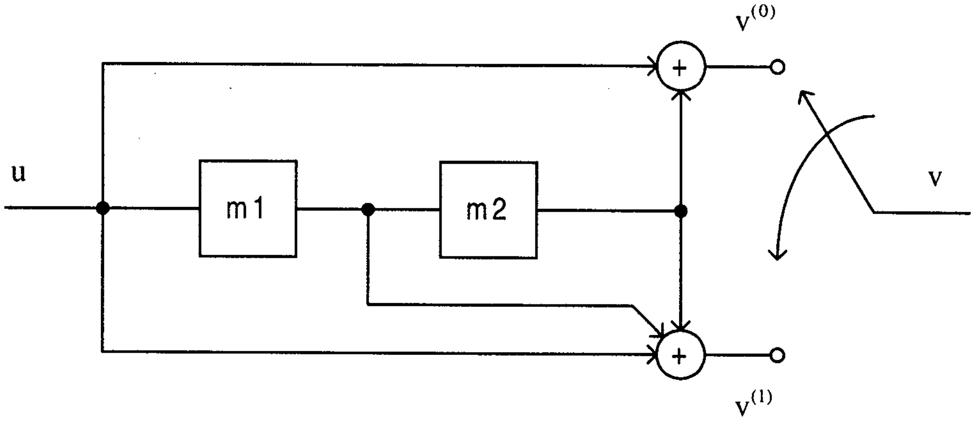


Figure 2.8 A (2, 1, 2) convolutional encoding circuit

The structure of convolutional encoders can be expressed in a number of ways. One of these ways is using the impulse response of the encoder. The impulse response of the encoding circuit is obtained by letting the input u = (1 0 0 0 ...) and observing the output sequences as u enters the encoding circuit. An encoder with memory m generates an impulse response of length m + 1. The impulse response, also known as the generator sequence, is written in the form $g^{(0)} = (g_0^{(0)}, g_1^{(0)}, g_2^{(0)}, \dots, g_m^{(0)})$ and $g^{(1)} = (g_0^{(1)}, g_1^{(1)}, g_2^{(1)}, \dots, g_m^{(1)})$. For the encoder in Figure 2.8 the generator sequences are $g^{(0)} = (1\ 0\ 1)$ and $g^{(1)} = (1\ 1\ 1)$. The two encoder output sequences can be thought of as a linear convolution of the information sequence with the impulse response. The encoding equations can be written as

$$v^{(0)} = u * g^{(0)} \tag{2.15a}$$

$$v^{(1)} = u * g^{(1)} \tag{2.15b}$$

where * denotes discrete convolution using modulo-2 operations. The output at time $t = \tau$ can be written as

$$v_\tau{}^{(j)} = \sum_{i=0}^{m} u_{\tau-i} \cdot g_i{}^j$$

$$= u_\tau \cdot g_0{}^{(j)} + u_{\tau-1} \cdot g_1{}^{(j)} + u_{\tau-2} \cdot g_2{}^{(j)} + \ldots + u_{\tau-m} \cdot g_m{}^{(j)}. \tag{2.16}$$

For the encoder of Figure 2.5, (2.15) reduces to

$$v_\tau{}^{(0)} = u_\tau \oplus u_{\tau-2} \tag{2.17a}$$

$$v_\tau{}^{(0)} = u_\tau \oplus u_{\tau-1} \oplus u_{\tau-2} \tag{2.17b}$$

where ⊕ denotes modulo-2 addition. The encoding of the information sequence u = (1 0 1 1 0 1) is illustrated in Table 2.6. At time = 0, the contents of memory are initially set to zero. At time = 1, the first information bit is fed into the encoder, and the output of the encoder is obtained by using (2.17). This process continues until all of the information bits have entered into the encoder. At this point, information bits are still contained in memory. Two more clock cycles are needed to move the last bits through the encoder. ($k \cdot m$) zeros are fed into the input to move the last information bits through the encoding circuit. The encoded sequence is v = (11 01 00 10 10 00 01 11).

2.3.2 Decoding of Convolutional codes

The Viterbi algorithm is a widely used method for the decoding of convolutional codes. The algorithm was developed by A. J. Viterbi in 1967 [6], and is a maximum likelihood decoder.

Table 2.6 Encoding of the information sequence u = (1 0 1 1 0 1)

Time	Input	m_1	m_2	$v^{(0)}$	$v^{(1)}$
0	-	0	0	-	-
1	1	1	0	1	1
2	0	0	1	0	1
3	1	1	0	0	0
4	1	1	1	1	0
5	0	0	1	1	0
6	1	1	0	0	0
7	-	0	1	0	1
8	-	0	0	1	1

2.3.2.1 State Diagram

For every encoding circuit, there will be a corresponding state diagram. An encoder with memory m will have 2^m possible states, illustrating the contents of the shift registers in the encoding circuit. There are k binary inputs to the encoder for each clock cycle, which results in 2^k branches entering and exiting each state. The process details used to create the state diagram are given in Table 2.7. For the encoding circuit of Figure

2.8, there will be 4 states, and 2 branches entering and exiting each state. The state diagram for the circuit in Figure 2.8 is given in Figure 2.9.

Table 2.7 Development of the state diagram

Initial State	m_1	m_2	Input u	Output v	New State
S_0	0	0	0	0 0	S_0
S_0	0	0	1	1 1	S_1
S_1	1	0	0	0 1	S_2
S_1	1	0	1	1 0	S_3
S_2	0	1	0	1 1	S_0
S_2	0	1	1	0 0	S_1
S_3	1	1	0	1 0	S_2
S_3	1	1	1	0 1	S_3

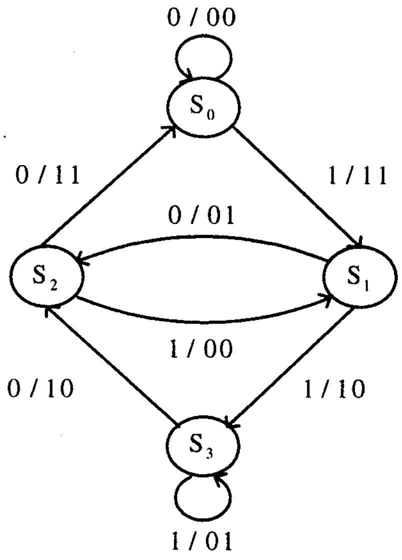


Figure 2.9 State transition diagram for the encoder in Figure 2.8

2.3.2.2 Trellis Diagram

A trellis diagram is a state diagram extended to include the passage of time. The encoding of a sequence of data corresponds to a unique path through the trellis diagram. The trellis diagram for the state diagram in Figure 2.8 is shown in Figure 2.10. If an information sequence of length $k \cdot L$ bits is fed through an encoding circuit, where L is the total number of k bit codewords, the resulting codeword will be of length $N = n \cdot (L + m)$ bits. Each of the 2^L code words of length N is represented by a unique path through the trellis.

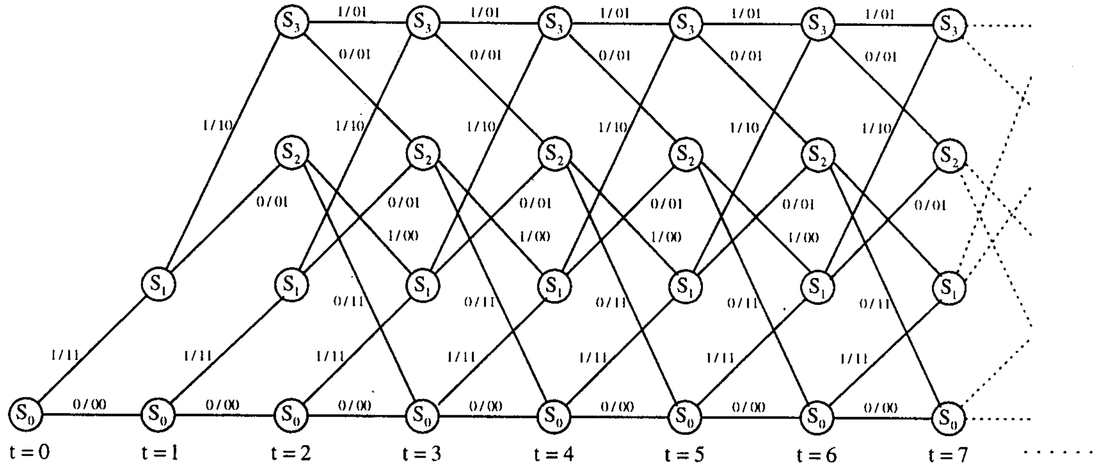


Figure 2.10 Trellis diagram (based on the encoder in Figure 2.8)

The convolutional coding problem is shown in Figure 2.11. Assume an information sequence $u = (u_0, u_1, \dots, u_{L-1})$ of length $k \cdot L$ bits is encoded into a code word

$y = (y_0, y_1, \ldots, y_{L+m-1})$ of length $N = n \cdot (L+m)$ bits. A noise-corrupted version of the transmitted sequence r is received where r and y have the following form

$$r = (r_0^{(0)}, r_0^{(1)}, \ldots, r_0^{(n-1)}, r_1^{(0)}, r_1^{(1)}, \ldots, r_{L+m-1}^{(n-1)})$$

$$y = (y_0^{(0)}, y_0^{(1)}, \ldots, y_0^{(n-1)}, y_1^{(0)}, y_1^{(1)}, \ldots, y_{L+m-1}^{(n-1)}) .$$

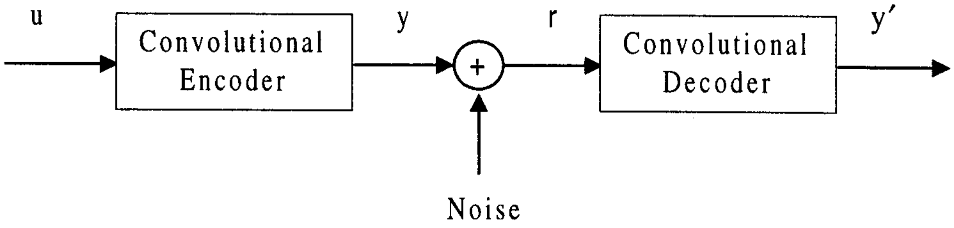


Figure 2.11 Convolutional coding system

The Viterbi algorithm generates the estimate y' of the transmitted sequence r which maximizes the conditional probability $p(r \mid y)$. Assuming the channel is memoryless, each received bit will be independent of the noise process affecting all of the other bits. Therefore $p(r \mid y)$ can be expressed as

$$p(r \mid y) = \prod_{i=0}^{L+m-1} [\, p(r_i^{(0)} \mid y_i^{(0)}) \cdot p(r_i^{(1)} \mid y_i^{(1)}) \cdots p(r_i^{(n-1)} \mid y_i^{(n-1)})]$$

$$= \prod_{i=0}^{L+m-1} (\prod_{j=0}^{n-1} p(r_i^{(j)} \mid y_i^{(j)}) \qquad (2.18)$$

The log-likelihood function is obtained by taking the logarithm of each side of (2.18).

$$\log p(r \mid y) = \sum_{i=0}^{L+m-1} \sum_{j=0}^{n-1} \log p(r_i^{(j)} \mid y_i^{(j)}) \qquad (2.19)$$

This is done because it is, in general, easier to implement summations rather than multiplication in hardware. The log likelihood function, $\log p(r \mid y)$, is called the Path Metric associated with the path y and is denoted $M(r \mid y)$. The terms $\log p(r_i^{(j)} \mid y_i^{(j)})$ are called Bit Metrics

$$M(r_i^{(j)} \mid y_i^{(j)}) = \log p(r_i^{(j)} \mid y_i^{(j)}) \tag{2.20}$$

In the hardware implementation of the Viterbi decoder, it is more convenient to use positive integers for the metric values rather than the actual bit metrics. This can be accomplished by using

$$M(r_i^{(j)} \mid y_i^{(j)}) = a \cdot [\log p(r_i^{(j)} \mid y_i^{(j)}) + b] \tag{2.21}$$

where a and b are chosen to obtain small, positive integer values for the metrics which can be implemented easier in hardware. The path metric for a codeword y is then calculated as

$$M(r \mid y) = \sum_{i=0}^{L+m-1} \sum_{j=0}^{n-1} M(r_i^{(j)} \mid y_i^{(j)}). \tag{2.22}$$

The k th branch metric for a codeword y is defined as the sum of the bit metrics

$$M(r_k \mid y_k) = \sum_{j=0}^{n-1} M(r_k^{(j)} \mid y_k^{(j)}). \tag{2.23}$$

The k th partial metric for a path is obtained by summing all of the branch metrics for the first k branches the path follows.

$$M^k(r \mid y) = \sum_{j=0}^{k-1} M(r_i \mid y_i) \tag{2.24}$$

$$= \sum_{i=0}^{k-1} \sum_{j=0}^{n-1} M(r_i^{(j)} \mid y_i^{(j)}). \tag{2.25}$$

The Viterbi Algorithm finds the path through the trellis with the largest path metric, which is the maximum likelihood estimate y' of the received word r. At each time interval the algorithm, computes the partial metrics entering each state. The largest metric is chosen as the surviving path at each state, and all other paths entering that state are discarded. This process is continued until the end of the trellis is reached. The final surviving path is the maximum likelihood estimate y of the codeword.

2.3.2.3 The Viterbi Algorithm [6]

1. At time $t = m$, compute the partial metric for the single path entering each state. Store the value of this metric at each state.

2. Increase t by 1. Compute the partial metric for the path entering each state. This will be equal to the branch metric entering the state plus the surviving metric from the previous state. Out of the 2^k paths entering each state, the path with the largest metric is chosen and the remaining paths are discarded. The metric of the surviving path is stored at each state.

3. If $t < L + m$, repeat step 2. If not, stop. At time $t = L + m$, all paths have returned to the all zero state. There will be only one path remaining, and this path is the maximum likelihood estimate y'.

2.3.2.4 Hard Decision Decoding

In hard decision decoding, the receiver determines whether a zero or one was transmitted. These zeros and ones are the input to the Viterbi decoder. If the channel is memoryless and if the probability of a bit error is independent of the transmitted bit, then the channel is said to be a binary symmetric channel (BSC). The BSC is shown in Figure 2.12 where p is the probability that a bit is in error. The conditional probabilities for the BSC are given in Table 2.8.

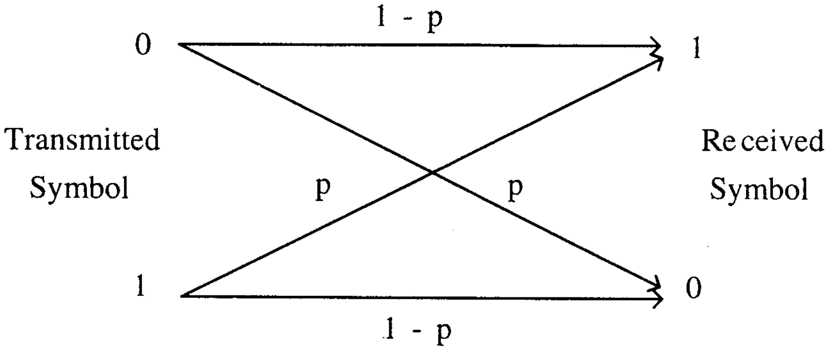


Figure 2.12 Binary symmetric channel model

Table 2.8 Conditional probabilities for BSC

$p(r_i^{(j)} \mid y_i^{(j)})$	$y_i^{(j)} = 0$	$y_i^{(j)} = 1$
$r_i^{(j)} = 0$	1 - p	p
$r_i^{(j)} = 1$	p	1 - p

For the BSC, choosing $a = [\log_2 p - \log_2(1-p)]^{-1}$ and $b = -\log_2(1-p)$ in (2.21) yields the bit metrics in Table 2.9 [6]. The maximizing of the bit metrics $M(r_i^{(j)} \mid y_i^{(j)})$ coincides with the minimization of the Hamming distance. For the BSC case, the path metric is simply the Hamming distance d(r, y), and the Viterbi algorithm will choose the surviving paths as the ones that have the minimum partial path metrics.

Table 2.9 Metric table for BSC

$M(r_i^{(j)} \mid y_i^{(j)})$	$y_i^{(j)} = 0$	$y_i^{(j)} = 1$
r = 0	0	1
r = 1	1	0

Consider the information sequence u = (1 0 1 1 0 1) that was encoded using the (2, 1, 2) convolutional encoder in Figure 2.8. The encoded sequence is y = (11 01 00 10 10 00 01 11). If this sequence is transmitted on a BSC, and no errors occur in the transmission, r = y. The decoding of this received sequence is illustrated in Figure 2.13. Note that the final path has a path metric value of 0. The decoded sequence is (1 0 1 1 0 1) which is obtained by tracing back the maximum likelihood path noting the input bit associated with each branch.

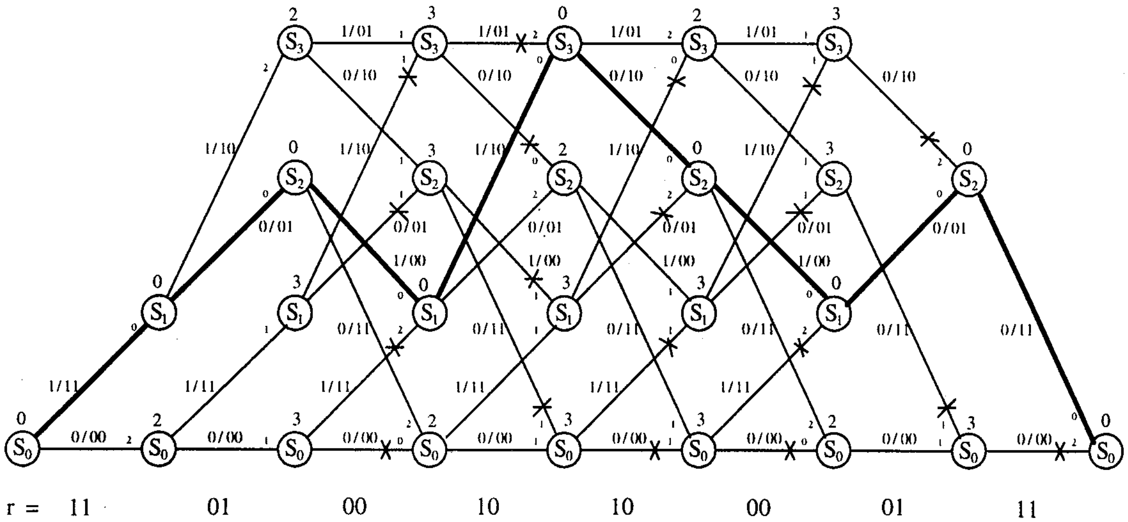


Figure 2.13 Hard decision Viterbi decoding of r = (11 01 00 10 10 00 01 11)

If the same sequence y = (11 01 00 10 10 00 01 11) is transmitted and the received sequence is r = (11 $\bar{1}$1 00 10 $\overline{00}$ 00 $\bar{1}$1 11) where the erroneous bits are denoted with the bar over the bit, the decoding is illustrated in Figure 2.14. The decoded sequence is (1 0 1 1 0 1), which is identical to the information sequence. The decoder corrected the three errors in the received sequence.

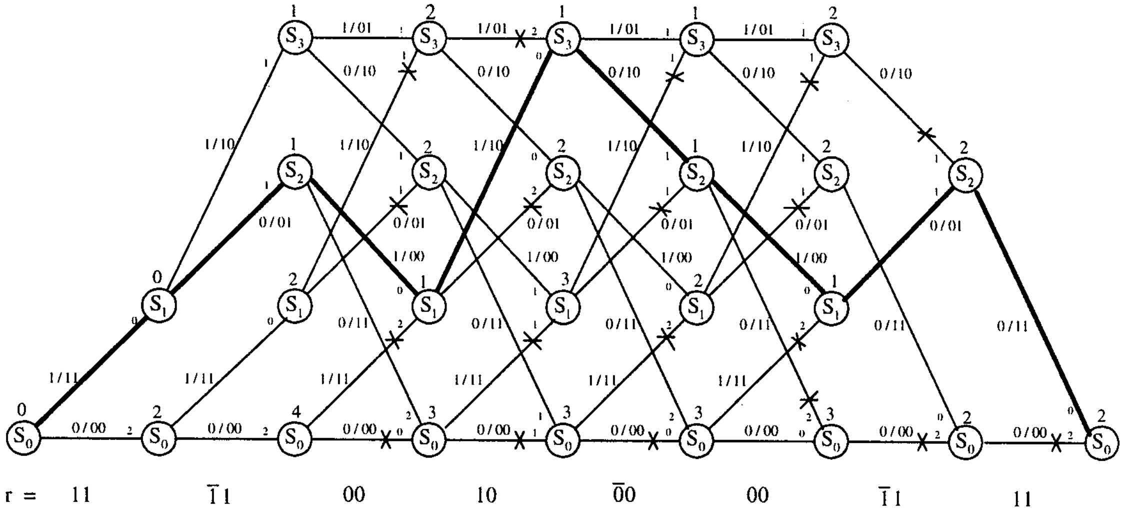


Figure 2.14 Hard decision Viterbi decoding
of r = (11 $\bar{1}$1 00 10 $\overline{00}$ 00 $\bar{1}$1 11)

2.3.2.5 Soft Decision Decoding

Hard decision decoding simply assigns a zero or a one at the receiver, utilizing only two decision regions. Soft decision decoding makes use of q-bit quantization which results in multiple decision regions ranging from a “strong-one” to a “strong-zero”. Using soft decisions results in approximately 2 dB gain over hard decision Viterbi decoding [9]. A discrete memoryless channel (DMC) is shown in Figure 2.15. The

DMC is completely described by a set of transition probabilities between a zero or a one being transmitted and one of $Q = 2^q$ levels at the receiving end. It has been found that

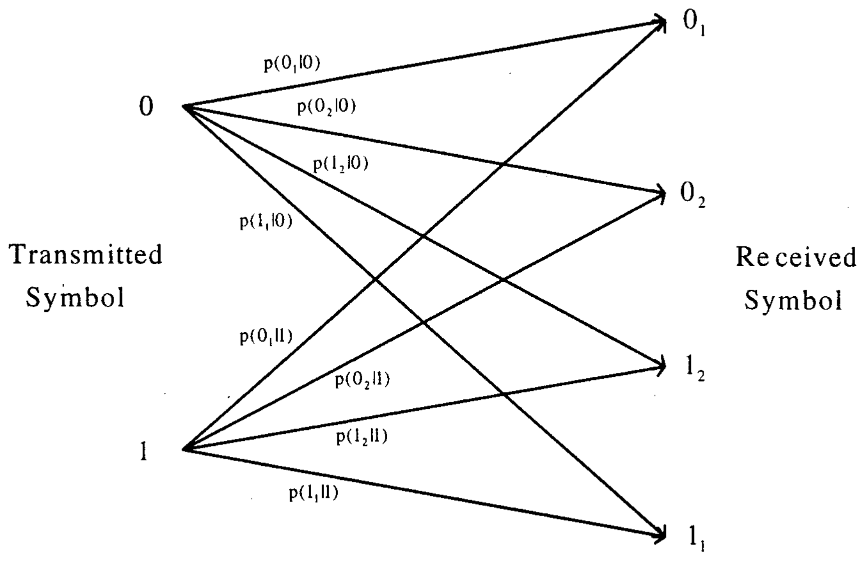


Figure 2.15 DMC channel model for Q = 4 levels

using 8-level quantization results in only a 0.25 dB loss when compared with using infinitely fine quantization [11]. Consider the transition probabilities given in Table 2.10. The modified metric table is given in Table 2.11, and is obtained by using (2.21) with b = 1 and a = 17.3. In choosing a and b, a is typically chosen to obtain a metric value equal to zero for the smallest metric value. The metric values obtained by using (2.20) are, in general, real valued. Simply rounding these values off to the nearest integer may lead to round off errors. The scaling factor b in (2.21) is chosen to make the metrics as close as possible to being integer values, while keeping the values as low as possible. This will reduce some of the error that may occur when rounding off the metric values.

If the same codeword y = (11 01 00 10 10 00 01 11) is transmitted over the channel, and the received sequence is $r = (1_1 1_2\ 0_1 1_1\ 0_1 0_2\ 1_2 1_2\ 1_2 0_1\ 0_1 0_2\ 0_2 0_2\ 0_2 0_1)$, then the decoding process using soft decision Viterbi decoding is illustrated in Figure 2.16. The decoded sequence is u = (1 0 1 1 0 1), which is identical to the information sequence that was transmitted.

Table 2.10 Conditional probabilities for DMC

$p(r_i^{(j)} \mid y_i^{(j)})$	$y_i^{(j)} = 0$	$y_i^{(j)} = 1$
$r_i^{(j)} = 0_1$	.4	.1
$r_i^{(j)} = 0_2$	.3	.2
$r_i^{(j)} = 1_2$	.2	.3
$r_i^{(j)} = 1_1$	.1	.4

Table 2.11 Metric Table for DMC

$M(r_i^{(j)} \mid y_i^{(j)})$	$y_i^{(j)} = 0$	$y_i^{(j)} = 1$
$r = 0_1$	10	0
$r = 0_2$	8	5
$r = 1_2$	5	8
$r = 1_1$	0	10

2.3.2.6 Truncation length

In practice, information sequences are very long. It is not practical to wait until the entire sequence is received to begin decoding. This would result in long delays and require large amounts of storage. It has been found that a decision can be made on the k information bits that were received (t - δ) time units before, where δ is called the decision depth, or truncation length. If the truncation length is made 4 to 5 times the constraint length, there will be very little loss in performance [10].

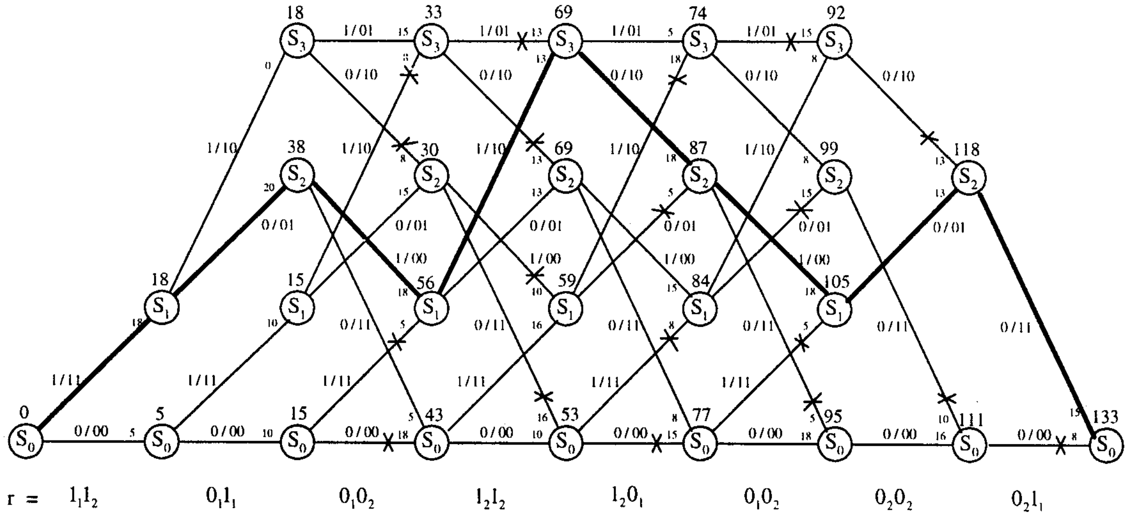


Figure 2.16 Soft decision Viterbi decoding of $r = (1_1 1_2\ 0_1 1_1\ 0_1 0_2\ 1_2 1_2\ 1_2 0_1\ 0_1 0_2\ 0_2 0_2\ 0_2 0_1)$

The implementation of the truncated Viterbi decoder makes use of 2^m shift registers, each of length $k \cdot \delta$. At any time t, there are 2^m surviving paths, with one surviving path terminating in each of the 2^m states. For each surviving path, the only information that needs to be stored are the information bits associated with that path. No information about the route the path took is necessary, just the information (output) bits associated with that path. At time t, n bits are input into the decoder. The branch metric is calculated, and the surviving path is chosen as the path with the largest metric, as in the standard Viterbi decoder. The path information for each state at time t is equal to the previous state at time (t - 1) shifted k bits to the left. The k information bits associated with the surviving branch at time t are then shifted into the register. Consider the Viterbi soft decoding example given in Figure 2.16. For a decoding depth of δ = 5, the shift registers, surviving branches, and metric values are shown in Figure 2.17. At time t = 0, the shift registers are empty. At each time interval, the survivor branch is chosen, and the contents of the shift register at the pervious state are copied to the new state, shifted to the left, and the information bit associated with the branch is inserted into the register. This process continues until the register is full (i. e. t ≥ δ). At this point, the path with the highest metric is chosen as the surviving path. Only one bit is output at a time, so this corresponds to the leftmost position in the shift register, which was the information bit

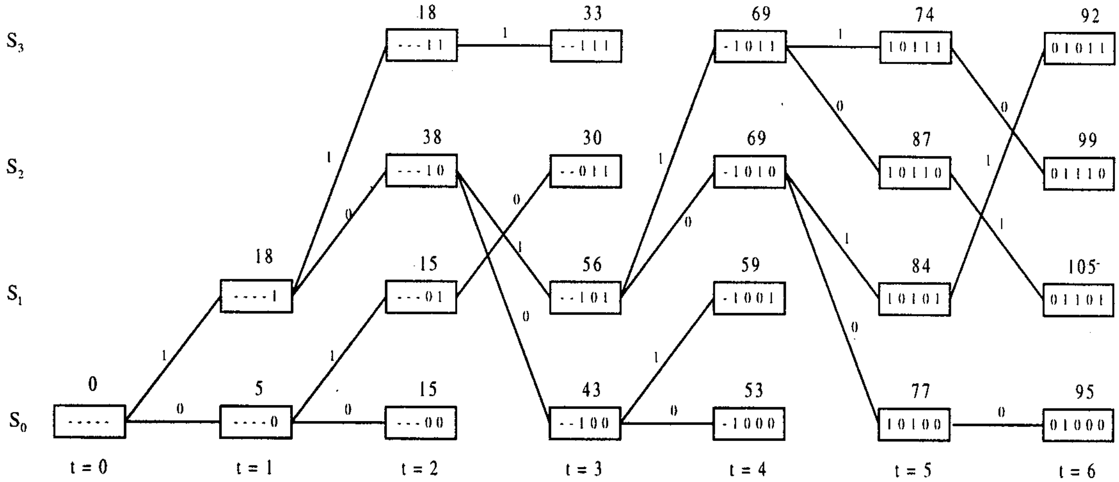


Figure 2.17 Shift register contents for the soft decision decoding in Figure 2.16

inserted t - δ time units before. In Figure 2.17 at t = 5 = δ, the path with the highest metric has a path metric of 87, and terminates in state S_2. The leftmost bit in the shift register is equal to 1, and is the output for this time interval. Now, since t ≥ δ, the decoder can take in n bits, compute the path metric and determine the surviving path for each state. The decoder then chooses the path with the maximum metric, and outputs the leftmost bit. At time t = 6 in Figure 2.17, the path with the highest metric has a path metric equal to 105, and terminates in state S_1. The leftmost bit in that shift register is 0, and is the output bit. This process continues for the remainder of the decoding operation.

2.3.3 Soft Output Viterbi Algorithm

The Viterbi algorithm can be modified to give either a reliability value or a probability that a given bit is correct. The method used to implement the Soft Output Viterbi Algorithm (SOVA) is based upon Hagenauer and Hoeher's method [4]. For simplicity, this discussion will only consider convolutional codes where k = 1. The reliability of a binary random variable can be defined in terms of a log-likelihood value

$$L(u) = \log \frac{\mathrm{Prob}(u=1)}{\mathrm{Pr\,ob}(u=0)} \tag{2.26}$$

where the sign of L(u) corresponds to the hard decision (i.e. if L(u) > 0, u = 1 and if L(u) < 0, u = 0) and the magnitude |L(u)| is the reliability of this decision. The larger the magnitude, the greater the reliability of the decision.

At each of the 2^m states at time t, the Viterbi decoder selects the survivor path as the one with the largest path metric. The accumulated path metric and the path probability are related by

$$M(r \mid y) = \log_2 p(r \mid y) = \sum_{i=1}^{t} \sum_{j=0}^{n-1} \log p(r_i^{(j)} \mid y_i^{(j)}). \tag{2.27}$$

If the bit metrics are given by $M(r_i^{(j)} \mid y_i^{(j)}) = a \cdot [\log_2 p(r_i^{(j)} \mid y_i^{(j)}) + b]$, then the path probability is

$$p(r \mid y) = e^{\frac{\ln 2}{a} M(r \mid y) - ntb} \tag{2.28}$$

Each state S_k ($k = 0, 1, \ldots, 2^m - 1$) will have two entering paths, a survivor path with metric M_1 and a competing path with metric M_2 ($M_1 > M_2$). The probability of choosing the wrong path is given by

$$p_{S_k} = \frac{\text{Prob(path 2)}}{\text{Prob(path 2)} + \text{Prob(path 1)}} \tag{2.29}$$

$$= \frac{e^{\frac{\ln 2}{a} M_2 - nkb}}{e^{\frac{\ln 2}{a} M_2 - nkb} + e^{\frac{\ln 2}{a} M_1 - nkb}} \tag{2.30}$$

$$= \frac{1}{1 + e^{\Delta}} \quad \text{where } \Delta = \frac{\ln 2}{a}(M_1 - M_2) \tag{2.31}$$

With this probability the Viterbi decoder has made an error in the path it has chosen as the survivor path. Consider the two paths merging in state S_0 at time t in Figure 2.18. The all zero path is the survivor path with metric M_1, and the other is the competing path with metric M_2. The two paths are the same up to time t - δ_m (δ_m = 6 in this case). At this point, the paths diverge and there are three differing information bits between the two

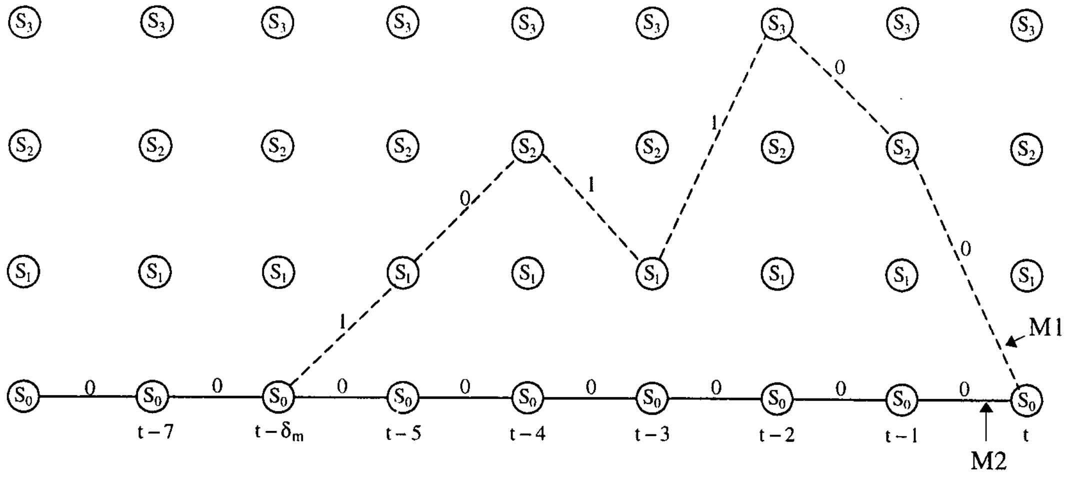


Figure 2.18 Example of the SOVA

paths at times t-2, t-3, and t-5. Using the probability of selecting the wrong path given in (2.31), the probability associated with each bit p_j can be modified for all bits u_j for times j = t, t-1, ..., t - δ_m [4].

$$p_j = p_j \text{ if } u_{1_j} = u_{2_j} \tag{2.32a}$$

$$p_j = p_j(1-p_{S_k}) + (1-p_j)p_{S_k} \quad \text{if } u_{1_j} \neq u_{2_j} \tag{2.32b}$$

where u_{1_j} and u_{2_j} are the output bits at time j on paths 1 and 2 respectively. The first case can be neglected because choosing path 2 instead of path 1 would result in no error for the j th bit. Because the case in (3.32a) can be neglected, there is only a need to check $u_{1_j} \neq u_{2_j}$ for times t-m, t-m-1, ..., t-δ_m+1. This is because in order to terminate in any given state at time t, the m input bits prior to time t must be equal in order to create the given state, S_k. The probability in (2.32b) can be transformed into a bit reliability. From (2.26), the log-likelihood for the j th bit can be expressed as

$$L_j = \log \frac{1 - p_j}{p_j} \tag{2.33}$$

This can be combined with (2.31) and (2.32) to obtain an expression for updating the likelihood function.

$$L_j = \log \frac{1 - p_j(1 - p_{sk}) + (1 - p_j)p_{sk}}{p_j(1 - p_{sk}) + (1 - p_j)p_{sk}}$$

$$= \log \frac{1 - p_j(\frac{e^{\Delta}}{1 + e^{\Delta}}) - (1 - p_j)\frac{1}{1 + e^{\Delta}}}{p_j(\frac{e^{\Delta}}{1 + e^{\Delta}}) + (1 - p_j)\frac{1}{1 + e^{\Delta}}}$$

$$= \log \frac{p_j + e^{\Delta} - p_j e^{\Delta}}{p_j e^{\Delta} + 1 - p_j} \quad = \log \frac{1 + \frac{1 - p_j}{p_j} e^{\Delta}}{e^{\Delta} + \frac{1 - p_j}{p_j}}$$

$$L_j = \log \frac{1 + e^{L_j + \Delta}}{e^{\Delta} + e^{L_j}} \tag{2.34}$$

A good approximation of this expression is to simply take the minimum of L_j and Δ as the new reliability [4].

$$L_j = \min(L_j, \Delta) \tag{2.35}$$

For register exchange mode with truncation depth δ, each state S_k will have a shift register of $\delta \cdot q$ bits where q-1 bits represent the magnitude of the likelihood value L_j

and one bit for the sign of L_j, which corresponds to the output bit u_j. The procedure can be summarized as follows. For every state at time t, compute the path metrics for the two paths entering state S_k. Choose the path with the higher metric as the survivor path and update the path information in the register for S_k. The reliability at time t, L_t is initially set to ∞. Compute the metric difference $\Delta = \frac{\ln 2}{a}(M_1 - M_2)$. For j = t - m, t - m - 1, ..., t - δ_m compare the information of the two paths. If $u_{1_j} \neq u_{2_j}$, then update the new reliability using (2.35). After the surviving paths in each state have been determined, and the reliability information has been updated for each state, the state with the highest path metric is determined and the reliability and output bit at t - δ are the output for the decoder for time t.

Chapter 3

Simulation Techniques

Computer simulation plays an important role in the design and testing of communications systems [5]. The results obtained from simulation can give a good indication of how an actual system will perform under a variety of conditions. Performance evaluation of complex communication systems using analytical techniques can be difficult, if not impossible. Testing of the concatenated coding schemes presented in this report using computer simulation will give an estimate of how these codes perform under realistic conditions. In addition, the codes will be simulated using an "ideal" AWGN channel to test how the coding systems will perform under ideal conditions. Monte Carlo techniques were used to obtain the results for both simulations. Monte Carlo techniques are relatively simple to implement. Data is generated at the input to the simulation. This data is then run through the system being simulated. The data at the output of the simulation is compared to the data at the input to determine the number of errors. The probability of an error is simply the number of errors divided by the total number of simulation points. To be statistically confident in the results, the simulation should produce at least 50 errors [5]. For small values of bit error rate, a large number of bits will be needed, which results in longer processing times. This large amount of processing time is one of the drawbacks of using Monte Carlo techniques. The first step in simulating a communications system is to describe the system in block diagram form, where each block represents a signal processing operation. The model used for this simulation is given in Figure 3.1.

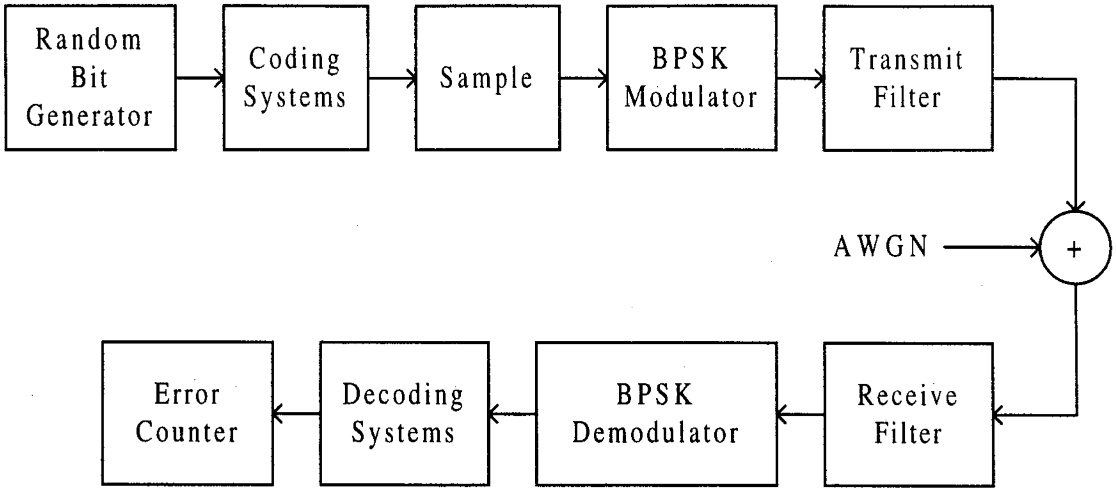


Figure 3.1: Simulation system model

Simulation can either be done at baseband or bandpass. Baseband simulations have no carrier frequency. This reduces the complexity of the system models such as the filters. Bandpass simulations require a higher sampling frequency than baseband, and therefore, more computational time. Bandpass simulations are necessary when studying upconversions, downconversions, and the effects of adjacent channel interference. Most simulations that involve a single information signal can be done at baseband [13]. The simulation used in this report was performed at baseband.

3.1 Random Number Generators

The various signals that exist in communication systems are random in nature. The information signals found in communication systems typically use random voltage or current waveforms to transmit information from one place to another. Noise is an unwanted random signal, and causes errors in the information being sent. In order to represent the random signals found in communication systems, a random number generator (RNG) will be needed. Random number generators do not produce truly random numbers, but a sequence of "pseudo" random numbers which repeat after a period of time. These sequences should be stationary and uncorrelated. The mean, variance, and other parameters computed for different segments of the RNG sequence should be equal for a RNG to be considered stationary [5]. Having a RNG with a period less than the simulation length will cause correlation in the RNG sequence. Choosing a RNG with a large period is desirable to avoid correlation.

3.1.1 Uniform Random Number Generator

Uniform RNG generate equiprobbile numbers within a given interval, typically between zero and one. Uniform RNG can be generated using the following multiplicative congruential algorithm

$$I_{j+1} = a \cdot I_j \text{ (modulus m)} \tag{3.1}$$

where a and m are integer constants. If a and m are chosen carefully, (3.1) will produce a sequence of random numbers with a maximum period of m. The random number generator used in this simulation can be found in [8] and has a very long period of $\approx 2.3 \text{ x } 10^{18}$. This is accomplished by combining two RNG sequences with

$m_1 = 2147483563$

$a_1 = 40014$

$m_2 = 2147483399$

$a_2 = 40692.$

In addition to a long period, this RNG passes all of the relevant statistical tests [8]. Assuming the probability of a '0' and a '1' are equal, the random bit stream can be obtained by using a uniform random number with the following conversion

$b_j = 0 \text{ if } I_j \leq 0.5$

$b_j = 1 \text{ if } I_j > 0.5$

3.1.2 Gaussian Random Number Generator

There are situations that call for random numbers with different distributions, and are typically generated by performing a transform of a uniform deviate. The generation of Additive Gaussian White Noise (AWGN) in simulations calls for a sequence of normally distributed random numbers. This normally distributed sequence can be generated by using the Box Muller Method [5, 8]. If X_1 and X_2 are two independent variables with uniform distribution between 0 and 1, then

$$Y_1 = \mu + \sigma \cdot \sqrt{-2 \cdot \ln(X_1)} \cdot \cos(2 \cdot \pi \cdot X_2) \tag{3.2a}$$

and

$$Y_2 = \mu + \sigma \cdot \sqrt{-2 \cdot \ln(X_1)} \cdot \sin(2 \cdot \pi \cdot X_2) \tag{3.2b}$$

are independent Gaussian variables with mean μ and standard deviation σ.

3.2 Sampling

In order to represent the signal in the simulation, the signals will need to be sampled. The Nyquist rate of $2 \cdot B$ is the minimum sampling rate for bandlimited signals of bandwidth B. For simulations, the sampling rate needs to be much higher to accurately represent the analog signal and to reduce the affects of aliasing. The number of samples per symbol should be an even integer between 8 and 16. Having the number of samples per symbol greater than 16 is not necessary for most simulations [13].

3.3 Filters

In Communication systems, filters are needed for the purposes of bandlimiting signals and rejecting out of band noise. The filtering in the process can produce something known as intersymbol interference.

3.3.1 Intersymbol Interference

Consider the effects of passing a series of impulses spaced T_b seconds apart through a low pass filter. Each impulse produces its own output from the filter. The output from one pulse extends into the output of pulse that starts T_b seconds later. This is known as Intersymbol Interference (ISI) and it can produce errors at the receiver.

The effects of ISI can be avoided by an appropriate choice of a low pass filter. Nyquist proposed a technique that theoretically produces zero ISI. This is accomplished by creating in the receiver a pulse that resembles the sin x/x shape, crossing the axis at intervals of T_b. The receiver samples the incoming wave at intervals of T_b, so at the sampling instant, the tails of the preceding outputs are crossing the axis, and are therefore zero. The only non-zero component is the pulse to be sampled, which solves the problem of interference from other symbols. The filter proposed by Nyquist is the "Raised Cosine" filter and theoretically produces zero ISI. The transfer function for this filter is

$$H(f) = \begin{cases} 1.0 & \text{for } |f| \leq f_l \\ \cos^2 \left(\frac{\pi \cdot (f - f_l)}{4 \cdot (B_0 - f_l)}\right) & \text{for } f_l < |f| \leq 2 \cdot B_0 - f_l \\ 0 & \text{for } |f| > 2 \cdot B_0 - f_l \end{cases}$$

where $B_0 = \frac{R_b}{2}$ is the filter bandwidth, α is the rolloff factor. The frequency f_l and the bandwidth B_0 are related by

$$\alpha = 1 - \frac{f_l}{B_o} \qquad 0 \leq \alpha \leq 1$$

The frequency response for different rolloff factors is given in Figure 3.2. The minimum bandwidth value of $B_0 = \frac{R_b}{2}$ is obtained when $\alpha = 0.0$. This value of rolloff is not obtainable in practice. Practical filters use rolloff values ranging from 0.2 to 1.0 [12].

In some applications, the filtering operation to produce zero ISI needs to be split between two filters, with one at the transmitter and one at the receiver. The optimum partition in the sense of optimizing the signal to noise ratio is to divide the filter transfer function equally between transmit and receive filter [5, 18].

$$H_{\text{Transmit}}(f) = H_{\text{Receive}}(f) = \sqrt{H(f)}$$

This is known as a Square Root Raised Cosine filter.

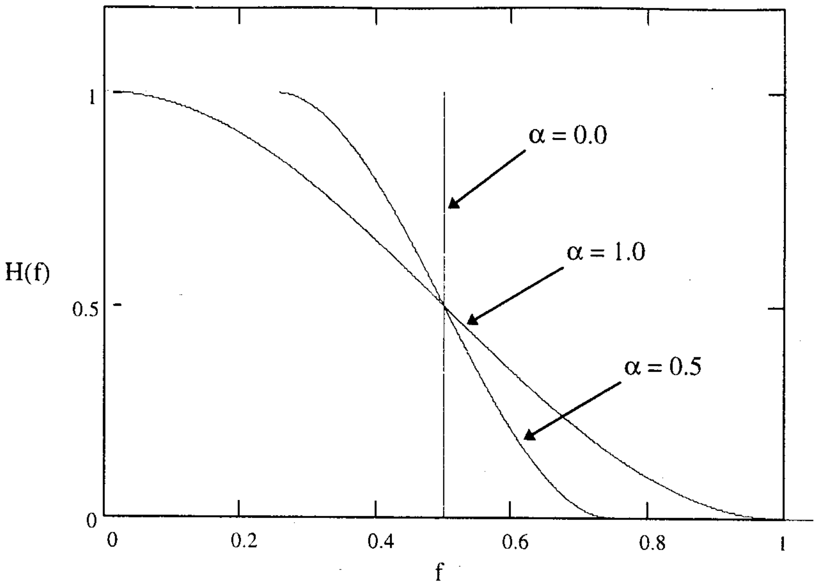


Figure 3.2: Frequency responses for different rolloff factors

The Raised Cosine filter produces zero ISI only when driven by an impulse. If the filter is not driven by an impulse, then the transfer function of the filter must be divided by the Fourier transform of the input signal. For a NRZ (Non-Return Zero) square pulse train, the Fourier transform has a spectrum with a sin x/x shape. The transfer function then becomes

$$H(f) = \begin{cases} \dfrac{\pi \cdot f \cdot T_s}{\sin(\pi \cdot f \cdot T_s)} & \text{for } |f| \leq f_1 \\ \dfrac{\pi \cdot f \cdot T_s}{\sin(\pi \cdot f \cdot T_s)} \cdot \cos^2\left(\dfrac{\pi \cdot (f - f_1)}{4 \cdot (B_0 - f_1)}\right) & \text{for } f_1 < |f| \leq 2 \cdot B_0 - f_1 \\ 0 & \text{for } |f| > 2 \cdot B_0 - f_1 \end{cases} \tag{3.3}$$

where T_S is the symbol period. Note that the raised cosine filter is bandlimited to $\frac{R_S}{2} \cdot (1+\alpha)$, and at f = R_S, x/sin x goes to infinity. Therefore, for this system to work, α must be less than 1.0 [12].

3.3.2 Digital Filters

In this simulation, filters will be needed for the purpose of band limiting signals, producing zero intersymbol interference (ISI), and rejecting out of band noise. Because this simulation is in discrete time, digital filters will be needed to accomplish the above objectives. There are two major types of digital filter design methods: Infinite Impulse Response (IIR), and Finite Impulse Response (FIR). Both have advantages and disadvantages, and neither is best for all situations. The optimal filter design method must be determined by analyzing the requirements and objectives of the application.

The IIR method uses the widely available filter functions from analog filter theory. This method starts with an analog filter transfer function, and then translates this analog function in such a way that makes it suitable for discrete-time systems. Filters designed using the IIR method will be recursive in nature (the output of the filter depends on previous filter outputs, as well as past and current values of the input), and the filter's impulse response will be infinite. IIR filters require fewer coefficients than FIR filters, and have a closed form design technique that does not require iteration. Some of the disadvantages of IIR filters include non-linear phase response, and the use of feedback in the implementation that can cause instability if not carefully implemented.

The FIR method does not rely on analog filter theory. Instead the frequency response of the desired filter is used to determine the digital filter coefficients. This design method is non-recursive in nature, and the impulse response has a finite number of terms. Filters designed using the FIR method are always stable, and have a linear phase. FIR filters need a high number of coefficients to adequately describe the impulse response of the filter. This large number of filter coefficients results in longer processing times, and can be a great disadvantage if used in real time applications. In addition, the design procedure may need to be performed numerous times to find the optimum number of coefficients to meet the requirements of the application.

For this simulation, Square Root Raised Cosine filters will be used on the transmitting and receiving end of the system. These filters are defined by the frequency response in Equation 3.3. Using FIR filter design, the filter can be designed directly from the filter response, whereas in IIR design, an appropriate analog filter containing poles and zeros is needed to begin the design. In addition, Raised Cosine filters need linear phase to achieve zero ISI [12]. Using FIR filters, linear phase can be achieved. The only drawback to the design of Square Root Raised Cosine filter using FIR method is the large number of coefficients needed to achieve an accurate magnitude response. For this simulation FIR filter design method will be used to design the Square Root Raised Cosine filter. The filter coefficients were generated using a program written in Mathcad.

3.3.3 FIR Filter Design of Transmit Filter

For illustration, the design procedure for obtaining the FIR filter coefficients for the transmit filter will be demonstrated below. The procedure for obtaining the receive filter coefficients is exactly the same, the only difference being the filter transfer function.

FIR filter design can either be done using the filter transfer function or the filter impulse response. The square root raised cosine filter is both specified in the frequency domain and in the time domain. The transmit filter needs to be cascaded with x/sin(x) in order to obtain zero ISI for a NRZ input, so it is more convenient to start in the frequency domain. The transfer function for the transmit filter is

$$H(f) = \begin{cases} \dfrac{\pi \cdot f \cdot T_s}{\sin(\pi \cdot f \cdot T_s)} & \text{for } |f| \le f_1 \\[2ex] \dfrac{\pi \cdot f \cdot T_s}{\sin(\pi \cdot f \cdot T_s)} \cdot \cos\left(\dfrac{\pi \cdot (f - f_1)}{4 \cdot (B_0 - f_1)}\right) & \text{for } f_1 < |f| \le 2 \cdot B_0 - f_1 \\[2ex] 0 & \text{for } |f| > 2 \cdot B_0 - f_1 \end{cases} \qquad (3.4)$$

and is shown in Figure 3.3. The impulse response is obtained by taking the inverse fast Fourier transform of the filter transfer function. This analog transfer function first needs to be sampled. The signals in the simulation have a symbol rate R_S, and symbol period of T_S. In the time domain, the signal waveform is sampled at 16 samples/symbol ($N_{SS} =$ 16). The sampling frequency is given by $f_S = R_S \cdot N_{SS}$ Hz, and the spacing between

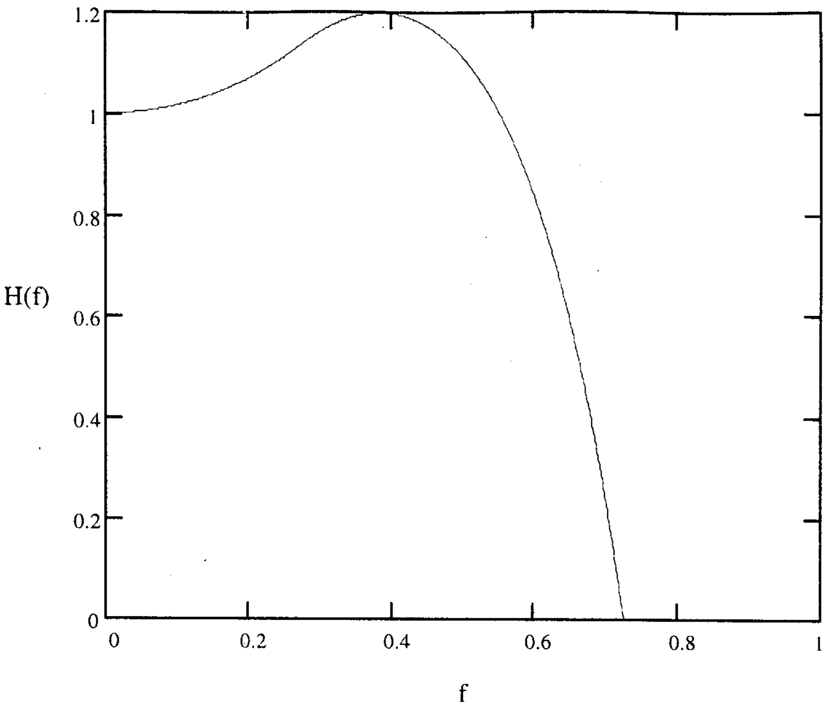


Figure 3.3: Square root raised cosine filter with x/sin(x) equalization and rolloff factor $\alpha = 0.45$ ($R_s = 1$)(normalized frequency)

samples is $\Delta t = \frac{T_s}{N_{ss}}$ seconds. In the frequency domain, the filter response is periodic in f_s (see Figure 3.4). The total frequency span is $f_s \cdot N_{ss}$, and for a FFT of length N, the frequency spacing is $\Delta f = \frac{f_s \cdot N_{ss}}{N}$. After the filter transfer function has been sampled with frequency spacing Δf, the impulse response of the filter is obtained by taking the inverse FFT (IFFT). The impulse response of the square root raised cosine filter is shown in Figure 3.5. The impulse response of this filter is infinitely long. Truncation of the impulse response will allow us to have a finite number of coefficients in our FIR filter. The impulse response should be truncated at a point where the response has sufficiently died down. In this case we will chose the number of one-sided coefficients (M) to be 100.

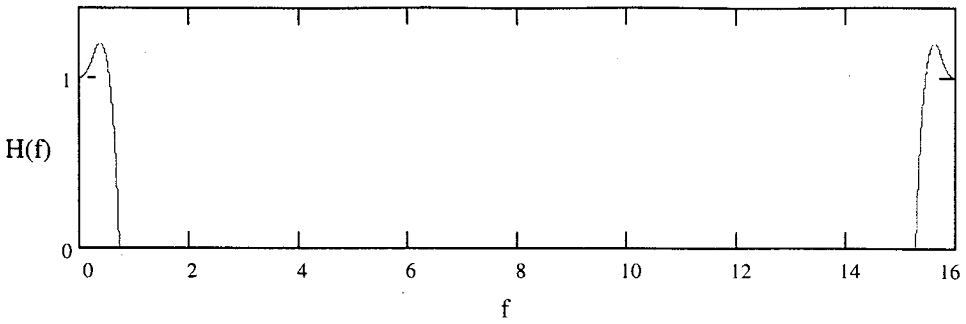


Figure 3.4: Transmit filter transfer function (normalized frequency)

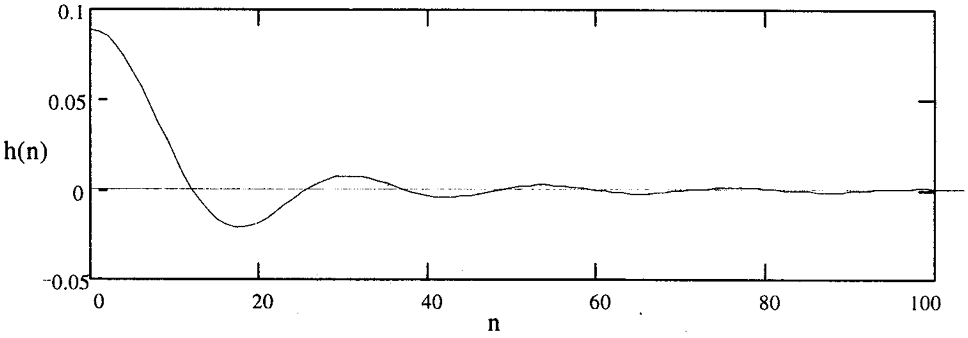


Figure 3.5: Filter impulse response

The impulse response is actually two sided (see Figure 3.6). Because there is filter output before t = 0, the filter is non-causal. In order to correct this, a delay will be introduced to make it causal. This is done by shifting the impulse response M coefficients to the right (see Figure 3.7). In order to improve the design of this filter, windowing techniques will be used. Instead of abruptly cutting off the coefficients at ±M, window functions smoothly reduce the filter coefficients to zero. For this filter, a Hamming window is used to accomplish this (See Figure 3.8). The filter coefficients are 'windowed' by the equation:

$$h(n) = h_{ideal} \cdot w(n) \quad n = 0,1 \ldots 2 \cdot M$$

where the window function for a Hamming window is given by:

$$w(n) = 0.54 - 0.46 \cdot \cos(\frac{\pi \cdot n}{M})$$

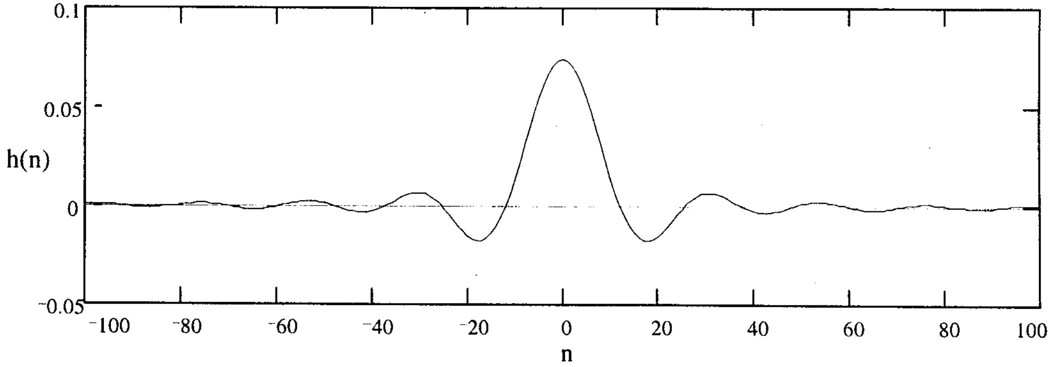


Figure 3.6: Non-causal impulse response

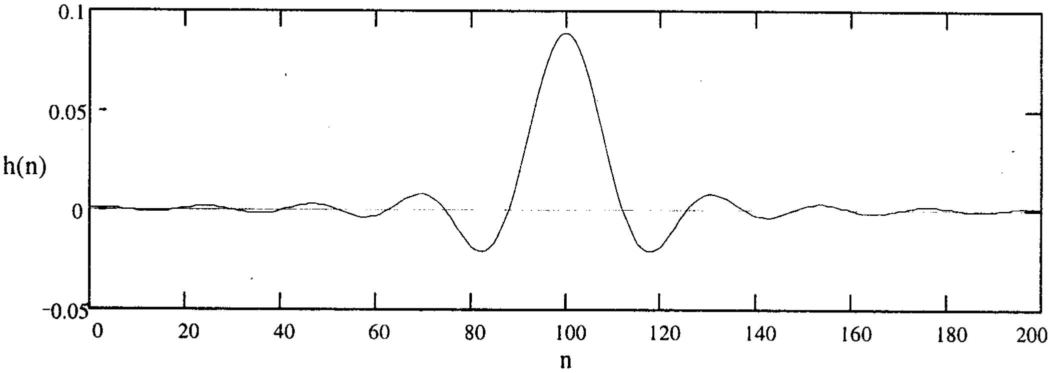


Figure 3.7: Causal impulse response

The resulting filter coefficients are used to filter a signal x(n) using the following equation:

$$y(n) = \sum_{k=0}^{2*M} h(k) \cdot x(n-k)$$

The output y(n) of the filter is dependent on the current input x(n), and the $2 \cdot M$ previous inputs. There is no feedback involved, so the filter is always stable. The magnitude response of the filter as compared to the original filter is given in Figure 3.9.

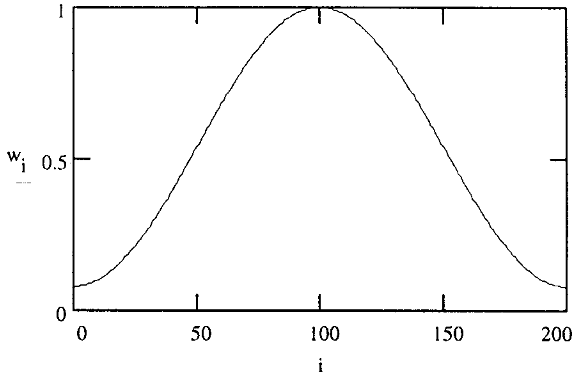


Figure 3.8: Hamming window function

By using more filter coefficients, the response of the FIR filter will be closer to the original transfer function of the filter.

3.4 Adding noise

In order to make the simulation as realistic as possible, the channel model is chosen to be the Additive Gaussian White Noise (AWGN) channel. The noise signal can be generated by producing a Gaussian, or normal sequence with standard deviation σ using (3.2).

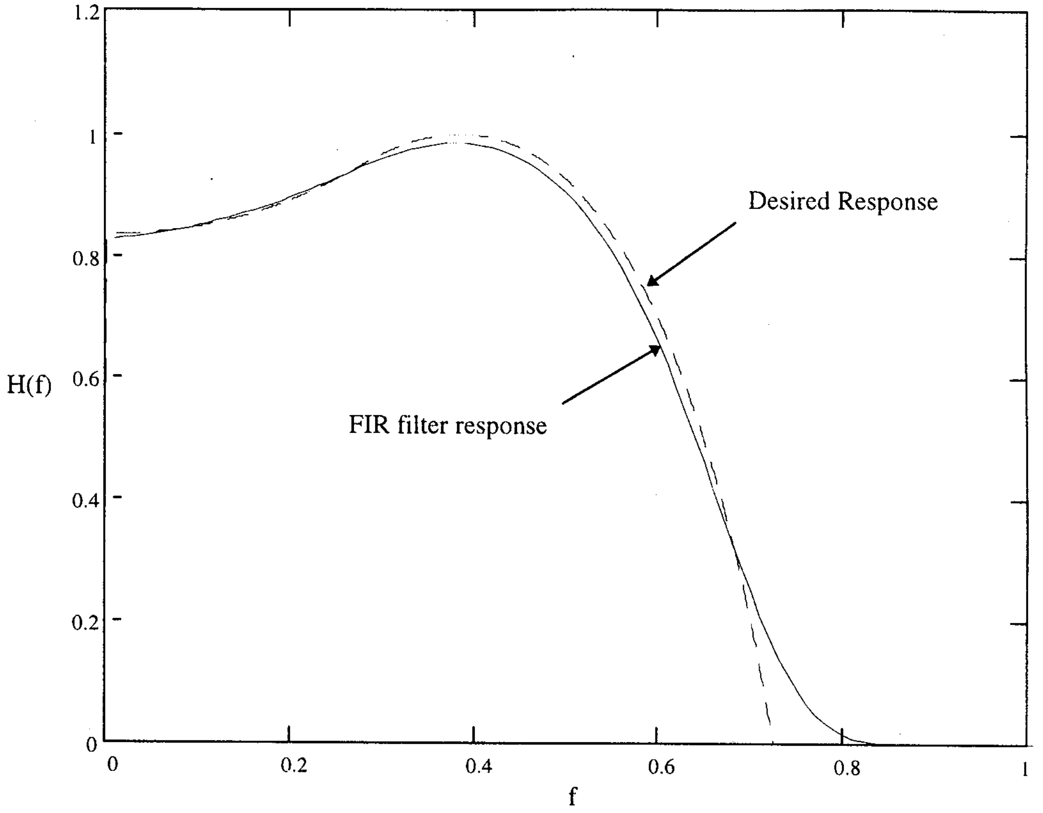


Figure 3.9: FIR frequency response vs. analog filter response for a square root Raised Cosine filter with x/sin(x) equalization

3.4.1 Noise Equivalent Bandwidth

The noise equivalent bandwidth B_N will have to be calculated in order to add the correct amount of noise to the simulation. Consider the lowpass filter in Figure 3.10 a) with transfer function H(f).

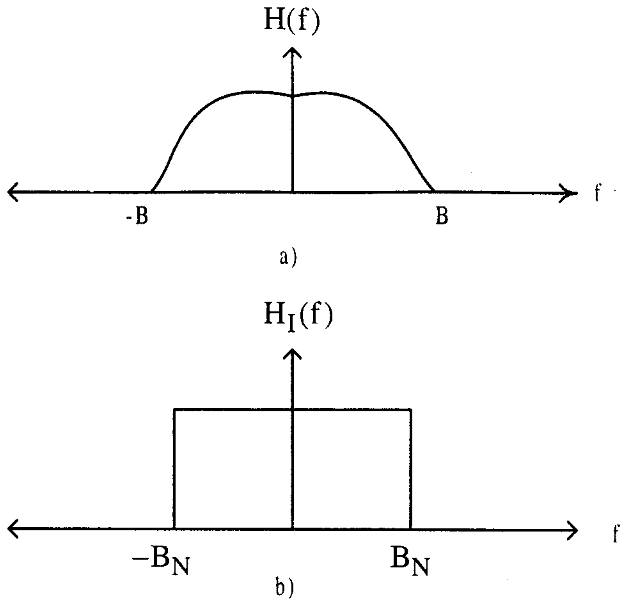


Figure 3.10 (a) Receive filter (b) Noise equivalent bandwidth filter

If white noise with a power spectral density $\frac{N_0}{2}$ is applied at the input to the filter, the total noise power at the output is

$$P = \int_{-\infty}^{\infty} \frac{N_0}{2} \cdot |H(f)|^2 \, df \qquad (3.4)$$

$$= N_0 \int_{0}^{\infty} |H(f)|^2 \, df \qquad (3.5)$$

3.4.2 Calculating the noise variance

Now consider an ideal filter $H_I(f)$ with single sided bandwidth B_N as in Figure 3.10 b). If the same white noise signal is applied to the input of the ideal filter, the total noise power at the output of the filter is

$$P_{ideal} = \int_{-B_N}^{B_N} \frac{N_0}{2} \cdot |H_I(f)|^2 \, df \tag{3.6}$$

$$= N_0 B_N |H(0)|^2 \tag{3.7}$$

The bandwidth of the ideal filter can be chosen so that the total noise power at the output of the ideal filter is equal to the total noise power at the output of the real filter. Equating (3.5) and (3.7) and solving for B_N yields

$$B_N = \frac{\int_0^\infty |H(f)|^2 df}{|H(0)|^2} \tag{3.8}$$

White noise has a constant PSD for all frequencies

$$S_{NN}(f) = \frac{N_0}{2} \quad \text{for} \; -\infty \leq f \leq \infty$$

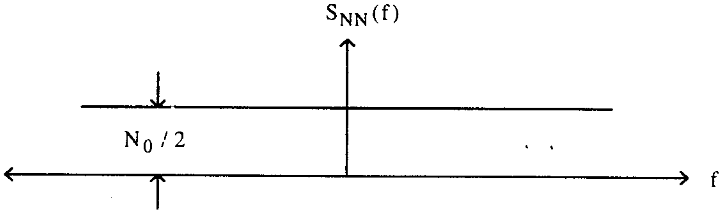
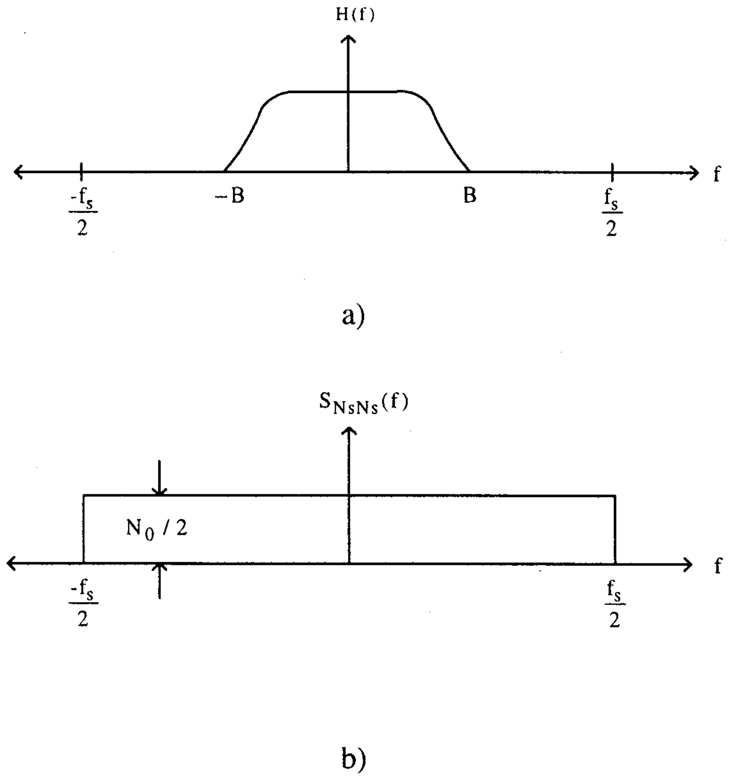


Figure 3.11: Power spectral density of AWGN

Unfortunately, this signal take an infinite amount of power to produce. For simulation purposes, we will be working with a system that has a finite bandwidth. The receive filter has a bandwidth B. The sampling frequency f_s is chosen to be greater than 2B. If we use bandlimmited white Gaussian noise with a constant PSD over the simulation bandwidth,

$$S_{N_sN_s}(f) = \frac{N_0}{2} \quad \text{for} \quad -\frac{f_s}{2} \leq f \leq \frac{f_s}{2}$$

the response of the system will be the same whether $S_{NN}(f)$ or $S_{N_sN_s}(f)$ is used [5].



a)

b)

Figure 3.12: a) Filter transfer function b) Bandlimited AWGN before filtering

The total power for the bandlimited AWGN is equal to

$$\sigma_x{}^2 = \frac{N_0 f_s}{2}. \tag{3.9}$$

After AWGN is passed through a filter with noise equivalent bandwidth B_N, the total noise power is equal to

$$\sigma_y{}^2 = \frac{N_0}{2} \cdot 2 \cdot B_N = N_0 B_N = \frac{P}{SNR} \tag{3.10}$$

where P is the total signal power and SNR is the signal to noise ratio. Combining (3.9) and (3.10) and solving for σ_x^2

$$\sigma_x^2 = \frac{P}{SNR} \cdot \frac{f_s}{2 \cdot B_N} \tag{3.11}$$

σ_x^2 is the total noise power that is to be added to the channel before the Rx filter. The noise can be generated by using (3.2) to create a sequence of Gaussian random variables with $\mu = 0$ and $\sigma = \sigma_x$.

Sometimes it is convenient to express the probability of a bit error P_e as a function of $\rho = \frac{E_b}{N_0}$ where E_b is the bit energy and N_0 is the noise density. The noise variance will be expressed differently than (3.11).

Let $\rho = \frac{E_b}{N_0}$. In terms of N_0

$$N_0 = \frac{E_b}{\rho} \tag{3.12}$$

The signal power P is equal to

$$P = \frac{E_s}{T_s} = \frac{E_b \cdot k}{T_s} \tag{3.13}$$

where k is the number of bits per symbol. From (3.13), the energy per bit is

$$E_b = \frac{P \cdot T_s}{k} \tag{3.14}$$

The total noise power before filtering is equal to $\sigma_x^2 = \frac{N_0 f_s}{2}$ from Figure 3 b). Combining this with (3.12) and (3.14) yields

$$\sigma_x{}^2 = \frac{P \cdot T_s \cdot f_s}{2 \cdot \rho \cdot k} \tag{3.15}$$

The sampling frequency is defined as

$$f_s = R_s \cdot N_{ss} \tag{3.16}$$

so (3.16) can be rewritten as

$$\sigma_x{}^2 = \frac{P \cdot N_{ss}}{2 \cdot \rho \cdot k} \tag{3.17}$$

The noise can be generated by using (3.2) to create a sequence of Gaussian random variables with $\mu = 0$ and $\sigma = \sigma_x$.

3.4.3 Ideal Channel model

An ideal channel can be used to obtain results that are not degraded by the filters. There is no need to simulate the analog signal, and modulate, filter, and add noise. The binary data can have Gaussian noise added directly to the data bits, and then these bits can either be soft or hard decoded. For each binary bit u_j, the noise corrupted bit can be obtained by using (3.2) with $\mu = 1$ for $u_j = 1$, and $\mu = -1$ for $u_j = 0$. The variance of the noise is

$$\sigma^2 = \frac{1}{2\,\rho}$$

where ρ is the signal to noise ratio. For hard decisions v_j, if the noise corrupted u_j is greater than zero, then $v_j = 1$. Otherwise, $v_j = 0$. Using this method, ideal BPSK is obtained.

3.5 Simulation Results

The simulation was run without coding systems to verify the simulation was working properly. The results were compared to the theoretical BPSK results, which are given by

$$P_e = \frac{1}{2} \cdot \text{erfc}\left(\sqrt{\frac{E_b}{N_0}}\right)$$

where E_b is the energy per bit, N_0 is the single sided noise power spectral density, and

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \cdot \int_x^\infty e^{-u^2/2}\, du\,.$$

The results of the simulation are shown in Figure 3.13. Looking at the results of the simulation, two observations can be made. First, for greater values of E_b / N_0, the simulation results deviate from the theoretical curve. This can be attributed to the effects of filtering. Second, for higher values of rolloff, the performance of the simulation becomes closer to the theoretical. This can be attributed to the increase in bandwidth which comes from an increase in the rolloff value.

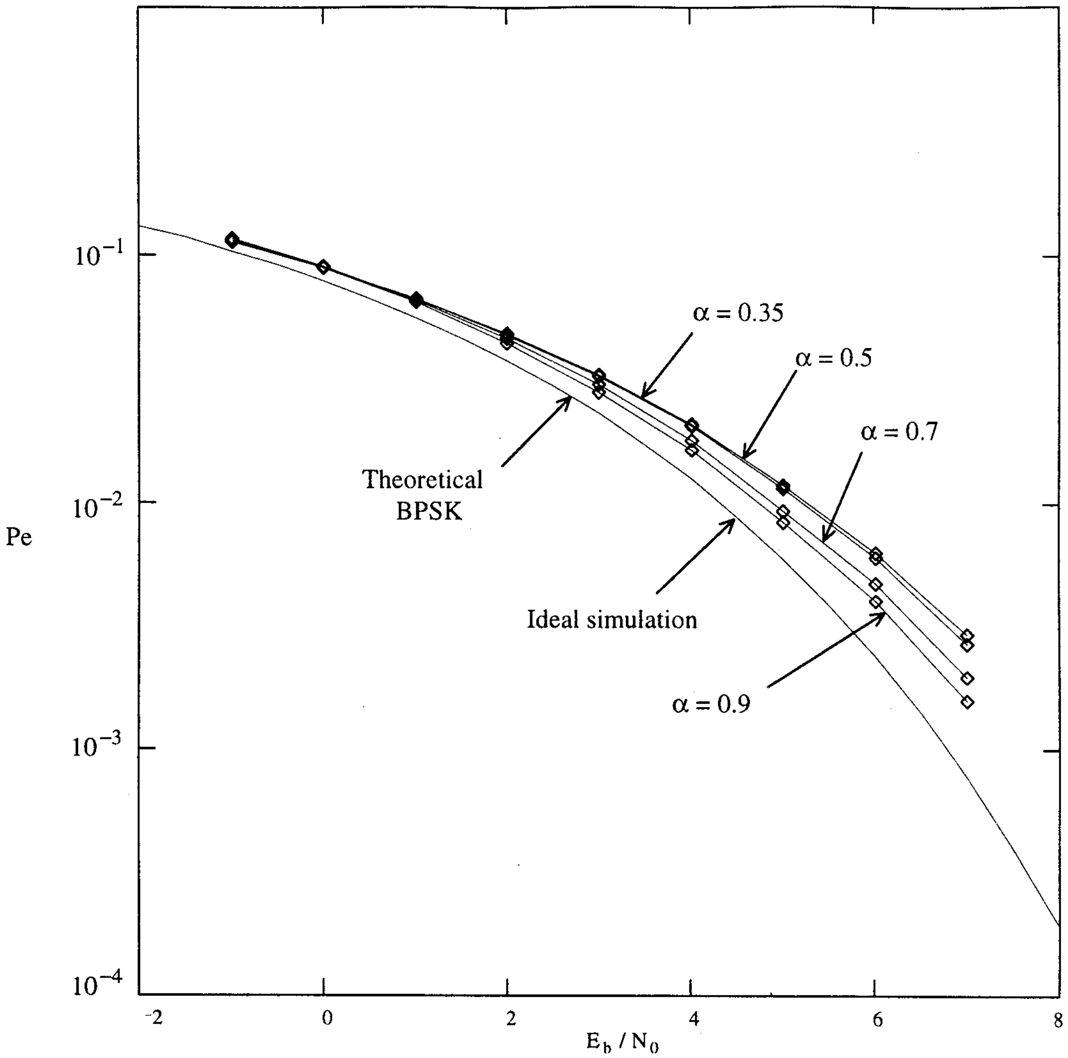


Figure 3.13 Results of the BPSK simulation for different values of filter rolloff.

Chapter 4

Erasure Methods and Simulation Results

The purpose of this chapter is to investigate the performance of the concatenated system given in Figure 1.1, and to discuss and propose methods of declaring RS symbol erasures to improve performance. It is the goal of these methods to recognize symbols that are in error and erase them, thus utilizing the full capacity of the errors and erasures Reed Solomon decoder. The success of these methods is highly dependent on being able to successfully identify the errors in each Reed Solomon codeword (RSW). The performance of the concatenated system without erasures was tested. In addition, a basic method for erasure declaration using the reliability information provided by the SOVA was implemented. This method is based on [14]. Two new procedures for declaring erasures are proposed. Both use the reliability information from the SOVA in addition to information provided by successfully decoded RSW in the deinterleaving frame. The results of these two methods are presented and compared to Paaske's [7] method.

4.1 Concatenated system simulation results

The performance of the concatenated system presented in Figure 1.1 was investigated. More specifically, the concatenated system used by NASA for deep space communication has been simu!ated. This system uses a rate 1/2, K = 7 convolutional code with Viterbi decoding as the inner code and a (255, 223) Reed Solomon code as the outer code. This coding system was tested using a real system containing square root raised cosine FIR filters using a rolloff value of $\alpha = 0.5$, and an ideal system. The performance evaluation studied the effects of using no interleaving, and interleaving depths I = 2, 6, 8. In addition, the effect of increasing the truncation lengths from $\delta = 32$ to $\delta = 100$, and soft decision levels from L = 8 to L = 64 were simulated.

Five systems have been developed to study the effects of varying the truncation length and number of soft decision levels, and are presented in Table 4.1. These systems will be simulated for no interleaving, and for interleaving depths I = 6 and 8. For the following results, the gain is measured at a bit error rate of 10^{-5}.

Table 4.1 Various systems simulated

	Real/Ideal	Truncation length δ	Soft decision levels L
System 1	Real	32	8
System 2	Ideal	32	8
System 3	Ideal	32	64
System 4	Ideal	100	8
System 5	Ideal	100	64

Figure 4.1 contains the simulation results for the concatenated coding system with no interleaving used. It is shown that the real system with $\delta = 32$ and 8 level soft decision

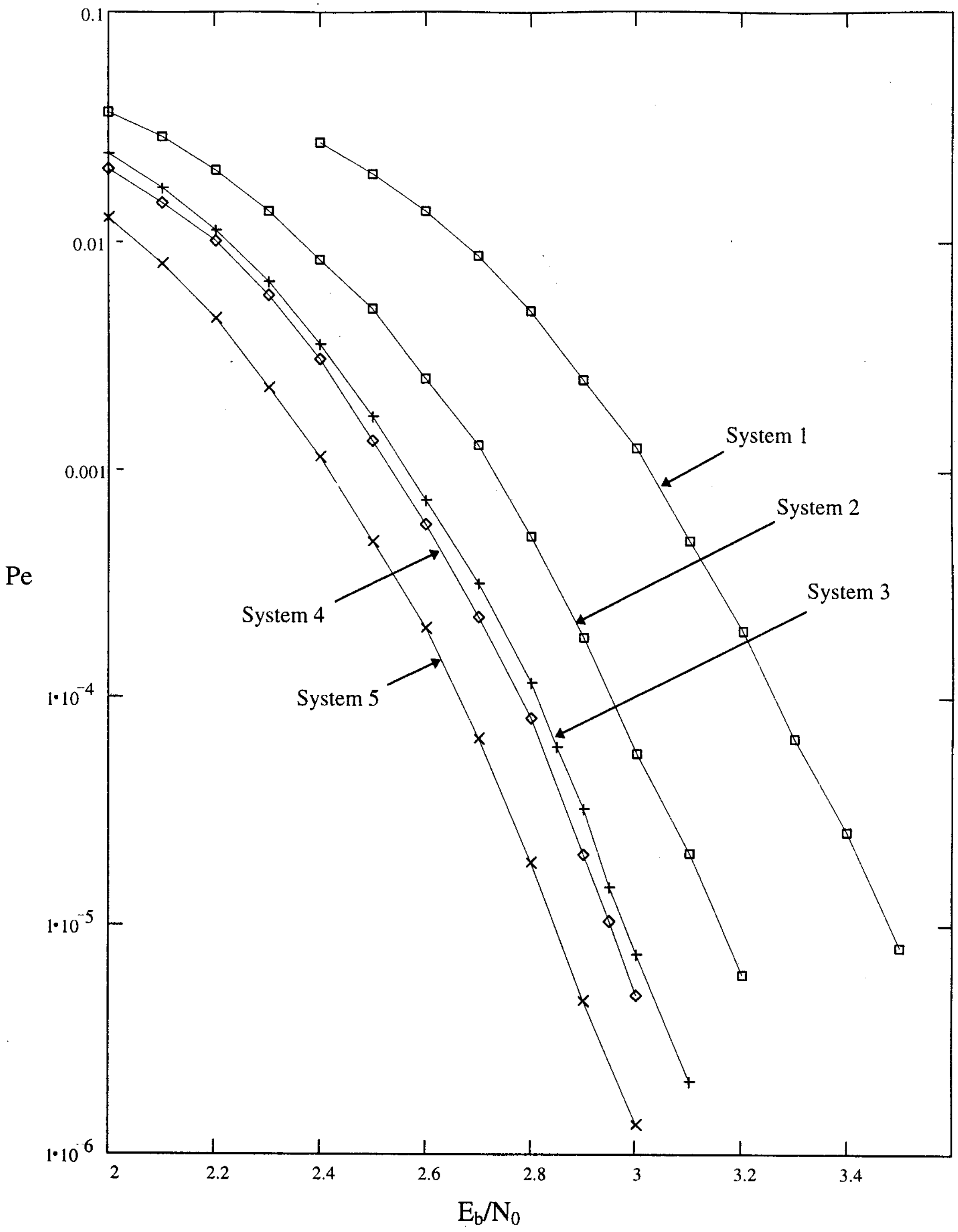


Figure 4.1 Simulation results for the concatenated system with no interleaver

Viterbi decoding (system 1) results in a loss of about 0.3 dB when compared with the ideal system with the same truncation length and soft decision levels (system 2). There is approximately a gain of 0.2 dB when the decision depth δ is increased from $\delta = 32$ to $\delta = 100$. This can be illustrated in Figure 4.1 by comparing the difference in performance between system 2 ($\delta = 32$, L = 8) and system 4 ($\delta = 100$, L = 8). Likewise, there is also a 0.2 dB gain when the decision depth in system 3 is increased from $\delta = 32$ to $\delta = 100$ to form system 5. When the number of soft decision levels is increased from L = 8 in system 2 to L = 64 in system 4, the result is about a 0.15 dB gain. Likewise, the increase in soft decision levels from system 3 to system 5 results in roughly 0.15 dB gain. The combined use of $\delta = 100$ and L = 64 results in about 0.35 dB gain over the ideal system using $\delta = 32$ and L = 8. These gains are consistent for interleaving depths I = 6 (Figure 4.2) and I = 8 (Figure 4.3).

The benefits of using an interleaver is demonstrated in Figure 4.4, where the simulation results for system 5 using no interleaving, and depths I = 2, 6, and 8 are presented. It has been shown that the use of interleaving depth I = 2 can provide approximately 0.2 dB gain over the system using no interleaving. When the interleaving depth is increased from I = 2 to I = 6, there is an additional gain of slightly more than 0.2 dB, for a total of roughly 0.4 dB over the system using no interleaving. The use of an interleaver with depth I = 8 provided a minor gain over the system using I = 6. The total gain of using I = 8 when compared to the non-interleaved system is approximately 0.45 dB. It should also be noted that the concatenated coding system using interleaving depths greater than 8 resulted in little or no improvement over the system using depth I = 8.

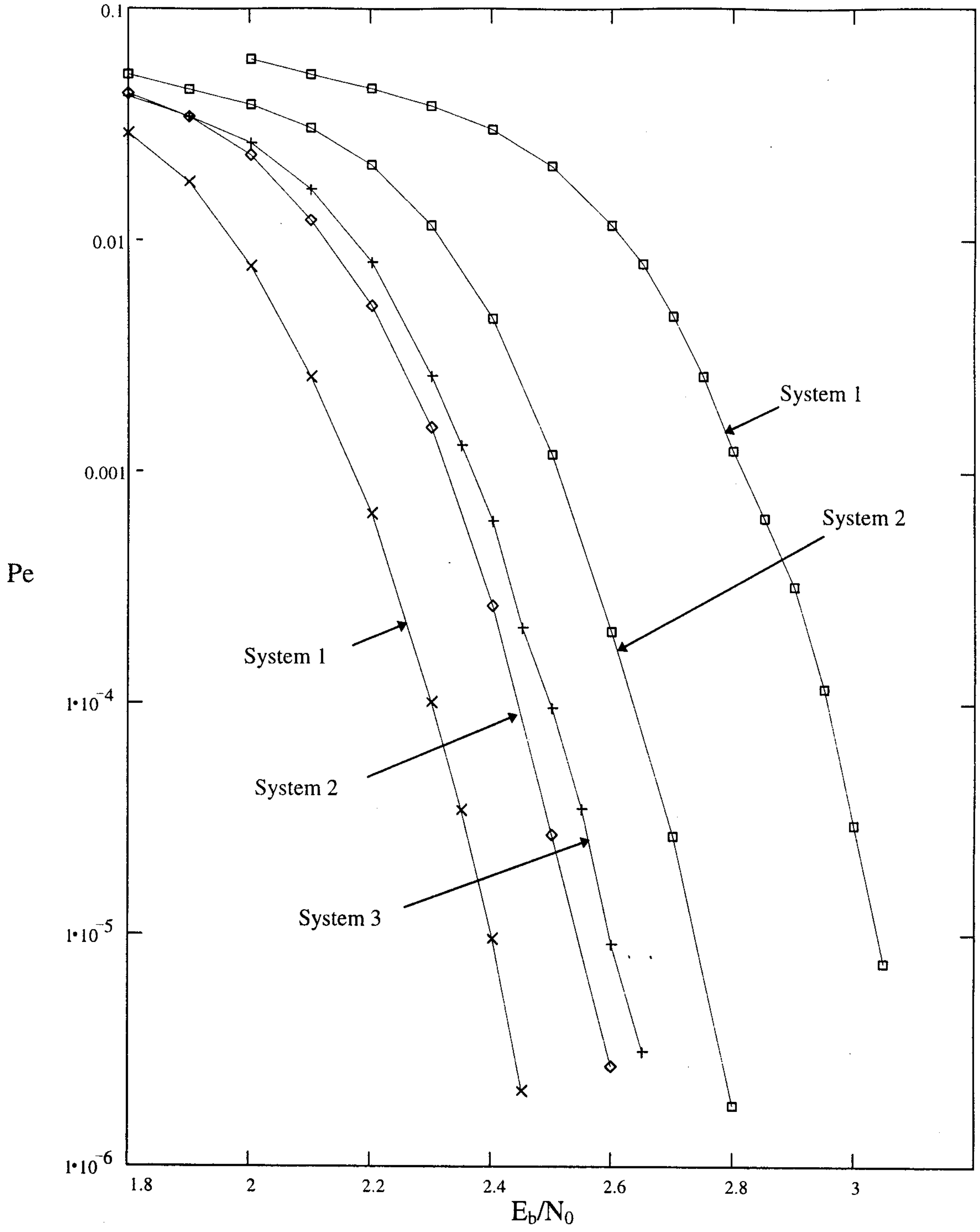


Figure 4.2 Simulation results for the concatenated system with interleaver depth I = 6

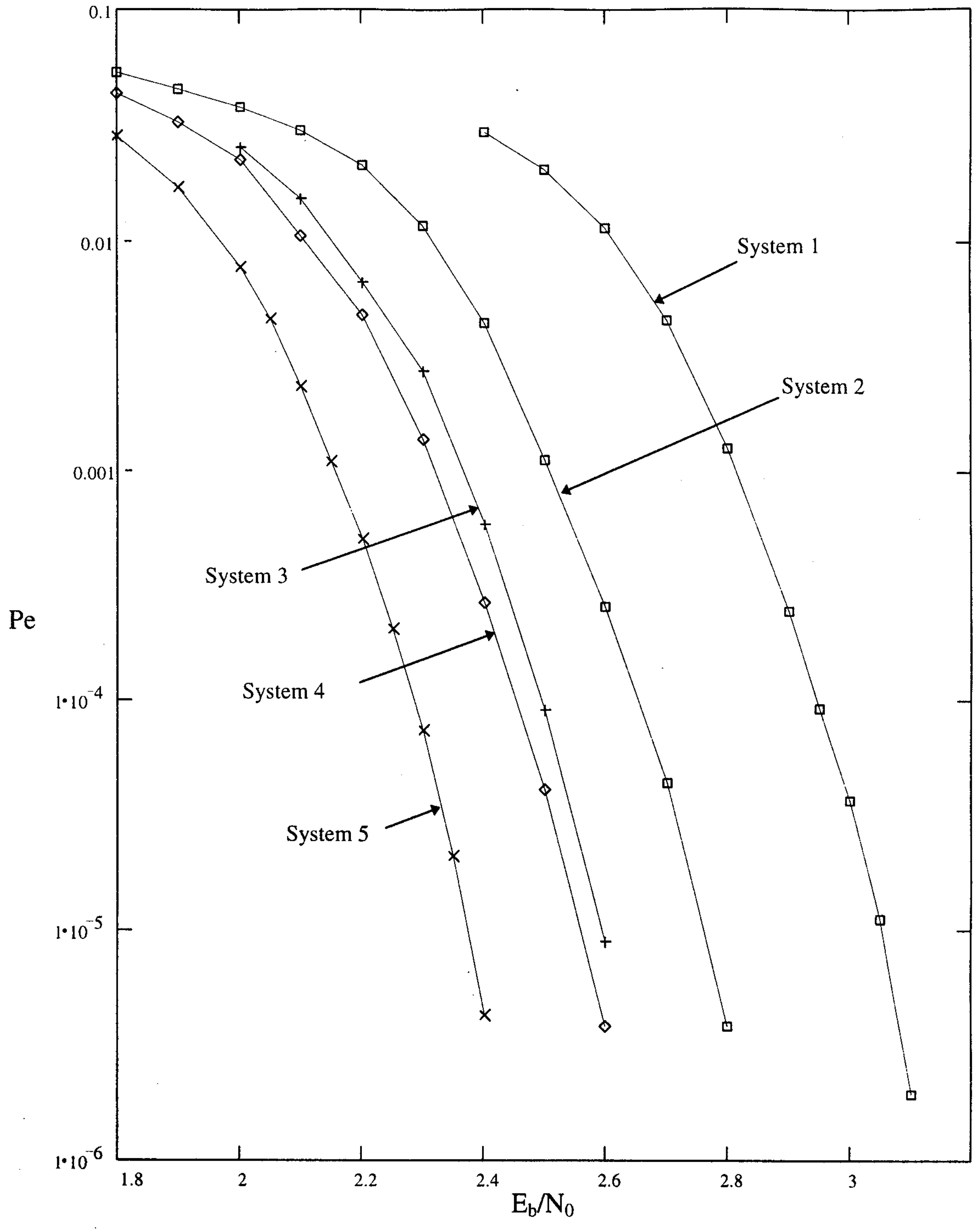


Figure 4.3 Simulation results for the concatenated system with interleaver depth I = 8

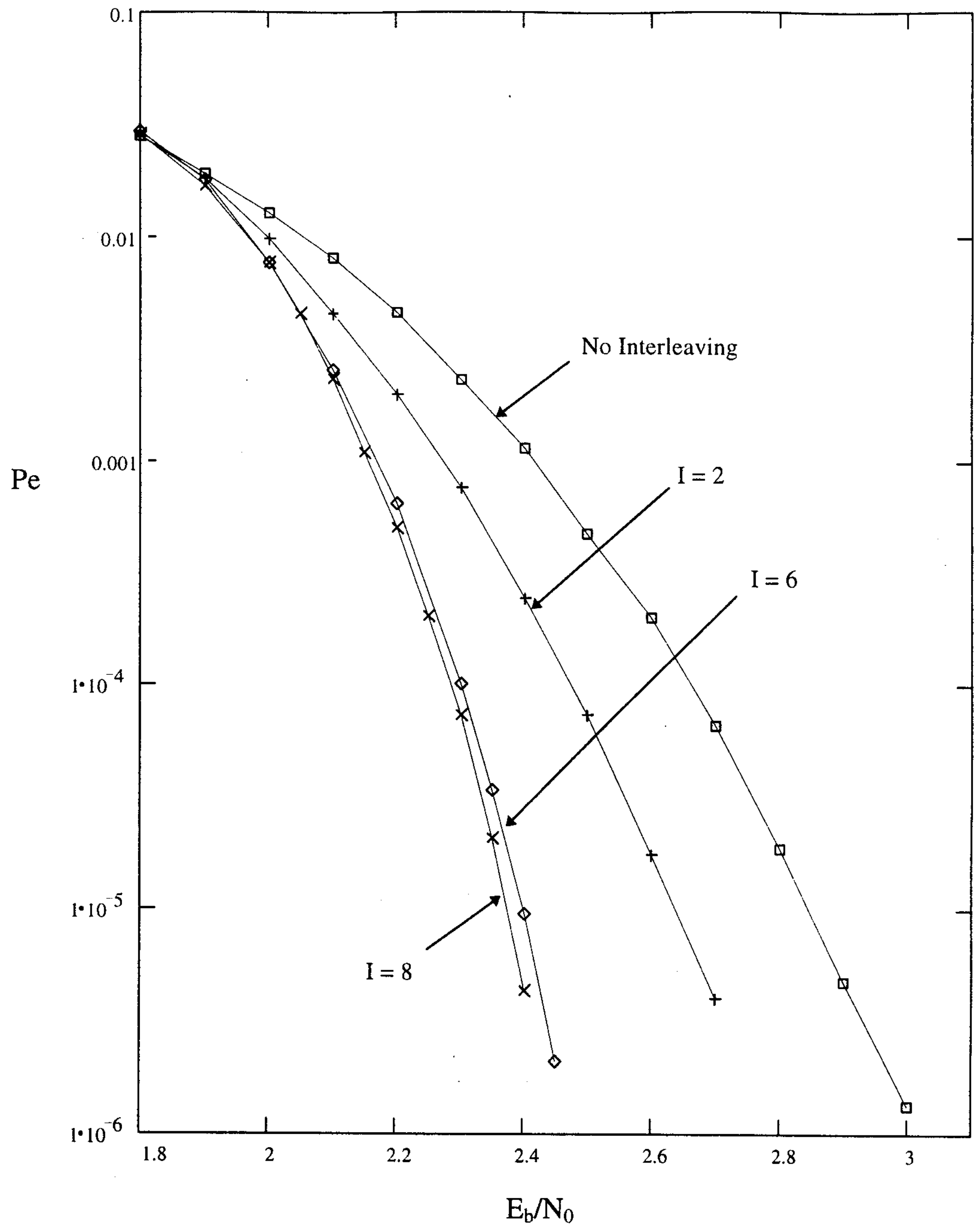


Figure 4.4 Simulation results for system 5 with various interleaving depths

As was expected, gains can be obtained in the concatenated coding system by increasing the truncation length and the number of soft decision levels used. The use of an interleaver between the convolutional code and the Reed Solomon code also provided gain over the system using no interleaving. The gains obtained through these improvements do not require any additional coding or bandwidth expansion, only additional hardware size and complexity.

Another way gain can be obtained without the use of additional coding or bandwidth expansion is through the use of erasures. A Reed Solomon code with minimum distance d_{min} can correct v errors and ρ erasures as long as the inequality $d_{min} \leq 2v + \rho$ is satisfied. Clearly, if errors are transformed into erasures, performance can be improved. Because the error positions are not known ahead of the decoding process, it is necessary to find methods that can identify unreliable Reed Solomon symbols. Once these symbols are identified, they can be erased. There is the possibility that correct symbols may be erased, however. For the performance to be improved, the erasure method must erase more errors than correct symbols.

For the concatenated system in Figure 1.1, the Viterbi decoder produces hard output $u_j \in \{0, 1\}$. It has been shown in section 2.3.3 that the soft output Viterbi algorithm produces a reliability value L_j associated with each outgoing bit u_j. This reliability information can give an indication to which Reed Solomon symbols are in error. In the first method investigated, bit reliabilities are transformed into RS symbol reliabilities. A table of the least reliable symbols for each RS codeword can then be compiled, and the least reliable symbols can be erased. The first method for declaring erasures in presented in greater detail below.

4.2 Erasure Method 1

The first method for declaring RS symbol erasures is based on the method found in [14]. The SOVA is used to obtain the reliabilities of each output bit from the Viterbi decoder. The output of the SOVA is quantized using q+1 bits, where q bits represent the magnitude of the decision (the reliability), and 1 bit represents the hard decision. In general, these reliabilities are real valued, and have a range between 0 and infinity. It is not necessary to quantize the reliability values between 0 to infinity, but rather between 0 and some value L_{max}. The reliability values produced by the SOVA will be assigned one of 2^q levels between 0 and L_{max}. If a reliability is greater than L_{max}, then the reliability level is set to 2^q (i.e. the maximum level). For this simulation q = 8 and L_{max} = 8 were used. From here on out, the "reliability value" L_j for a bit will refer to the q-bit quantized level.

Before declaring RS symbol erasures, the bit reliabilities from the output of the SOVA need to be converted to RS symbol reliabilities. This can be accomplished by simply using the minimum bit reliability in a symbol as the symbol reliability. This can be rationalized as follows. If a bit has a small reliability, and is in error, the symbol will be in error also. Therefore, the best information contained in the symbol reliability is contained in the minimum bit reliability. For each RS codeword (RSW), a reliability table (RT) is formed. The RT has m_e positions, where m_e is the maximum number of erasures allowed. If the least reliable symbols in the RSW occur at positions k_j, where j = 1, 2, ..., m_e, then RT(j) = k_j. RT(1) contains the least reliable position, RT(2) contains the 2nd least reliable symbol, etc. The RS decoder will first attempt to decode without declaring erasures. If this is a successful decoding, stop. If not, the two symbols with the smallest reliabilities are erased (RT(1) and RT(2)), and decoding is attempted again. If successful, decoding stops. If unsuccessful, the four least likely symbols are erased (RT(1) through RT(4)) and decoding is attempted again. For each unsuccessful decoding, two more erasures are declared until either a successful decoding is achieved, or the maximum number of erasures m_e is reached. This maximum number is chosen to be 16 erasures. Using more than 16 erasures gave poorer results due to the fact that the symbols reliabilities at the output of the SOVA hit more correct symbols than errors. In addition, the probability of a decoding error is also increased. A decoding error occurs when a codeword contains more than t errors, and fails to notice that it does. The decoder claims that there are less than t errors. This is different than a decoding failure. A decoding failure happens when there are more than t errors, and the decoder detects that it contains more than t errors. For a t error correcting RS code, the probability of a decoding error is 1/t! [7, 14, 19]. The procedure for erasure decoding using Method 1 is presented below.

4.2.1 Procedure for erasure decoding using Method 1

The decoding of each RS codeword involves the following steps:

1. Set the number of erasures $n_e = 0$.

2. Attempt to decode the RS codeword using errors only decoding. If successful, stop.

3. Set $n_e = n_e + 2$.

4. Erase the n_e least reliable symbols as determined by the reliability table, and attempt error and erasures decoding. If successful, stop.

5. If $n_e = m_e$ (the maximum number of erasures allowed), then stop.

6. Go to step 3.

7. Stop.

4.2.2 Simulation Results for Method 1

The results of the simulation for Method 1 are presented in Figures 4.5 through 4.7. These figures contain the results using system 1, 2, and 5 for no interleaving (Figure 4.5), and interleaving depths I = 6 (Figure 4.6) and I = 8 (Figure 4.7). The results of the corresponding concatenated code using no erasures is also presented for comparison purposes. From these figures, it is evident that the use of erasure Method 1 results in approximately 0.1 dB gain over the non-erasure concatenated system. This result was consistent regardless of the truncation length, number of soft decision levels, and interleaving depth used by the concatenated system. It should be noted that the gains obtained by using different interleaving depths with erasure Method 1 were identical to the gains obtained when using different interleaving depths in the standard concatenated system. In addition, it should also be noted that increasing the decision depth and number of soft decision levels for the systems simulated using erasure Method 1 were consistent with the results for the concatenated system presented in section 4.1.

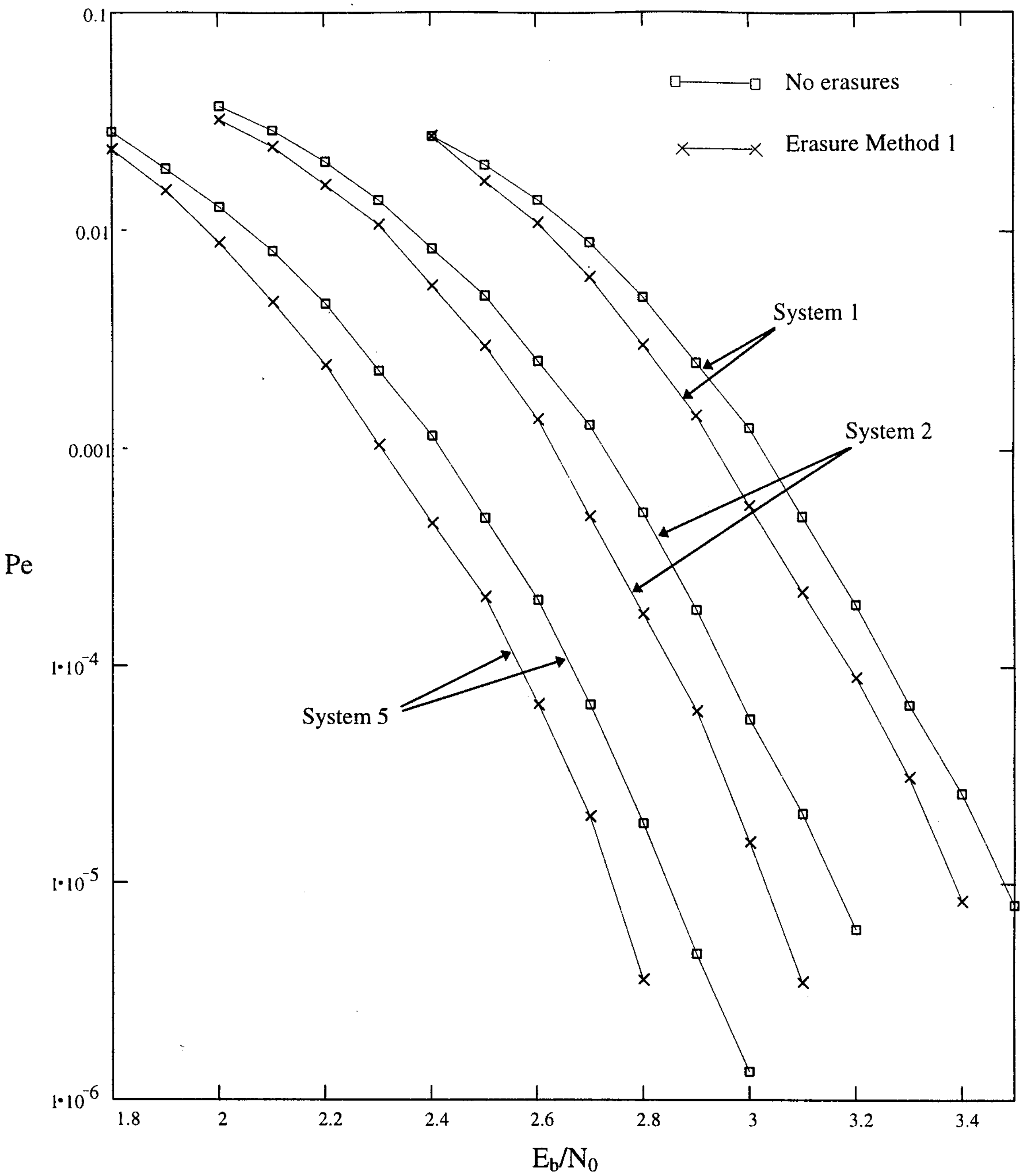


Figure 4.5 Simulation results for Method 1 with no interleaver

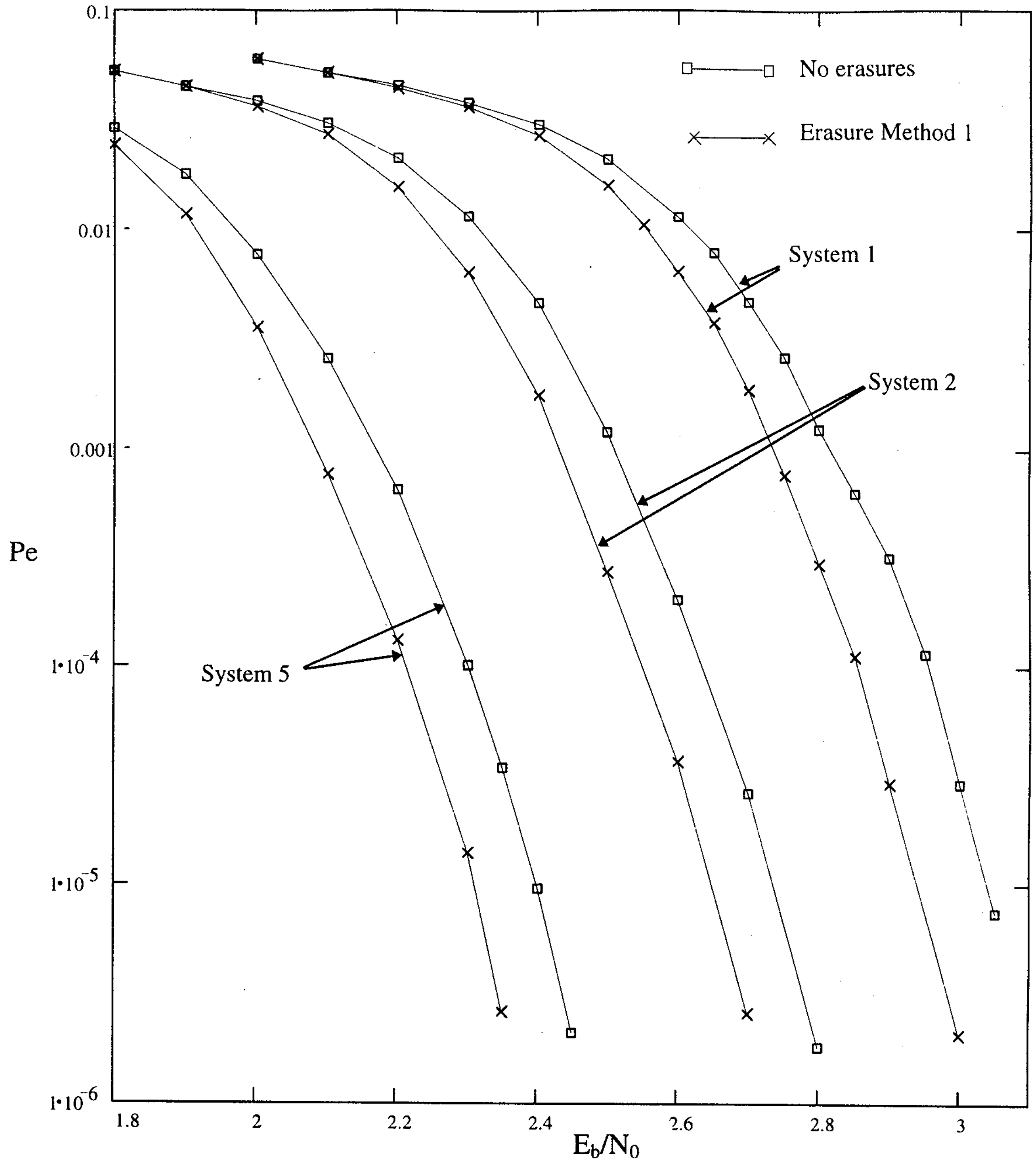


Figure 4.6 Simulation results for Method 1 with interleaver depth I = 6

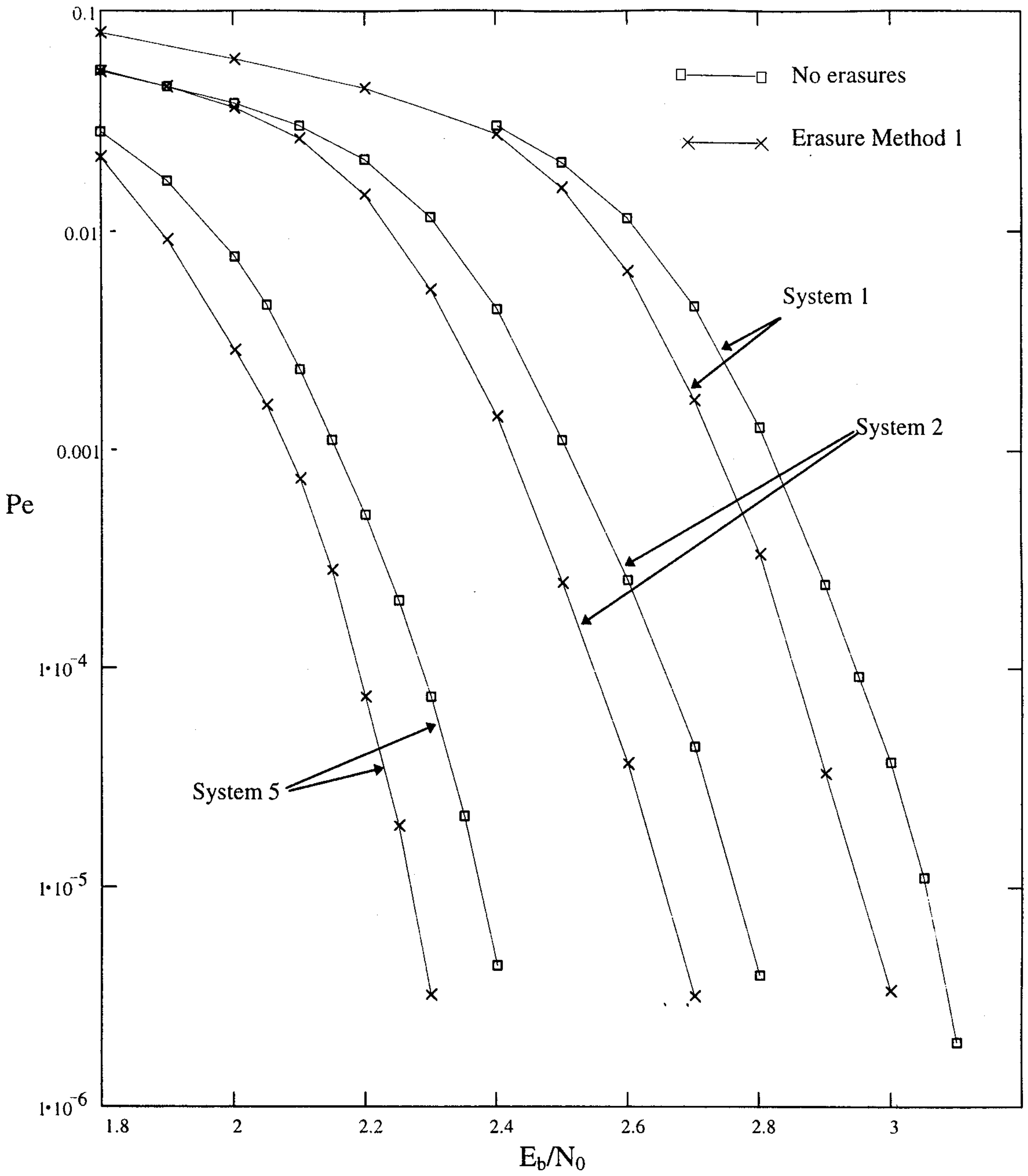


Figure 4.7 Simulation results for Method 1 with interleaver depth I = 8

It has been shown that the use of erasure Method 1 results in approximately 0.1 dB gain over the concatenated system using no erasures. The reason that the performance is not greatly increased is because the erasure declaring process is not optimal. In general, when erasing two symbols at a time, gain is only achieved when both erased symbols are errors (GE). If only 1 erasure is a GE, the capacity of the code stays the same, and if neither erasure is a GE, then capacity of the code actually decreases. Method 1 uses the reliability information generated from the SOVA, and the symbols with the smallest reliabilities are systematically erased. A symbol with a small reliability does not guarantee a GE. For example, the smallest reliability value ($L_j = 0$) occurs when there is a metric difference $\Delta = 0$. Because the path metrics are equally likely, the probability the wrong path has been chosen is 0.5. This is why a small reliability value does not guarantee a wrong path has been taken. The reliability table used in Method 1 can be modified to give more reliable information. This can be accomplished by using decoded Reed Solomon codewords in the deinterleaving frame to provide information to the non-decoded Reed Solomon codewords. This information can be used to construct a reliability table with more accurate information, and then the unreliable symbols can be converted into RS symbol erasures. The second method for declaring erasures is presented below.

4.3 Erasure Method 2

It has been found that the burst errors at the output of the SOVA contain the same reliabilities within the burst. Consider the two paths merging in state S_k at time t in Figure 4.8. The survivor path has metric M_1 and the competing path has metric M_2.

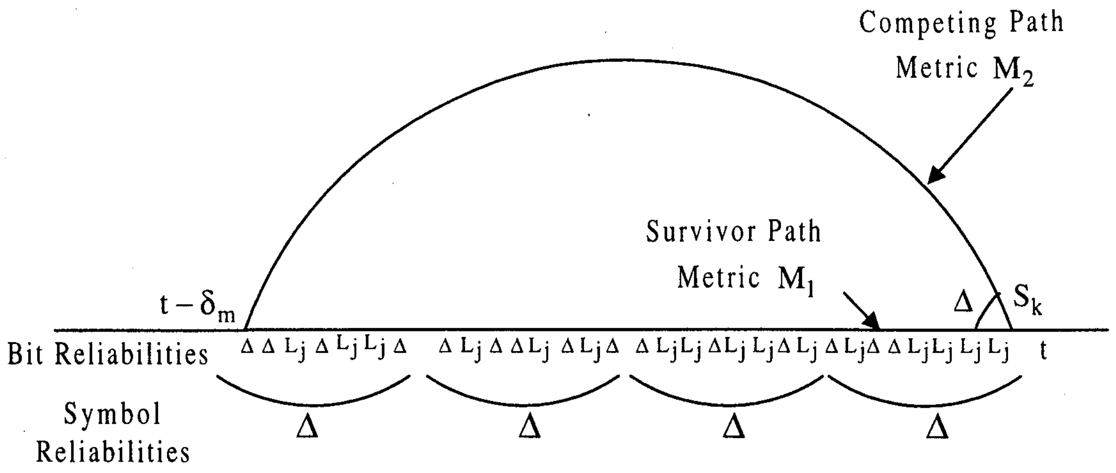

Competing Path
Metric M_2
Survivor Path
Metric M_1
$t-\delta_m$
Δ
S_k
Bit Reliabilities
$\Delta\ \Delta\ L_j\ \Delta\ L_j\ L_j\ \Delta\quad \Delta\ L_j\ \Delta\ \Delta L_j\ \Delta L_j \Delta\quad \Delta\ L_j L_j\ \Delta L_j\ L_j \Delta\ L_j\ \Delta L_j \Delta\ \Delta\ L_j L_j\ L_j\ L_j$
t
Symbol
Reliabilities
Δ
Δ
Δ
Δ

Figure 4.8 The updating of symbol reliabilities in the SOVA

The reliabilities L_j on the surviving path in the SOVA are updated by selecting the minimum between L_j and Δ, where Δ is the difference between the two paths merged in state S_k. When the bit reliabilities are converted to symbol reliabilities, the minimum bit reliability is chosen as the symbol reliability. Assuming that Δ is the minimum and this is part of the surviving path, then consecutive symbols in the surviving path have identical reliabilities. In the standard Viterbi decoder, error events occur when at time t the decoder chooses the wrong path. This is also what happens in the SOVA. If the path with the maximum metric is the wrong path, then all of the reliabilities for the differing bits will be equal to Δ. If this is the path selected by the SOVA as the output of the decoder, then the reliability will be equal for the length of the burst. It is apparent from Figure 4.8 that all symbol reliabilities in the erroneous path are equivalent. This information can be used to distinguish error paths and declare erasures, and can be accomplished by using a method similar to the one proposed by Paaske. Assume that after an initial decoding attempt of a deinterleaving frame, some of the RSW have been successfully decoded, while others have not. The error positions in the successfully decoded codewords are known. Using the above observation, if a neighboring symbol in an undecoded RSW has an identical reliability value as the corrected symbol in a decoded RSW, it is highly probable that that symbol is also an error and can be erased.

As mentioned previously, a symbol with a low reliability value does not necessarily denote a symbol error. From Figure 4.8, it can be seen that if Δ is small, then it may be contained in the reliability table for a non-decoded RSW. Assume that a RSW is decoded and the error positions are known. The correct positions are also known. These correct positions can be used to eliminate other correct symbols from being included in the reliability table for non-decoded RSW. If the neighboring positions of a correct symbol have the same reliability, then it is highly probable that this symbol is correct also. The procedure for erasure decoding using Method 2 is summarized below.

A deinterleaving frame with interleaving depth I contains I RSW. Decoding of each RSW in the frame is first attempted using Method 1. For each RSW, a table of the least reliable symbols is compiled, and for each unsuccessful decoding attempt, two additional symbols are erased, and decoding is attempted again. This process continues until either a successful decoding occurs, or a maximum amount of erasures has been reached. If less than I codewords are successfully decoded, redecoding is attempted. The RT used in Method 1 contains the least reliable symbols as determined by the output of the SOVA. The reliability table can be modified using information provided by the decoded RSW in the deinterleaving frame. Let $RSW_k(i)$ denote the k-th symbol of the i-th RSW in the deinterleaving frame, and let $L_k(i)$ be the reliability associated with $RSW_k(i)$ as determined by the SOVA. For convenience, if i + j > I, then $RSW_k(i+j)$ corresponds to $RSW_{k+1}(i+j-I)$ and if i + j < 1, $RSW_k(i-j)$ corresponds to $RSW_{k-1}(i-j+I)$. To simplify notation, $RSW_k(i \pm j)$ will be used even if the position is at k ± 1. In addition, each symbol will have a flag associated with it. Let $F_k(i)$ be the flag for the k-th symbol in the i-th RSW. There are three possible values for $F_k(i)$:

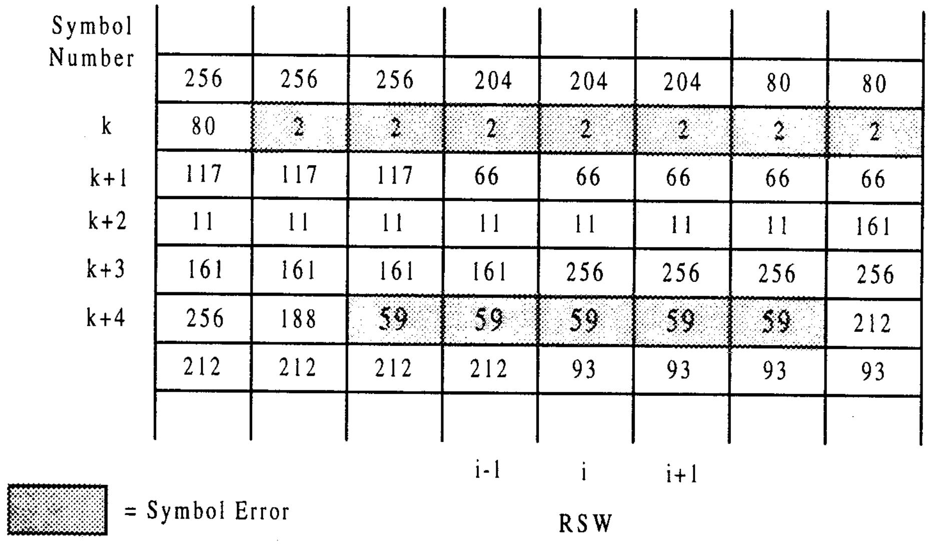
$F_k(i) = 0$ Unknown

$F_k(i) = 1$ Possible Good Erasure (PGE)

$F_k(i) = 2$ Possible Bad Erasure (PBE)

Using the already decoded RSW in the deinterleaving frame, the flags for all the symbols in the undecoded RSW will be updated. The table of least reliable symbols can be modified using the symbols that are flagged as a PGE ($F_k(i) = 1$) as the least reliable. The remainder of the table is filled with the symbols flagged as unknown ($F_k(i) = 0$) with the minimum reliabilities. Symbols that are presumed to be correct are flagged as a PBE ($F_k(i) = 2$) to avoid being used in the RT. It was found through simulation that the probability that a symbol flagged as a PGE is a GE is 0.93 and the probability that a symbol flagged as a PBE is a BE is 0.99.

Figure 4.9 shows a partial deinterleaving frame, two bursts, and the reliability values associated with each symbol. Note that the reliability in each of the burst errors is identical. Assume that RSW(i) has been successfully decoded in the deinterleaving frame of Figure 4.9. After the decoding, the error positions in RSW(i) are known. The error at position k has a reliability of $L_k(i) = 2$. Erasure Method 2 checks the reliability at position k in RSW(i-1). If the two have equal reliabilities (i.e. $L_k(i-1) = L_k(i)$), then RSW(i-1) position k is flagged as a PGE ($F_k(i-1) = 1$). If the two are not equal, searching in this direction stops, and searching in the other direction begins. Position k in RSW(i+1) is checked for equal reliabilities, and if so, position k in RSW(i+1) is flagged a PGE. Searching RSW(i±j) continues in both directions until either $L_k(i \pm j) \neq L_k(i)$, or j = I-1. The later condition can be reasoned by noting that position k in RSW(i+I) is simply position k+1 in RSW(i).



Symbol Number								
	256	256	256	204	204	204	80	80
k	80	▨ 2	▨ 2	▨ 2	▨ 2	▨ 2	▨ 2	▨ 2
k+1	117	117	117	66	66	66	66	66
k+2	11	11	11	11	11	11	11	161
k+3	161	161	161	161	256	256	256	256
k+4	256	188	▨ 59	▨ 59	▨ 59	▨ 59	▨ 59	212
	212	212	212	212	93	93	93	93
				i-1	i	i+1		

▨ = Symbol Error

RSW

Figure 4.9 Deinterleaving frame with reliability information

Assume that the two least reliable symbols contained in the RT for the first decoding try (Method 1) are symbol k with reliability $L_k = 2$ and symbol k + 2 with reliability $L_{k+2} = 11$. If two erasures are declared, position k would be a GE and position k + 2 would be a BE. After decoding RSW(i), it has been found that position k + 2 is correct. Because L_{k+2} is a small reliability, it is most likely contained in the RT for each RSW that has $L_{k+2} = 11$. The same method used to flag PGE can be used to flag PBE. After doing so, position k+2 will not be in the RT for the next iteration of decoding. The procedure used to update the flags (UFP) in the undecoded RSW using information in decoded word RSW(i) is as follows:

4.3.1 Procedure for updating the flags (UFP)

```
1. Set $k = 0$
2. If $RSW_k(i)$ is an error set $F_k(i) = 1$ (PGE). Else, set $F_k(i) = 2$ (PBE).
3. If $L_k(i) \geq L_{max}$ go to step 13
3. Initialize $j = 1$.
Start looping backwards. $RSW_k(i - j)$   $j = 1, 2, ..., I - 1$.
4. If $L_k(i - j) \neq L_k(i)$, go to step 8.
5. If $L_k(i - j) = L_k(i)$ set
        $F_k(i - j) = F_k(i)$
        $j = j + 1$
6. If $j = I - 1$, go to step 8.
7. Go to step 4.
Start looping forwards. $RSW_k(i + j)$   $j = 1, 2, ..., I - 1$.
```

8. Initialize j = 1.

9. If $L_k(i+j) \neq L_k(i)$, go to step 13.

10. If $L_k(i+j) = L_k(i)$ set

$$F_k(i+j) = F_k(i)$$

$$j = j + 1$$

11. If j = I - 1, go to step 13.

12. Go to step 9.

13. k = k + 1.

14 If k = 254, Stop else go to step 2.

After the flags have been updated using the information provided by the successfully decoded RSW, the RT for each of the yet to be decoded RSW needs to be updated. The procedure for updating the RT (URTP) is presented below.

4.3.2 Procedure for updating the reliability table (URTP):

1. Determine the number of PGE (n_{PGE}) in RSW(i) by checking the symbol flag $F_k(i)$. These GE occur in positions k_j , j = 1, 2, ..., n_{PGE}.

2. Fill the first n_{GE} positions of the reliability table with the positions where a GE has been flagged. RT(j) = k_j for j = 1, 2, ..., n_{GE}

3. If $n_{PGE} < m_e$, fill the remainder of the RT with the symbols with the minimum reliabilities determined by the SOVA output. Out of all the symbols that have $F_k(i) = 0$, determine the m_e - n_{GE} minimum symbol reliabilities. These minimum values occur in

positions k_ℓ, $\ell = 1, 2, ..., m_e - n_{PGE}$. Fill the remaining $m_e - n_{PGE}$ positions in RT with the symbols with the minimum reliabilities. $RT(n_{GE} + l) = k_\ell$, $\ell = 1, 2, ..., m_e - n_{PGE}$.

The procedure for decoding using Method 2 is presented below.

4.3.3 Procedure for decoding using Method 2

1. Attempt initial decoding using Method 1.
2. Set i_d = number of correctly decoded RSW.
3. If i_d = I, go to step 9.
4. If i_d = 0, go to step 9.
5. For each decoded codeword, declare flags in the undecoded RSW using UFP.
6. For each non decoded codeword, update the reliability table using URTP.
7. Attempt decoding using Method 1 with the updated RT.
8. If step 7 yields at least one successfully decoded RSW, go to step 2.
9. Stop.

4.3.4 An example of erasure decoding using Erasure Method 2

As an example, consider the deinterleaving frame in Figure 4.10. This frame was simulated at a signal to noise value of $E_b / N_0 = 1.9$ dB. The error positions are denoted by the shaded areas, and the reliability values are given. This frame contains 8 RSW with a total of 161 symbol errors. It should be noted that each RSW in the deinterleaving frame contains more than 16 errors. If no erasure information was used, every RSW in the frame would fail to decode. The reliability table for each RSW in the frame for the first iteration is presented in Figure 4.11, where the reliability value and symbol position are given (L_k, k). In addition, the number of decoding trials for each codeword is also shown. The RT is initially formed by using the minimum reliabilities generated by the SOVA. For the first iteration, decoding of each RSW is attempted using Method 1. RSW(1) successfully decodes after 6 decoding trials using 10 erasures. RSW(3) is successfully decoded after declaring 4 erasures (3 decoding trials), and RSW(8) decodes using 2 erasures (2 decoding trials). All other RSW fail to decode after 9 decoding trials apiece. Now, the RT is modified by updating the flags using UFP, and the updated RT is compiled using URTP giving the symbols with flag = 1 (PBE) the highest priority in the table. The modified RT is in Figure 4.12. Decoding is attempted again using Method 1 with the updated RT. For this iteration, RSW(2) decoded successfully using 14 erasures (8 decoding trials), RSW(4) using 12 erasures (7 trials), and RSW(7) using 8 erasures (5 trials). Decoding attempts for RSW(5) and RSW(6) are once again unsuccessful. The flags for RSW(5) and RSW(6) are updated using the information provided by RSW(2), RSW(4), and RSW(7). The updated RT is generated using the updated flags, and is presented in Figure 4.13. Decoding is attempted again using Method 1. RSW(5) is decoded using 16 erasures (9 decoding trials), and RSW(6) is decoded using 14 erasures (8 decoding trials). Erasure Method 2 required a total of 111 decoding trials to successfully decode this frame. This results in an average of 13.9 decoding trials per RSW.

k	RSW (1)	RSW (2)	RSW (3)	RSW (4)	RSW (5)	RSW (6)	RSW (7)	RSW (8)
3	256	256	1	1	13	211	230	256
-								
10	256	256	256	256	39	39	39	39
-								
16	10	10	10	10	10	10	256	256
17	256	254	254	256	256	256	256	207
18	105	21	105	184	184	2	2	79
19	79	79	79	79	107	107	107	256
-								
27	140	140	140	74	74	2	2	2
-								
33	166	6	166	166	237	256	256	256
34	122	218	256	256	256	256	256	256
35	256	256	4	4	4	122	256	256
-								
42	256	73	73	169	256	203	203	256
-								
52	2	2	245	245	256	256	256	256
-								
56	190	256	256	256	256	256	178	178
57	256	256	190	3	3	2	2	2
58	256	256	256	256	256	256	5	5
59	5	24	84	84	84	72	72	231
-								
66	5	5	5	5	5	5	169	256
67	256	256	256	256	256	256	256	126
68	194	194	210	256	256	46	46	46
-								
74	256	256	256	201	201	90	90	90
75	89	106	116	116	116	116	67	67
76	67	67	87	173	219	256	187	203
-								
82	256	222	83	83	83	178	256	256
83	256	169	122	73	73	179	179	256
84	256	256	256	256	256	256	6	6
85	16	16	73	73	1	1	1	1
86	27	27	94	94	179	256	256	256
87	256	256	243	243	116	256	256	91
88	91	91	91	224	228	228	256	256
89	209	209	87	213	213	23	23	23
90	23	23	29	34	61	61	215	215
-								
97	26	145	145	145	165	165	256	256
98	256	168	168	168	168	234	256	256
99	46	44	53	53	53	163	247	247
-								
103	256	256	256	256	16	16	111	256
-								
106	92	92	249	249	249	249	256	256
-								
109	256	256	256	254	256	256	43	43
110	43	23	22	23	68	68	203	256
111	256	256	256	256	178	9	9	9
112	9	9	9	9	58	58	172	256

k	RSW (1)	RSW (2)	RSW (3)	RSW (4)	RSW (5)	RSW (6)	RSW (7)	RSW (8)
125	256	231	256	256	30	30	2	2
126	30	256	256	2	2	2	2	2
127	100	127	157	175	256.	256	256	256
128	189	101	101	101	101	101	20	28
-								
142	117	117	141	90	90	90	113	161
-								
144	256	189	177	177	252	25	25	25
-								
149	256	256	256	193	256	256	19	19
150	19	19	19	31	31	31	31	22
-								
157	51	51	118	119	119	119	180	256
-								
160	256	256	256	256	256	256	256	126
161	38	38	69	69	69	69	69	69
-								
164	251	185	185	185	222	256	256	81
165	81	81	137	207	256	97	86	38
166	86	119	208	250	241	154	154	102
-								
170	256	193	51	51	51	63	97	256
-								
176	72	52	52	77	77	169	201	256
-								
179	256	256	172	256	124	48	5	5
180	5	174	256	256	256	256	256	256
-								
184	127	127	30	30	108	244	244	256
-								
189	256	256	256	96	33	33	256	256
190	111	111	57	57	33	255	256	201
191	215	256	256	256	256	256	6	6
192	6	6	6	6	6	95	256	256
193	233	172	172	185	201	201	114	114
194	119	10	10	10	10	10	146	256
-								
206	256	256	256	136	169	169	162	114
207	6	6	6	6	22	256	256	256
-								
212	256	256	66	66	66	66	66	159
-								
218	256	256	256	256	256	11	11	115
-								
229	256	256	39	39	39	133	256	256
230	256	174	3	3	3	3	3	14
231	14	16	49	49	49	256	256	256
-								
249	256	256	256	185	185	225	256	12
-								
253	256	256	256	31	31	31	31	200
# of Errors	18	22	17	22	23	23	19	17

Figure 4.10 An example of a deinterleaving frame with reliability values

i	RSW(1)	RSW(2)	RSW(3)	RSW(4)	RSW(5)	RSW(6)	RSW(7)	RSW(8)
1	▨ 5, 59	3, 213	1, 3	▨ 1, 3	1, 72	1, 72	1, 85	▨ 1, 85
2	▨ 5, 66	▨ 5, 66	▨ 3, 230	▨ 2, 126	▨ 1, 85	▨ 1, 85	▨ 2, 18	▨ 2, 27
3	5, 180	▨ 6, 33	▨ 4, 35	▨ 3, 57	2, 21	▨ 2, 18	▨ 2, 27	2, 57
4	▨ 6, 192	6, 192	▨ 5, 66	▨ 3, 230	▨ 2, 57	▨ 2, 27	2, 57	▨ 2, 126
5	6, 207	6, 207	▨ 6, 192	▨ 4, 35	2, 126	2, 57	▨ 2, 126	5, 58
6	▨ 7, 52	▨ 7, 52	6, 207	▨ 5, 66	▨ 3, 230	▨ 2, 126	3, 230	5, 179
7	9, 112	9, 112	9, 112	▨ 6, 192	▨ 4, 5	▨ 3, 230	5, 58	6, 84
8	▨ 10, 16	▨ 10, 16	10, 16	6, 207	▨ 5, 66	▨ 6, 66	▨ 5, 179	▨ 6, 191
9	▨ 14, 231	▨ 10, 194	▨ 10, 194	▨ 9, 112	▨ 6, 192	▨ 9, 111	6, 84	▨ 7, 125
10	▨ 16, 85	▨ 16, 85	▨ 19, 150	10, 16	10, 16	10, 16	▨ 6, 191	9, 111
11	▨ 19, 150	▨ 16, 231	▨ 22, 110	▨ 10, 194	▨ 10, 194	▨ 10, 194	▨ 7, 125	11, 55
12	22, 151	▨ 19, 150	▨ 29, 90	15, 141	▨ 13, 3	11, 55	▨ 9, 111	▨ 14, 230
13	▨ 23, 90	21, 18	▨ 30, 184	▨ 22, 110	▨ 16, 103	11, 218	11, 55	▨ 15, 249
14	26, 97	▨ 22, 110	32, 7	23, 40	22, 207	▨ 16, 103	11, 218	18, 13
15	▨ 27, 86	▨ 23, 90	39, 229	▨ 30, 184	30 ,125	▨ 23, 89	17, 133	▨ 19, 149
16	30, 126	▨ 24, 59	49, 231	▨ 31, 150	▨ 31, 150	▨ 25, 144	18, 13	▨ 20, 128
# of Trials	6	9	3	9	9	9	9	2

▨ Indicates an erasure hits an error (GE)

Figure 4.11 The reliability table for the first iteration

i	RSW(1)	RSW(2)	RSW(3)	RSW(4)	RSW(5)	RSW(6)	RSW(7)	RSW(8)
1		▨ PGE, 42		▨ PGE, 35	▨ PGE, 10	▨ PGE, 18	PGE, 10	
2		▨ PGE, 52		▨ PGE, 59	▨ PGE, 35	▨ PGE, 27	PGE, 27	
3		▨ PGE, 66		▨ PGE, 66	▨ PGE, 59	▨ PGE, 68	PGE, 68	
4		▨ PGE, 85		▨ PGE, 85	▨ PGE, 66	▨ PGE, 74	PGE, 74	
5		▨ PGE, 86		▨ PGE, 110	▨ PGE, 85	▨ PGE, 85	PBE, 85	
6		▨ PGE, 89		▨ PGE, 126	▨ PGE, 126	▨ PGE, 89	PGE, 89	
7		PGE, 90		▨ PGE, 161	▨ PGE, 161	PGE, 126	▨ PGE, 109	
8		▨ PGE, 106		▨ PGE, 170	▨ PGE, 170	▨ PGE, 144	▨ PGE, 125	
9		▨ PGE, 110		▨ PGE, 184	▨ PGE, 192	▨ PGE, 161	▨ PGE, 126	
10		▨ PGE, 150		▨ PGE, 192	▨ PGE, 194	▨ PGE, 194	▨ PGE, 128	
11		PGE, 157		▨ PGE, 194	▨ PGE, 230	▨ PGE, 230	▨ PGE, 144	
12		▨ PGE, 161		▨ PGE, 230	1, 72	1, 72	▨ PGE, 149	
13		▨ PGE, 176		▨ 3, 57	2, 21	▨ 2, 18	▨ PGE, 161	
14		▨ PGE, 192		15, 41	▨ 13, 3	▨ 6,66	▨ PGE, 191	
15		▨ PGE, 194		23, 40	▨ 16, 103	11, 218	▨ PGE, 230	
16		3, 213		▨ 31, 150	22, 207	▨ 16, 103	▨ 2, 18	
# of Trials	6	17	3	16	18	18	14	2

▨ Indicates an erasure hits an error (GE)

Figure 4.12 The reliability table for the second iteration

i	RSW(1)	RSW(2)	RSW(3)	RSW(4)	RSW(5)	RSW(6)	RSW(7)	RSW(8)
1					PGE, 10	PGE, 10		
2					PGE, 35	PGE, 18		
3					PGE, 59	PGE, 27		
4					PGE, 66	PGE, 68		
5					PGE, 82	PGE, 74		
6					PGE, 83	PGE, 89		
7					PGE, 89	PGE, 111		
8					PGE, 126	PGE, 126		
9					PGE, 142	PGE, 142		
10					PGE, 161	PGE, 144		
11					PGE, 170	PGE, 161		
12					PGE, 192	PGE, 194		
13					PGE, 194	PGE, 212		
14					PGE, 212	PGE, 230		
15					PGE, 230	PGE, 253		
16					PGE, 253	1, 72		
# of Trials	6	17	3	16	27	26	14	2

▨ Indicates an erasure hits an error (GE)

Figure 4.13 The reliability table for the third iteration

4.3.5 Results for Erasure Method 2

The statistics for Method 2 are presented in Table 4.2. The RS symbol error rate at the output of the Viterbi decoder is given. The percentage of frame failures and RSW failures after errors only RS decoding gives an indication how the standard concatenated system performs. The percentage of frames failures and the percentage of RSW failures using Method 2 are also compiled for various values of E_b / N_0. The number of decoding trials per RSW is also given.

Table 4.2 Simulation results using Method 2 with interleaving depth I = 8

		Without Erasures		Using Method 2		
E_b / N_0	Byte Rate after VD	% Frames in error	% RSW in error	% Frames in error	% RSW in error	# of trials per RSW
1.7	0.093	100.0	95.2	92.0	75.6	13.6
1.75	0.086	99.8	89.7	81.9	57.3	14.3
1.8	0.079	99.4	79.9	61.1	36.6	13.2
1.85	0.072	97.0	66.1	38.1	18.1	10.8
1.9	0.066	91.6	52.3	20.2	8.3	7.7
1.95	0.061	81.6	37.3	11.6	4.2	5.23
2.0	0.055	66.8	24.9	4.4	1.12	3.36
2.05	0.050	46.9	14.9	1.7	0.4	2.18
2.1	0.046	32.7	8.5	0.3	0.0	1.56
2.15	0.042	19.4	4.4	0.0	0.0	1.27
2.2	0.038	10.1	1.9	0.0	0.0	1.09
2.3	0.031	1.0	0.2	0.0	0.0	1.01
2.4	0.025	0.2	0.0	0.0	0.0	1.0

When the results in Table 4.2 are compared to Paaske's results, the first difference that is noticed is the difference in the RS symbol error rate at the output of the Viterbi decoder. The symbol error rate given in Table 4.2 is obtained using system 5 ($\delta = 100$ and $L = 64$). This symbol error rate is approximately $3 \cdot 10^{-3}$ lower than the symbol error rate in Paaske's results. The better performance at the output of the Viterbi decoder is most likely due to the increased soft decision levels (L = 64) as compared to the 8 level soft decision used by Paaske. This slightly smaller RS symbol error rate translates into slightly better results for the errors only decoding. The frame error rate and RSW error rate in Table 4.2 are typically 2% or 3% smaller than the results that Paaske obtained.

The results obtained using Method 2 are better than Paaske's results in three regards. The percentage of RSWs and frames in error is lower. For example, at E_b / N_0 = 1.9 dB, the percentage of frames and RSWs in error using Paaske's method are 26.8% and 16.3%, compared to 20.2% and 8.3% using Method 2. Method 2 also obtains these results using considerably less decoding trials. Paaske's method requires an average of 112.4 decoding trials per RSW to obtain the reduction. Method 2 requires an average of 7.7 decoding trials per RSW. The third improvement is in the ability to obtain improvement for lower values of E_b / N_0. The results in [7] indicate that no improvement is obtained at E_b / N_0= 1.8 dB. Using Method 2, the percentage of frames in error was reduced from 99.4% to 61.1% and the percentage of RSW in error was reduced from 79.9% to 36.6%. The average number of decoding trials per RSW needed to accomplish this is 13.2. Error reductions were obtained for values of E_b / N_0 as low as 1.7 dB. Both Method 2 and Paaske's method obtained 0.3 dB gain over the system using no erasures.

The reduction in average number of RSW trials as compared to Paaske's method can be attributed to the ability to use erasures in the first decoding attempt. This is useful for decoding RSW that contain more than 16 errors, as in the example in Figure 4.10. Each RSW in the frame contains more than 16 errors. Paaske's method would obtain no successful decoding in the first iteration, and would have to resort to using EP4 to obtain the first successful decoding. EP4 randomly erases two symbols and attempts decoding. This repeats until a successful decoding or this has been attempted T_{max} times. It can easily be seen that this frame would require a large amount of decoding trials. In addition, Paaske's method requires two successfully decoded RSW with errors in the identical positions in order to use EP1. Erasures declared using EP1 have a probability of 0.96 of being a GE. If not, a less reliable procedure must be used (EP2-EP4). Erasure Method 2 only requires 1 RSW to be able to declare erasures with a probability of 0.93 of being a GE. It can be seen that erasure Method 2 converges on the erasures quicker than Paaske's method, and thus, requires less decoding trials per RSW.

4.4 Erasure Method 3

Method 2 can be modified to reduce the number of decoding trials per RSW by making a few observations. In Method 1, erasures were erased two at a time until either a decoding success was obtained, or a maximum number of erasures were declared. This was done because symbols with low reliabilities were not guaranteed to be GE. Erasing more than two at a time may cause the decoding capability of the code to be decreased. This occurs if the number of BE is greater than the number of GE. In Method 2, when a PGE is declared, it is highly probable that this is a GE (0.93). Instead of erasing two symbols from the RT at a time, all symbols flagged as PGE are automatically erased. If, after decoding with these symbols erased it is still not successful, then two more symbols are erased until the maximum allowable erasures is reached.

A second modification updates the flags after a RSW has been decoded correctly, rather than after all of the RSW in the frame have been attempted. This allows for the possibility for the highly reliable PGE to be declared in the first decoding pass. Both of these modifications reduce the average number of RS decoding trials when compared to Method 2, with very little effect on the performance. The procedure can best be demonstrated with an example.

4.4.1 An example of erasure decoding using Erasure Method 3

Consider the deinterleaving frame in Figure 4.10. The reliability table before the first decoding attempt is given in Figure 4.11. Much like Method 2, decoding of RSW(1) is first attempted using erasure Method 1 (i.e. decoding is attempted using no erasures, and decoding is repeated until either a successful decoding or a maximum number of erasures has been reached). RSW(1) is finally decoded after 6 decoding trials and 10 erasures. In Method 2, the next step would be to attempt decoding of RSW(2) using Method 1. Method 3 instead updates the flags immediately after RSW(1) has been successfully decoded. This results in 13 PGE being declared in RSW(2), 12 of which are GE. This modified reliability table is given in Figure 4.14. At this point, RSW(2) is ready to be decoded. Instead of erasing two symbols at a time as with Method 2, all PGE are erased and decoding is attempted. For RSW(2), this still does not yield a successful decoding, but as can be seen from Figure 4.14, erasing two at a time would achieve the same results, but with more decoding trials. Because the number of PGE are less than the maximum number of erasures allowed, two more symbols are erased, and decoding is attempted. RSW(2) finally decodes once 16 erasures have been declared. This is accomplished in 3 decoding trials, as compared the 17 trials needed to decode RSW(2) using Method 2. The flags are now updated using the information provided by RSW(2), and the resulting reliability table is given in Figure 4.15. The flags declared from RSW(1) and RSW(2) yield 10 PGEs in RSW(3), 8 of which are GE. It should also be noted that the PBE that were declared removed potential BE from the table. Symbols at positions 207 and 112 would have been included in the reliability table. These would have resulted in BE if used. The decoding of RSW(3) is accomplished by erasing the 10 PGE in the table. This resulted in a decoding success in only 1 decoding trial. The flags are modified using the information provided by RSW(3), and the resulting reliability table is given in Figure 4.16. There are 11 PGE in RSW(4), and all are GE. This RSW requires all 16 erasures for a successful decoding, and is accomplished with 4 decoding trials. The flags are updated and the resulting reliability table is shown in Figure 4.17. There are 15 PGE in RSW(5) and 14 are GE. The codeword contains 23 errors, and cannot be decoded with the 16 erasures in the table. This codeword will require 15 out of 16 erasures to be GE for a decoding success. Two decoding trials were attempted on this codeword. Decoding of RSW(6) is attempted by declaring erasures at the positions where the 9 PGE have been flagged. This does not yield a successful decoding, so erasures are declared 2 at a time until the maximum is reached, at which point a decoding success has not been obtained. This required 5 decoding trials. RSW(7) is successfully decoded by erasing all 12 positions with PGE flags, and decoding. RSW(7) required only 1 decoding trial to decode successfully. The flags are updated, and the resulting reliability table is given in Figure 4.18. RSW(8) contains 16 PGE, and 13 are GE. There are only 17 errors in RSW(7), and is easily decoded in one trial by declaring 16 erasures. The flags are once again updated and the resulting reliability table is given in Figure 4.19. Decoding of RSW(5) is attempted a second time. New information has been provided by RSW(7) and RSW(8). There are 17 PGE in RSW(5), and all are erased in the first decoding attempt. In Method 3, more than 16 erasures can be declared if all are PGE. RSW(5) successfully decodes and the updated reliability table is given in Figure 4.20. RSW(6) has 20 PGE, with 19 of these being GE. All 20 symbols are erased, and the word is successfully decoded. The total number of trials for this frame is 25, which corresponds to an average number of trials per RSW of 3.13. This is significantly less than the number of trials required for erasure Method 2 to decode the same frame. The results of the simulation are given in Table 4.3. It can be seen from the table that a gain of 0.25 dB is obtainable.

i	RSW(1)	RSW(2)	RSW(3)	RSW(4)	RSW(5)	RSW(6)	RSW(7)	RSW(8)
1		PGE, 16	PGE, 16	PGE, 16	PGE, 16	PGE, 16	PGE, 58	PGE, 58
2		PGE, 52	PGE, 66	PGE, 66	PGE, 66	PGE, 89	PGE, 75	PGE, 75
3		▨ PGE, 66	▨ PGE, 76	▨ PGE, 192	▨ PGE, 192	1, 72	▨ PGE, 89	▨ PGE, 89
4		▨ PGE, 76	▨ PGE, 150	▨ 1, 3	1, 72	▨ 1, 85	▨ PGE, 109	▨ PGE, 109
5		▨ PGE, 85	▨ PGE, 192	▨ 2, 126	▨ 1, 85	▨ 2, 18	▨ PGE, 149	▨ PGE, 149
6		▨ PGE, 86	1, 3	▨ 3, 57	2, 21	▨ 2, 27	▨ PGE, 191	▨ PGE, 191
7		▨ PGE, 89	▨ 3, 230	▨ 3, 230	▨ 2, 57	2, 57	1, 85	▨ PGE, 230
8		▨ PGE, 90	▨ 4, 35	▨ 4, 35	▨ 2, 126	▨ 2, 126	▨ 2, 18	▨ 1, 85
9		▨ PGE, 106	▨ 10, 194	▨ 10, 194	▨ 3, 230	▨ 3, 230	▨ 2, 27	▨ 2, 27
10		▨ PGE, 150	▨ 22, 110	15, 141	▨ 4, 5	▨ 6, 66	2, 57	2, 57
11		PGE, 157	▨ 29, 90	▨ 22, 110	▨ 10, 194	▨ 10, 194	▨ 2, 126	▨ 2, 126
12		▨ PGE, 161	▨ 30, 184	23, 40	▨ 13, 3	11, 55	▨ 3, 230	6, 84
13		▨ PGE, 192	32, 7	▨ 30, 184	▨ 16, 103	11, 218	6, 84	▨ 7, 125
14		3, 213	39, 229	▨ 31, 150	22, 207	▨ 16, 103	▨ 7, 125	11, 55
15		▨ 6, 33	49, 231	32, 7	30 ,125	▨ 25, 144	11, 55	▨ 15, 249
16		▨ 10, 194	▨ 51, 170	33, 61	▨ 31, 150	30, 125	11, 218	18, 13
# of Trials	6	3	-	-	-	-	-	-

▨ Indicates an erasure hits an error (GE)

Figure 4.14 Reliability table after RSW(1) successfully decodes

i	RSW(1)	RSW(2)	RSW(3)	RSW(4)	RSW(5)	RSW(6)	RSW(7)	RSW(8)
1			PGE, 16	PGE, 16	PGE, 16	PGE, 16	PGE, 58	PGE, 58
2			PGE, 66	▨ PGE, 66	▨ PGE, 66	▨ PGE, 89	PGE, 75	PGE, 75
3			▨ PGE, 76	▨ PGE, 192	▨ PGE, 192	PGE, 194	▨ PGE, 89	▨ PGE, 89
4			▨ PGE, 150	PGE, 97	▨ PGE, 194	1, 72	▨ PGE, 109	▨ PGE, 109
5			▨ PGE, 192	▨ PGE, 110	1, 72	▨ 1, 85	▨ PGE, 149	▨ PGE, 149
6			▨ PGE, 42	▨ PGE, 194	▨ 1, 85	▨ 2, 18	▨ PGE, 191	▨ PGE, 191
7			PGE, 97	▨ 1, 3	2, 21	▨ 2, 27	1, 85	▨ PGE, 230
8			▨ PGE, 110	▨ 2, 126	▨ 2, 57	2, 57	▨ 2, 18	PGE, 164
9			▨ PGE, 176	▨ 3, 57	▨ 2, 126	▨ 2, 126	▨ 2, 27	▨ 1, 85
10			▨ PGE, 194	▨ 3, 230	▨ 3, 230	▨ 3, 230	2, 57	▨ 2, 27
11			1, 3	▨ 4, 35	▨ 4, 5	▨ 6, 66	▨ 2, 126	2, 57
12			▨ 3, 230	15, 141	▨ 13, 3	11, 55	▨ 3, 230	▨ 2, 126
13			▨ 4, 35	23, 40	▨ 16, 103	11, 218	6, 84	6, 84
14			▨ 29, 90	▨ 30, 184	22, 207	▨ 16, 103	▨ 7, 125	▨ 7, 125
15			▨ 30, 184	▨ 31, 150	30 ,125	▨ 25, 144	11, 55	11, 55
16			39, 229	▨ 31, 253	▨ 31, 150	30, 125	11, 218	▨ 15, 249
# of Trials	6	3	1	-	-	-	-	-

▨ Indicates an erasure hits an error (GE)

Figure 4.15 Reliability table after RSW(2) successfully decodes

i	RSW(1)	RSW(2)	RSW(3)	RSW(4)	RSW(5)	RSW(6)	RSW(7)	RSW(8)
1				▨ PGE, 66	▨ PGE, 66	▨ PGE, 89	PGE, 58	PGE, 58
2				▨ PGE, 192	▨ PGE, 192	▨ PGE, 194	PGE, 75	PGE, 75
3				▨ PGE, 110	▨ PGE, 194	▨ PGE, 161	▨ PGE, 89	▨ PGE, 89
4				▨ PGE, 194	▨ PGE, 35	▨ PGE, 230	▨ PGE, 109	▨ PGE, 109
5				▨ PGE, 35	▨ PGE, 59	1, 72	▨ PGE, 149	▨ PGE, 149
6				▨ PGE, 59	▨ PGE, 161	▨ 1, 85	▨ PGE, 191	▨ PGE, 191
7				▨ PGE, 85	▨ PGE, 170	▨ 2, 18	▨ PGE, 161	▨ PGE, 230
8				▨ PGE, 161	▨ PGE, 230	▨ 2, 27	▨ PGE, 230	PGE, 164
9				▨ PGE, 170	1, 72	2, 57	1, 85	▨ PGE, 161
10				▨ PGE, 184	▨ 1, 85	▨ 2, 126	▨ 2, 18	▨ 1, 85
11				▨ PGE, 230	2, 21	▨ 6, 66	▨ 2, 27	▨ 2, 27
12				▨ 2, 126	▨ 2, 57	11, 55	2, 57	2, 57
13				▨ 3, 57	▨ 2, 126	11, 218	▨ 2, 126	▨ 2, 126
14				15, 141	▨ 3, 230	▨ 16, 103	6, 84	6, 84
15				23, 40	▨ 4, 5	▨ 25, 144	▨ 7, 125	▨ 7, 125
16				▨ 31, 150	▨ 13, 3	30, 125	11, 55	11, 55
# of Trials	6	3	1	4	-	-	-	-

▨ Indicates an erasure hits an error (GE)

Figure 4.16 Reliability table after RSW(3) successfully decodes

i	RSW(1)	RSW(2)	RSW(3)	RSW(4)	RSW(5)	RSW(6)	RSW(7)	RSW(8)
1					▨ PGE, 66	▨ PGE, 89	PGE, 58	PGE, 58
2					PGE, 192	PGE, 194	PGE, 75	PGE, 75
3					▨ PGE, 194	▨ PGE, 161	PGE, 89	PGE, 89
4					▨ PGE, 35	▨ PGE, 230	PGE, 109	▨ PGE, 109
5					▨ PGE, 59	▨ PGE, 111	▨ PGE, 149	▨ PGE, 149
6					▨ PGE, 161	▨ PGE, 126	▨ PGE, 191	▨ PGE, 191
7					▨ PGE, 170	▨ PGE, 142	▨ PGE, 161	▨ PGE, 230
8					▨ PGE, 230	▨ PGE, 150	▨ PGE, 230	PGE, 164
9					▨ PGE, 82	▨ PGE, 212	▨ PGE, 111	▨ PGE, 161
10					PGE, 83	1, 72	▨ PGE, 126	PGE, 111
11					▨ PGE, 89	▨ 1, 85	PGE, 150	▨ PGE, 126
12					▨ PGE, 126	▨ 2, 18	▨ PGE, 212	▨ 1, 85
13					▨ PGE, 142	▨ 2, 27	1, 85	▨ 2, 27
14					▨ PGE, 150	2, 57	▨ 2, 18	2, 57
15					▨ PGE, 212	▨ 6, 66	▨ 2, 27	6, 84
16					1, 72	11, 55	2, 57	▨ 7, 125
# of Trials	6	3	1	4	2	5	1	-

▨ Indicates an erasure hits an error (GE)

Figure 4.17 Reliability table after RSW(4) successfully decodes

i	RSW(1)	RSW(2)	RSW(3)	RSW(4)	RSW(5)	RSW(6)	RSW(7)	RSW(8)
1					PGE, 66	▨ PGE, 89		▨ PGE, 89
2					PGE, 192	▨ PGE, 194		▨ PGE, 109
3					PGE, 194	▨ PGE, 161		▨ PGE, 149
4					PGE, 35	▨ PGE, 230		▨ PGE, 191
5					PGE, 59	▨ PGE, 111		▨ PGE, 230
6					▨ PGE, 161	▨ PGE, 126		PGE, 164
7					▨ PGE, 170	▨ PGE, 142		▨ PGE, 161
8					▨ PGE, 230	PGE, 158		PGE, 111
9					▨ PGE, 82	▨ PGE, 212		▨ PGE, 126
10					PGE, 83	▨ PGE, 18		▨ PGE, 125
11					▨ PGE, 89	▨ PGE, 27		▨ PGE, 128
12					▨ PGE, 126	▨ PGE, 68		▨ PGE, 27
13					▨ PGE, 142	▨ PGE, 74		▨ PGE, 68
14					▨ PGE, 150	▨ PGE, 144		▨ PGE, 74
15					▨ PGE, 212	▨ PGE, 253		▨ PGE, 144
16					▨ PGE, 253	1, 72		PGE, 179
# of Trials	6	3	1	4	2	5	1	1

▨ Indicates an erasure hits an error (GE)

Figure 4.18 Reliability table after RSW(7) successfully decodes

i	RSW(1)	RSW(2)	RSW(3)	RSW(4)	RSW(5)	RSW(6)	RSW(7)	RSW(8)
1					PGE, 66	PGE, 89		
2					PGE, 192	PGE, 194		
3					PGE, 194	PGE, 161		
4					PGE, 35	PGE, 230		
5					PGE, 59	PGE, 111		
6					PGE, 161	PGE, 126		
7					PGE, 170	PGE, 142		
8					PGE, 230	PGE, 150		
9					PGE, 82	PGE, 212		
10					PGE, 83	PGE, 18		
11					PGE, 89	PGE, 27		
12					PGE, 126	PGE, 68		
13					PGE, 142	PGE, 74		
14					PGE, 150	PGE, 144		
15					PGE, 212	PGE, 253		
16					PGE, 253	PGE, 85		
17					PGE, 85			
# of Trials	6	3	1	4	3	5	1	1

▨ Indicates an erasure hits an error (GE)

Figure 4.19 Reliability table after RSW(8) successfully decodes

i	RSW(1)	RSW(2)	RSW(3)	RSW(4)	RSW(5)	RSW(6)	RSW(7)	RSW(8)
1						PGE, 89		
2						PGE, 194		
3						PGE, 161		
4						PGE, 230		
5						PGE, 111		
6						PGE, 126		
7						PGE, 142		
8						PGE, 150		
9						PGE, 212		
10						PGE, 18		
11						PGE, 27		
12						PGE, 68		
13						PGE, 74		
14						PGE, 144		
15						PGE, 253		
16						PGE, 85		
17						PGE, 57		
18						PGE, 90		
19						PGE, 103		
20						PGE, 112		
# of Trials	6	3	1	4	3	6	1	1

▨ Indicates an erasure hits an error (GE)

Figure 4.20 Reliability table after RSW(5) successfully decodes

Table 4.3 Simulation results using Method 3 with interleaving depth I = 8

		Without Erasures		Using Method 3		
E_b / N_0	Byte Rate after VD	% Frames in error	% RSW in error	% Frames in error	% RSW in error	# of trials per RSW
1.7	0.093	100.0	95.2	94.7	78.1	9.7
1.75	0.086	99.8	89.7	83.1	59.2	8.56
1.8	0.079	99.4	79.9	64.1	37.7	6.68
1.85	0.072	97.0	66.1	42.8	20.5	4.54
1.9	0.066	91.6	52.3	23.6	9.6	3.03
1.95	0.061	81.6	37.3	13.2	4.6	2.19
2.0	0.055	66.8	24.9	6.7	1.9	1.51
2.05	0.05	46.9	14.9	2.8	0.6	1.27
2.1	0.046	32.7	8.5	0.6	0.08	1.13
2.15	0.042	19.4	4.4	0.2	0.0	1.07
2.2	0.038	10.1	1.9	0.0	0.0	1.03
2.3	0.031	1.0	0.2	0.0	0.0	1.0
2.4	0.025	0.2	0.0	0.0	0.0	1.0

As can bee seen comparing the results of the two methods in Tables 4.2 and 4.3, the significant difference is the reduction in the average number of decoding trials needed per RSW when using Method 3. The tradeoff in this reduction in the average number of decoding trials is a slight increase in the percentage of frame and word errors using Method 3. Method 2 outperforms Method 3 by a few percent. The BER curves for Method 2 and Method 3 are given in Figure 4.21. Using erasure Method 2 and Method 3 resulted in approximately 0.3 dB and 0.25 dB gain respectively over the concatenated system using no erasures.

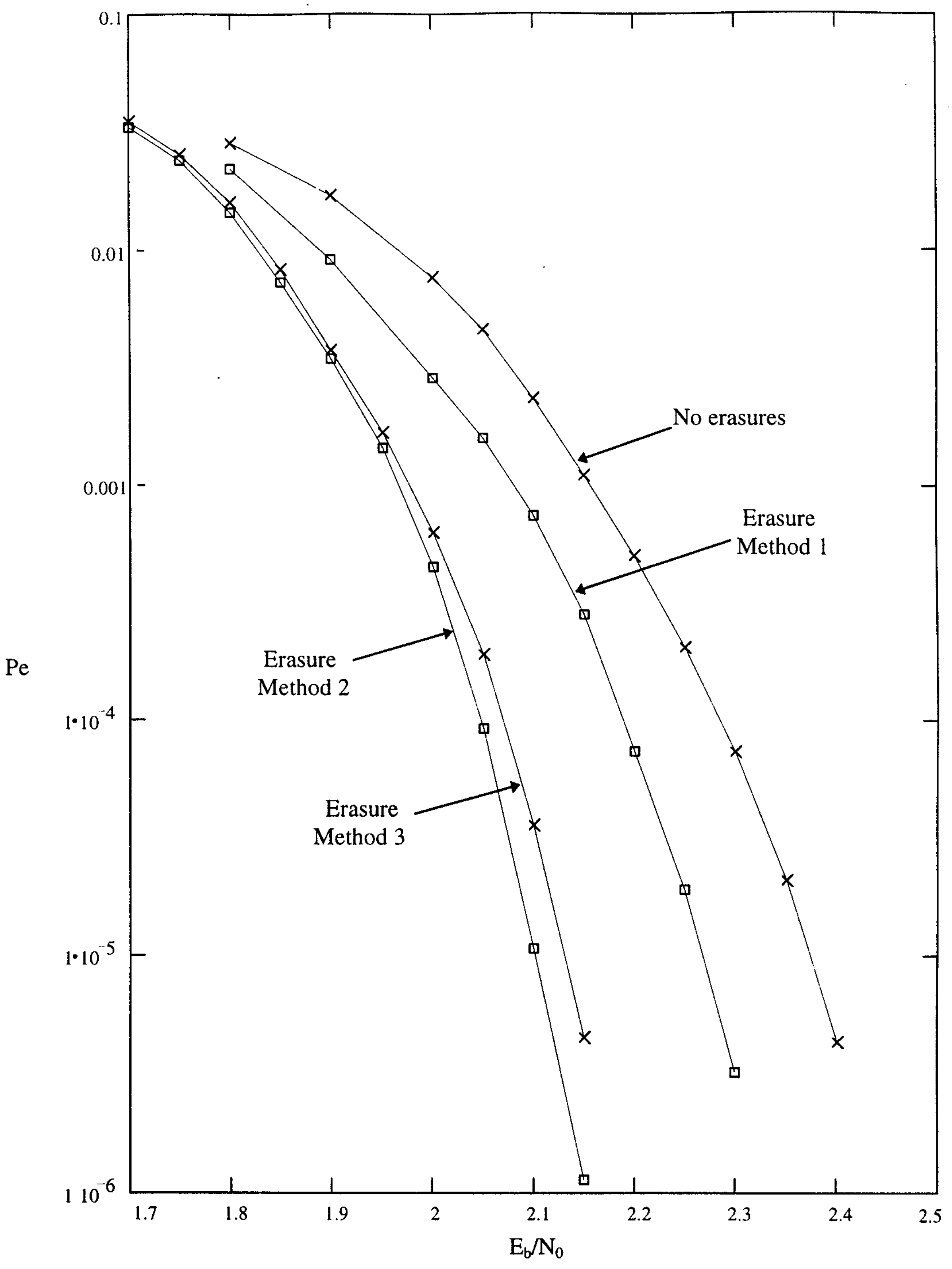


Figure 4.21 Simulation results of the concatenated system using various erasure methods

Chapter 5

Conclusions

Performance of the concatenated coding system through the use of RS symbol erasures has been demonstrated. The first method investigated uses reliability information generated by a modified Viterbi decoder. This information is derived from the metric difference of two paths merging in each state. The reliability for the bits along the surviving path are updated using this metric difference. This method yielded about a 0.1 dB improvement over the concatenated system using no erasures. This gain was independent of the truncation length, interleaving depth, and number of soft decision levels.

In the second method proposed, the reliability table is refined using information provided by the decoded RSW in the deinterleaving frame. The error positions are known in the decoded word, and the method searches for equal reliabilities in neighboring, non-decoded RSW. If the reliabilities are the same, then symbols are erased. This method yielded approximately 0.3 dB gain over the standard concatenated system. The average number of decoding trials per RSW is substantially less than the results presented by Paaske [7]. Using Method 3, the number of trials per RSW is reduced even further. In addition, Methods 2 and 3 perform reasonably well at very low values of Eb/No (i.e. < 1.9 dB). The cost is the increased complexity of having to use the SOVA in place of the standard Viterbi decoder.

It has been shown that the use of convolutional code as the inner code and a Reed Solomon code as the outer code provide considerable gains. The rate 1/2 K = 7 convolutional code concatenated with a (255, 223) t = 16 error correcting Reed Solomon code yielded about 8 dB gain over uncoded BPSK and approximately 9 dB with an interleaving depth of I = 8. The use of an interleaver with depth I = 8 gives a gain of approximately 0.5 dB as compared to the same system using no interleaver. It has been demonstrated that the use of a real system results in approximately a 0.3 dB loss when compared to the ideal system using the same truncation length and number of soft decision levels. It has been shown that a 0.15 dB improvement can be obtained by increasing the number of soft decisions used from L = 8 to L = 64. A gain of 0.2 dB is attainable if the truncation length in the Viterbi decoder is increased from $\delta = 32$ to $\delta = 100$. These gains were obtained regardless of the interleaving depth used.

5.1 Future Research

Erasure method 1 declares RS symbol erasures by using reliability information provided by the SOVA. This method could be improved by using the maximum a posterori (MAP) algorithm in place of the SOVA. The MAP algorithm is an alternate method for decoding convolutional codes. The Viterbi decoder finds the maximum a posterori probability for the entire path through the trellis. The MAP algorithm, on the other hand, finds the maximum a posterori probability for each outgoing bit. The MAP

Algorithm was designed to minimize the word error probability rather than the sequence error probability as the Viterbi algorithm does. This algorithm is more complex than the Viterbi, but easily provides log-likelihood (reliability) values at its output. The reliability values generated by the MAP algorithm yield superior results when compared to the SOVA, but is very complex. The Map algorithm could be used in place of the SOVA in the concatenated coding system presented here.

The performance of Method 2 could also be improved upon. The special case where GE are flagged BE, and vise versa, might be able to be detected and solved. Several methods were tried unsuccessfully to fix the problem. Performance improvements could be obtained by incorporating some of the elements used in Paaske's method, such as EP1, EP3, and EP4. In Paaske's method, when the highly reliable methods (EP1 and EP2) can not decode some of the RSW in the frame, the backup methods are used to exhaustively try and decode, which results in a high average number of decoding trials per RSW. But if through the use of this exhaustive search, a previously undecodable RSW is now correctly decoded, then this RSW can possibly help in the decoding of other non-decoded RSW in the frame. This is where a large number of decoding trials may be justifiable. When Methods 2 and 3 fail to decode a frame, there is no backup method that might provide improvement. The lack of a backup procedure, is one of the reasons that the number of trials is low compared to Paaske's method. This lack of a backup procedure is somewhat justified. In Paaske's method, EP4 is used to systematically guess the positions of the errors. This guessing is effective when the number of errors is not much larger than 18 or 19. For Methods 1 and 2, most RSW with

18 or 19 errors can be decoded using the initial decoding using Method 1. The SOVA output provides the decoder with enough useful information to decode some of the RSW that Paaske's method would require EP4 to decode. For example, in section 4.5, there is an example RS frame decoded using Method 2. Every RSW in the frame contains more than 16 errors. If Paaske's method was used to decode this frame, then EP4 would have to be used to obtain the first decoded RSW. Using Method 2, three RSW were successfully decoded on the first trial, and the others were successfully decoded using information provided by these decoded RSW.

Another possible improvement is the use of iterative decoding. Iterative decoding takes the output of the Reed Solomon decoder and feeds this to the input to the Viterbi decoder. Paaske [7] used iterative decoding in his paper. If there are still undecodable RSW in the deinterleaving frame, the corrected codewords are sent back to the input to the Viterbi decoder. The corrected bits give the Viterbi decoder some of the states that the correct path took. Forcing the Viterbi decoder through these known states may help the Viterbi decoder in a better estimate of the correct path. The error rate at the output will be reduced, which will result in more RSW being decoded on the second trial. Iterative decoding could be used with Methods 2 and 3 presented in this report. There is the potential for larger gains using this repeated Viterbi decoding because the first iteration will have corrected some of the RSW, and provide the second Viterbi trial with additional information. It should be noted that this repeated Viterbi decoding is effective only when the output of the RS decoder contains some correctly decoded RSW. Methods 2 and 3 allow for a reduction in RSW failure for Eb/No values at as low as 1.7 dB SNR.

Even though the reduction at the very low values of Eb/No is not very large, this little bit of reduction would help greatly if used in repeated Viterbi decoding trials.

The systems investigated in this report could be simulated using SPW. SPW is a useful tool for modeling of communication systems. Many signal processing blocks such as filters, modulators, and channel models are contained in SPW's library for use in the design of larger, complex systems. The SOVA and the errors and erasures RS decoder are currently not in any of the libraries found in SPW. SPW does allow for "custom coded blocks". This tool can take C code, and create an SPW block based on this code. This custom coded SPW block can then be used in combination with other SPW blocks in the design of communication systems.

The SOVA could be implemented in a single chip design. The method presented by Hagenauer and Hoeher could be used, or another method presented in [2].

Appendix A

Simulation Flow Charts

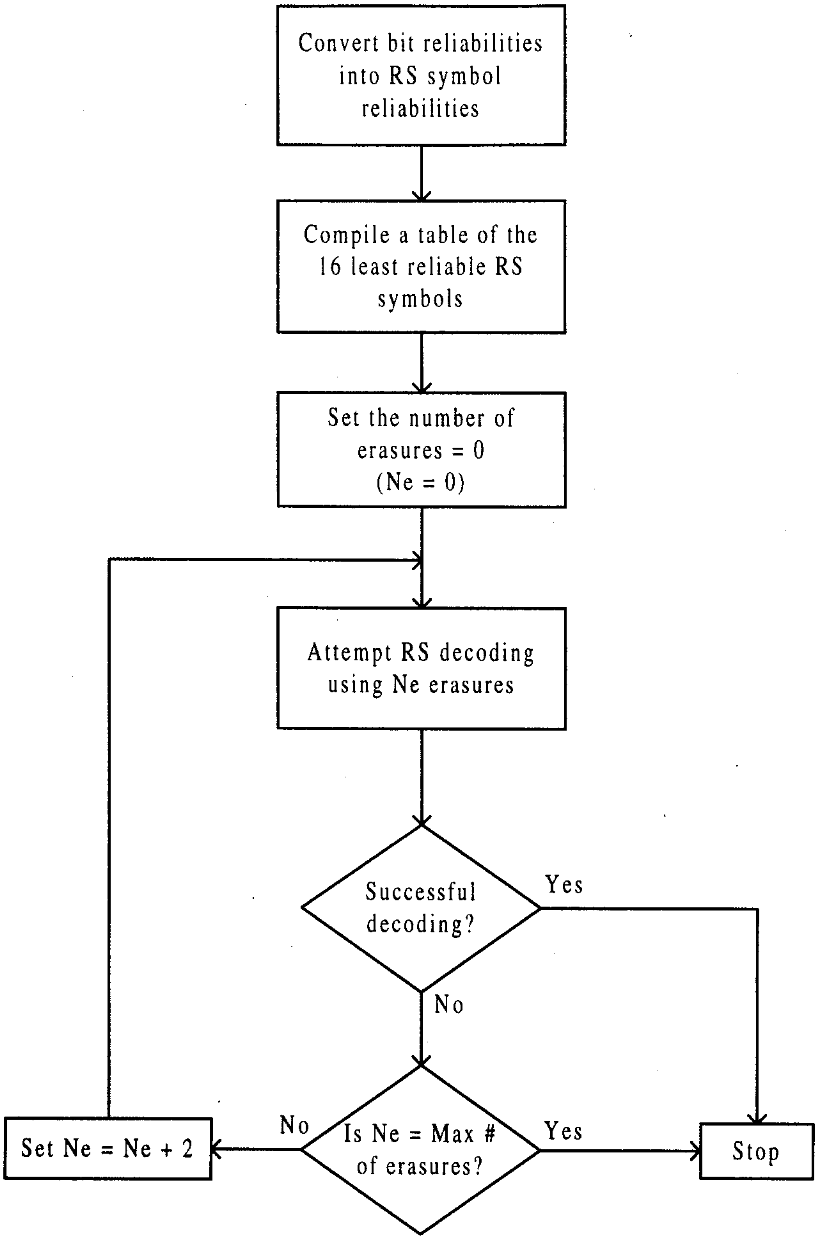

Convert bit reliabilities into RS symbol reliabilities
Compile a table of the 16 least reliable RS symbols
Set the number of erasures = 0 (Ne = 0)
Attempt RS decoding using Ne erasures
Successful decoding?
Yes
No
Is Ne = Max # of erasures?
No
Yes
Set Ne = Ne + 2
Stop

Figure A.1 Flow chart for erasure Method 1

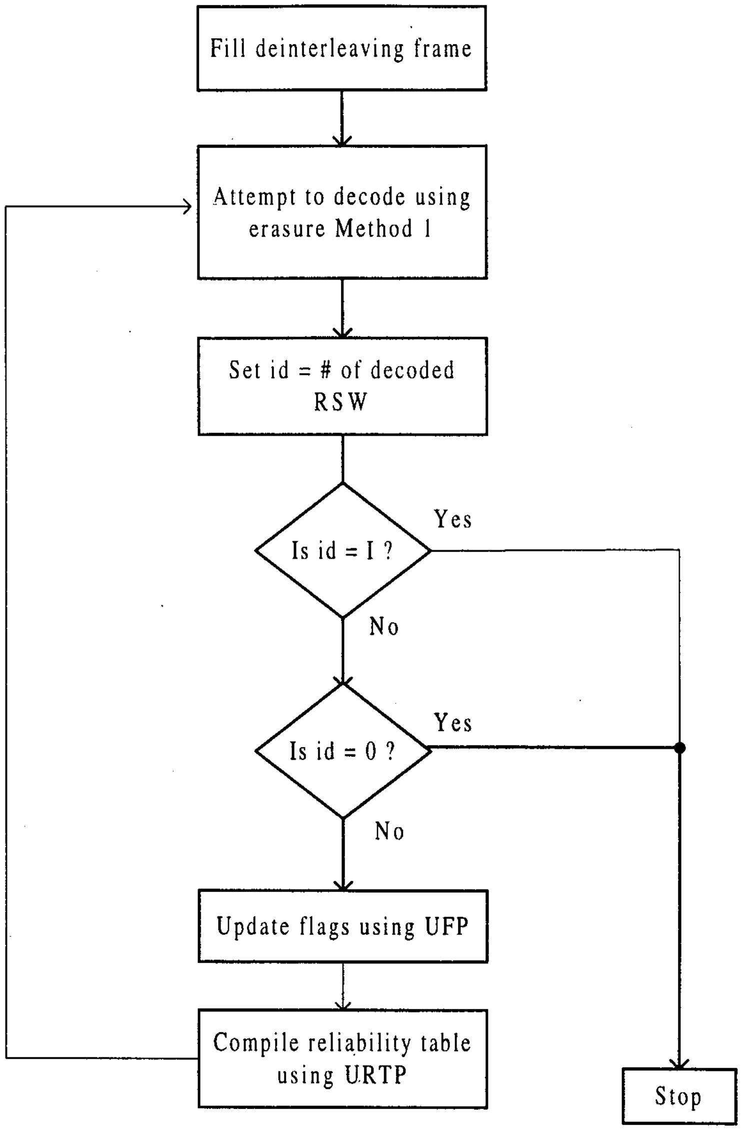

Fill deinterleaving frame
Attempt to decode using erasure Method 1
Set id = # of decoded RSW
Is id = I ?
Yes
No
Is id = 0 ?
Yes
No
Update flags using UFP
Compile reliability table using URTP
Stop

Figure A.2 Flow chart for erasure Method 2

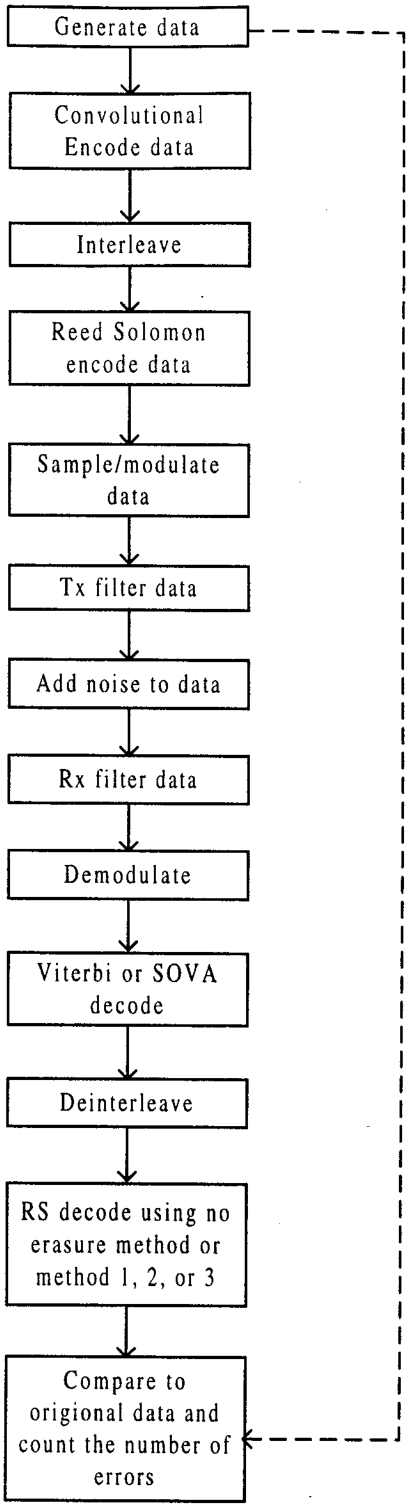


Figure A.3 Flow chart for the real simulation

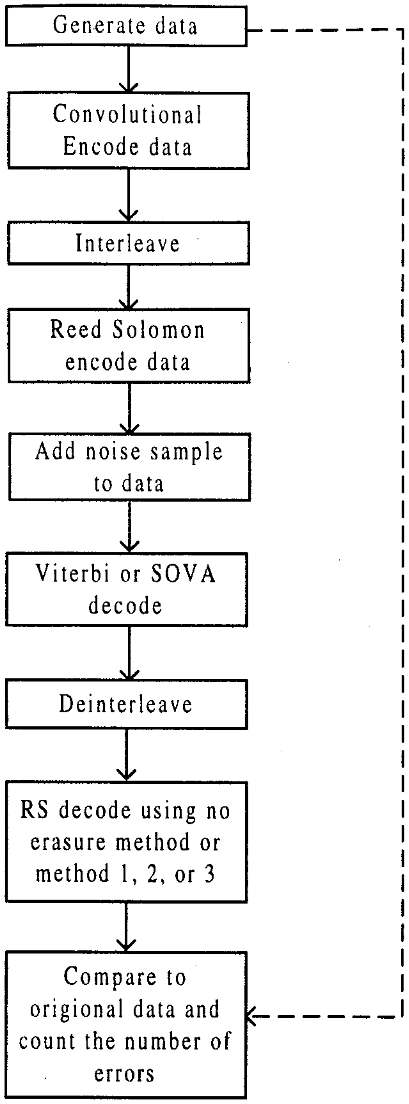

Generate data
Convolutional Encode data
Interleave
Reed Solomon encode data
Add noise sample to data
Viterbi or SOVA decode
Deinterleave
RS decode using no erasure method or method 1, 2, or 3
Compare to origional data and count the number of errors

Figure A.4 Flow chart for the ideal simulation

Appendix B

Program Listing

B.1 Memory allocation functions

```
int *ivector(long nh)
{
        int *v;

        v = (int *)calloc(nh, sizeof(int));
        return v;
}


void free_2d_int_matrix(int row, int **a)
{
        int i;

        for(i=0;i<row;i++)
                free(*a);
        free(a);
}

int **int_matrix_2d(int row, int col)
{
        int i;
        int **a;

        a = (int **)calloc(row, sizeof(int *));
        for(i=0;i<row;i++) {
                a[i] = (int *)calloc(col, sizeof(int));
        }
        return a;
}

double **double_matrix_2d(int row, int col)
{
        int i;
        double **a;

        a = (double **)calloc(row, sizeof(double *));
        for(i=0;i<row;i++) {
                a[i] = (double *)calloc(col, sizeof(double));
        }
        return a;
}

void free_2d_double_matrix(int row, double **a)
{
        int i;

        for(i=0;i<row;i++)
                free(*a);
        free(a);
}


double *dvector(long nh)
{
        double *v;

        v = (double *)calloc(nh, sizeof(double));
        return v;
}
```

B.2 Random number generators

```
double ran2(long *idum)

/* Ran2 is a long period randum number generator ( > 2*10^18).  Returns  */
/* a uniform random deviate between 0.0 and 1.0.  Call with idum a       */
/* negative integer to initialize; thereafter, do idum between successive */
/* deviates in a sequence. This subroutine is taken from the book        */
/* "Numerical recipies in C" by Saul A. Teukolsky, William T. Vetterling, */
/* and Brian P. Flannery.                                        */

{
  int j;
  long k;
  static long idum2 = 123456789;
  static long iy = 0;
  static long iv[NTAB];
  double temp;

  if (*idum <= 0)
  {
    if (-(*idum) < 1) *idum=1;
    else *idum = -(*idum);
    idum2 = (*idum);
    for (j=NTAB+7; j>=0; j--)
    {
          k = (*idum)/IQ1;
          *idum=IA1*(*idum - k*IQ1) - k*IR1;
          if (*idum < 0) *idum += IM1;
          if (j < NTAB) iv[j] = *idum;
    }
    iy = iv[0];
  }
  k = (*idum)/IQ1;
  *idum = IA1*(*idum - k*IQ1) - k*IR1;
  if (*idum < 0) *idum += IM1;
  k = idum2/IQ2;
  idum2 = IA2*(idum2 - k*IQ2) - k*IR2;
  if (idum2 < 0) idum2 += IM2;
  j=iy/NDIV;
  iy = iv[j] - idum2;
  iv[j] = *idum;
  if (iy < 1) iy += IMM1;
  if ((temp = AM*iy) > RNMX) return RNMX;
  else return temp;
}

float gasdev2(long *idum)

/*  This function returns a normally distributed deviate with zero mean and unit variance */
/*  This function is more computationally efficient than gasdev1 because there are no trig */
/*  function calls and the funtion saves the extra deviate for the next funtion call.  This  */
/*  subroutine is taken from the book "Numerical recipies in C" by Saul A. Teukolsky,  */
/* William T. Vetterling, and Brian P. Flannery.                                       */
{
  static int iset = 0;
  static float gset;
  float fac, rsq, v1, v2;

  if(iset == 0)
  {
    do
    {
      v1 = 2.0*ran2(idum) - 1.0;
      v2 = 2.0*ran2(idum) - 1.0;
      rsq = v1*v1+v2*v2;
    }  while (rsq >= 1.0 || rsq == 0.0);
    fac = sqrt(-2.0*log(rsq)/rsq);

    /*  Now make the Box-Muller transformation to get two normal deviates.  */
    /*  Return one and save the other for the next call */

    gset = v1*fac;
    iset = 1;           /*  Set flag  */
    return v2*fac;
  }
  else
  {
    /*  We have an extra deviate handy.  Unset the flag and return it  */
    iset = 0;
    return gset;
  }
}


int bitgen(long *idum)
{
  double a;
  int bit;

  a = ran2(idum);
  if (a >= 0.5)
     bit  = 1;
  else
    bit  = 0;
  return bit;
}


void add_noise(long *idum, double *data, double nn, double sigpow, double EbNo,
           double gain)
{
  int i;
  double  var, sd, A;

  A = gain*pow(10.0, (EbNo/10.0));

  var =  sigpow*Nss/(2*A);

  sd = sqrt(var);

  for (i=0; i<nn; i++)
    data[i] += sd*gasdev2(idum);

}
```

B.3 Filter

```
void filter(double *data, int M, double *h, double *x, long nn)
{

  int i, j;
  double  sum, *x;

  x = dvector(2*M+1);

  for(i=0; i<=2*M; i++)  x[i] = 0.0;
  x[2*M-1] = data[0];
  x[2*M] = data[1];

  for(i=2; i<nn+2*M; i++)
  {
    sum = 0.0;
    for(j=0; j<2*M; j++)
    {
      sum += x[j]*h[j];
      x[j] = x[j+1];
    }
    x[2*M] = data[i];
    data[i-2] = sum;
  }
  free(x);
}
```

B.4 Modulator/Demodulator

```
void modulate (double *data, int *v , long N)

{
  int i, j;
  double A;

  A = sqrt(2.0*Eb/Tb);

  for(i=0;i<N;i++)
  {
    if(v[i]==0)
    {
      for(j=0;j<Nss;j++)
        data[i*Nss + j] = -A;
    }
   else
    {
     for(j=0;j<Nss;j++)
       data[i*Nss + j] = A;
    }
 }

}

void demod(double *data, int *out, long N)

{
  int i;
  double I;

  for(i=0;i<N;i++)
  {

    I =  data[i*Nss+Nss/2 -1]  + data[i*Nss+Nss/2];

    if(I >= 0.0) out[i] = 1;
      else out[i] = 0;

  }

}

void demodsoft(double *data, int *out, long N,  int **soft_metric, int Q, int a)

{
  int i, j, level, **num_bits_per_level;
  double I, b, amp, *prob;

  amp = 2.0;

  num_bits_per_level = int_matrix_2d(2, Q);
  prob = dvector(Q);
  for(i=0; i<2; i++)
  {
    for(j=0; j<Q; j++)
      num_bits_per_level[i][j] = 0;
  }

  for(i=0;i<N;i++)
  {
    I =  (data[i*Nss+Nss/2 -1]  + data[i*Nss+Nss/2])/2.0;

    for(j=0; j<Q; j++)
      if(I>=amp*(2*j-Q)/Q && I<= amp*(2*(j+1)-Q)/Q)
         level = j;

    if( I >= amp) level = Q-1;
    if(I <= -amp)  level = 0;

    num_bits_per_level[out[i]][level] += 1;

    out[i] = level;

  }


    for(i=0; i<Q; i++)
      prob[i] = (double)(num_bits_per_level[0][i] + num_bits_per_level[1][Q-i-1] + 1)/(2*N+Q);

    b = -log(prob[Q-1])/log(2.0);

    for(i=0; i<Q; i++)
    {
      soft_metric[0][i] = (int)(floor)(a*(log(prob[i])/log(2.0) + b));
      soft_metric[1][Q-i-1] = soft_metric[0][i];
    }

    free_2d_int_matrix(2, num_bits_per_level);
    free(prob);
}
```

B.5 Calculate power

```
double calc_power(double  *data, long nn)
{
  int i;
  double  sum, P;

  sum = 0.0;
  for(i=0; i<nn; i++)
    sum +=  pow(data[i], 2);

  P =  sum/(nn);
  return P;
}
```

B.6 Interleaver and deinterleaver

```
void interleave(int *v, int rows, int rs_n, int rs_m, int I, int num)
{
  /*  rows = rs_m*rs_n ,  I  = interleaving depth */

  int ii, i, j, l, **matrix;

  matrix = int_matrix_2d(rows, I);

  /*  Fill matrix by column  */

  for (ii=0; ii<num; ii++)
  {
    for(i=0; i< I; i++)
      for(j=0; j<rows; j++)
        matrix[j][i] = v[ii*rows*I + i*rows +j];

  /*  Exit matrix by row  */

    for (i=0; i<rs_n; i++)
      for(j=0; j<I; j++)
        for(l=0; l<rs_m; l++)
          v[ii*rows*I + i*I*rs_m +j*rs_m + l] = matrix[i*rs_m+l][j];
  }

  free_2d_int_matrix(rows, matrix);

}

void deinterleave(int *v, int rows, int rs_n, int rs_m, int I, int num)
{
  /*  rows = rs_m*rs_n ,  I  = interleaving depth */

  int ii, i, j, l, **matrix;

  matrix = int_matrix_2d(rows, I);


  /*  Fill matrix by row  */

  for (ii=0; ii<num; ii++)
  {
    for (i=0; i<rs_n; i++)
      for(j=0; j<I; j++)
        for(l=0; l<rs_m; l++)
          matrix[i*rs_m+l][j] = v[ii*rows*I + i*I*rs_m +j*rs_m + l] ;

  /*  Exit matrix by column  */
```

```
    for (i=0; i<I; i++)
      for(j=0; j<rows; j++)
        v[ii*rows*I + i*rows +j] = matrix[j][i];
  }

  free_2d_int_matrix(rows, matrix);

}
```

B.7 Convolutional encoder and decoders

```
int bin2dec(int *temp, int a)
{
  int i, sum;

  sum = 0;
  for(i=0; i<a; i++)   sum += temp[i]*pow(2, a-i-1);
  return sum;
}

void  dec2bin( int *temp, int dec, int a)
{
  int i, sum, c;

  sum = dec;
  for(i=0; i<a; i++)
  {
    c = pow(2, a-1-i);
    if (sum >= c)
    {
      temp[i] = 1;
      sum -= c;
    }
    else  temp[i] = 0;
  }
}


void conv_encode( int *u, int n, int k, int m, int num)
{
  int   ii, i, j, l, K, trunc_length, **g, **mem, *out;

  FILE *gen;

  gen = fopen("k1n2m6.dat","r");

  K = m+1;     /*  The constraint length   */
  trunc_length = 100;

  g = int_matrix_2d(n, k*K);
  mem = int_matrix_2d(k, K);
  out = ivector(n*num+trunc_length);
  /*  Obtain the generator matrix  */

  for(i=0; i<n; i++)
    for(j=0; j<K*k; j++)
      fscanf(gen, "%d", &g[i][j]);

  for(ii=0; ii<num+trunc_length; ii++ )
  {
    /*  Shift the contents of memory  */
    for(i=0; i<k; i++)
      for(j= K-2; j>=0; j--)
        mem[i][j+1] = mem[i][j];

    /*   Insert new bits into the encoder  */
    for(i=0; i<k; i++)  mem[i][0] = u[ii*k + i];

    /*   Begin encoding process  */
    for(i=0; i<n; i++)
    {
      out[ii*n + i] = 0;
      for(j=0; j<k; j++)
        for(l=0; l<K; l++)
          out[ii*n + i]  ^=  g[i][j*K + l] & mem[j][l];
    }

  }

  for(i=0; i<n*num; i++)  u[i] = out[i];

  free(out);
  free_2d_int_matrix(n, g);
  free_2d_int_matrix(k, mem);
  fclose(gen);

}


int conv_decode(int *v, int n, int k, int m, int num, int **soft_metric)

{

  int  iii, ii, i, j, l, a, ll, K;
  int  starting_state, part_metric, old_state, flag;
  int  max_metric, max_state, min_metric, num_states, num_inputs;
  int  **g, **mem, **path, **path_next;
  int  **prev_state, **branch_out, *counter, **branch_in;
  int  *metric, *prev_metric, *out, *state;
  int  *input, *new_state, *r, trunc_length;

  FILE *gen;

 trunc_length = 32;
  K = m+1;  /*   The constraint length    */
```

```
num_states = pow(2, m*k);
num_inputs = pow(2, k);

prev_state = int_matrix_2d(num_states, num_inputs);
branch_in = int_matrix_2d(num_states, num_inputs);
branch_out = int_matrix_2d(num_states, num_inputs);
path = int_matrix_2d(num_states, trunc_length);
path_next = int_matrix_2d(num_states, trunc_length);


g = int_matrix_2d(n, k*K);
mem = int_matrix_2d(k, K);
metric = ivector(num_states);
prev_metric = ivector(num_states);
counter = ivector(num_states);
input = ivector(k);
out = ivector(k);
state = ivector(k*m);
new_state = ivector(k*K);
r = ivector(n);

gen = fopen("k1n2m6.dat","r");

/* Obtain the generator matrix */

for(i=0; i<n; i++)
  for(j=0; j<K*k; j++)
  {

    fscanf(gen, "%d", &g[i][j]);

  }

fclose(gen);

for(i=0; i<num_states; i++)  counter[i] = 0;

for(iii=0; iii< num_states; iii++)
{
  for(ii=0; ii< num_inputs; ii++)
  {
    /* Obtain the binary representation of the state of the encoder */
    dec2bin(state, iii, k*m);

    /* Initialize the encoder memory to the current state */
    for(ll=0; ll<k; ll++)
      for(l=0; l<m; l++)
        mem[ll][l] = state[ll*m + l];

    /* Shift the contents of memory */
   for(i=0; i<k; i++)
     for(j= K-2; j>=0; j--)
       mem[i][j+1] = mem[i][j];
```

```
        /*  Obtain the binary representation of the encoder input  */
        dec2bin(input, ii, k);

        /*  Insert new bits into the encoder  */
        for(i=0; i<k; i++)  mem[i][0] = input[i];

        /*  Begin encoding process  */
       for(i=0; i<n; i++)
       {
         out[i] = 0;
         for(j=0; j<k; j++)
           for(l=0; l<K; l++)
              out[i]  ^=  g[i][j*K + l] & mem[j][l];
       }

      /*  Find out the new state of the encoder  */
       for(ll=0; ll<k; ll++)
          for(l=0; l<m; l++)
             new_state[ll*m+l] = mem[ll][l];


       a = bin2dec(new_state, k*m);

        prev_state[a][counter[a]] = iii;
        branch_out[a][counter[a]] = bin2dec(out,n);
        branch_in[iii][ii] = a;
        ++counter[a];
     }
  }     /*  End obtaining the decoder information   */

  starting_state = 0;

  for(i=0; i< num_states; i++)  prev_metric[i] =  -99;
  prev_metric[starting_state] = 0;

  for(iii=0; iii<num+ trunc_length; iii++)
  {

    /* Input the latest word  */

    for(i=0; i<n; i++) r[i] = v[iii*n+i];

    for(i=0; i< num_states; i++)
    {

      metric[i] = -99;
      for(j=0; j< num_inputs; j++)
      {
        part_metric = prev_metric[prev_state[i][j]];
        for(l=0; l<n; l++)
          part_metric +=  soft_metric[((branch_out[i][j] >> l)&1)][r[n-l-1]];

        if(part_metric > metric[i])
    {
      metric[i] = part_metric;
      old_state = prev_state[i][j];
    }
  }

  /*  Now we have the old state and the metric, update the path info */

  /* shift the path information */

  for (l=0; l< trunc_length -1; l++)
    path_next[i][l] = path[old_state][l+1];

  /*  Insert the new branch input into the path  */

  for(l=0; l<num_inputs; l++)
    if(branch_in[old_state][l] == i)
    {
      path_next[i][trunc_length-1] = l;
    }

}

/*  Obtain the output for this iteration if trunc_length bits have entered the buffer */

if(iii >= trunc_length-1)
{
  max_metric = 0;

  /* Determine the path with the maximum metric */

  flag = 0;

  for(i=0; i< num_states; i++)
  {
    if(metric[i] > max_metric)
    {
      max_metric = metric[i];
      max_state = i;
      if(max_metric > 100000) flag = 1;
    }
  }

  if(flag)
  {
    min_metric = max_metric;
    for(i=0; i< num_states; i++)
    {
      if(metric[i] < min_metric)
      {
        min_metric = metric[i];
      }
    }
    for(i=0; i<num_states; i++) metric[i] -= min_metric;
  }
```

```
      /* Obtain the output from the path with the maximum metric */


      for(i=0; i<k; i++)  v[(iii-trunc_length+2)*k-i-1] = ((path_next[max_state][0]>>i)&1);
    }

    for(i=0; i< num_states; i++)  prev_metric[i] = metric[i];

    for(i=0; i<num_states; i++)
      for(j=0; j<trunc_length; j++)
        path[i][j] = path_next[i][j];
  }



  free_2d_int_matrix(num_states, prev_state);
  free_2d_int_matrix(num_states, branch_in);
  free_2d_int_matrix(num_states, branch_out);
  free_2d_int_matrix(num_states, path);
  free_2d_int_matrix(num_states, path_next);
  free_2d_int_matrix(n, g);
  free_2d_int_matrix(k, mem);
  free(metric);
  free(prev_metric);
  free(counter);
  free(input);
  free(state);
  free(new_state);
  free(r);
  free(out);

  return(0);

}



int SOVA(int *v, int n, int k, int m, int num, int **soft_metric, int a1)

{

  int  iii, ii, i, j, l, a, ll, K, s1, s2, m2;
  int  starting_state, part_metric[2], old_state[2], flag;
  long  max_metric, max_state, min_metric, num_states, num_inputs;
  int **g, **mem, **path, **path_next,**prev_state;
  int **branch_out, *conv_counter, **branch_in, *metric;
  int *prev_metric, *out, *state, *input, *new_state, *r;
  double c, delta, **L, **L_next;
  int  L_Q, L_max, nlevels, trunc_length;
```

```
FILE *gen;

gen = fopen("k1n2m6.dat","r");

K = m+1;  /*  The constraint length    */

L_Q = 8;
L_max = 8;
trunc_length = 32;
num_states = pow(2, m*k);
num_inputs = pow(2,k);
nlevels = pow(2, L_Q);

g = int_matrix_2d(n, k*K);
mem = int_matrix_2d(k, K);
metric = ivector(num_states);
prev_metric = ivector(num_states);
conv_counter = ivector(num_states);
input = ivector(k);
out = ivector(n);
state = ivector(k*m);
new_state = ivector(k*K);
r = ivector(n);
prev_state = int_matrix_2d(num_states, num_inputs);
branch_in = int_matrix_2d(num_states, num_inputs);
branch_out = int_matrix_2d(num_states, num_inputs);
path = int_matrix_2d(num_states, trunc_length);
path_next = int_matrix_2d(num_states, trunc_length);
L = double_matrix_2d(num_states, trunc_length);
L_next = double_matrix_2d(num_states, trunc_length);


/*  Obtain the generator matrix  */

for(i=0; i<n; i++)
  for(j=0; j<K*k; j++)
  {

    fscanf(gen, "%d", &g[i][j]);

  }

fclose(gen);

num_states = pow(2, m*k);
num_inputs = pow(2, k);

c = log(2.0)/(double)(a1);

for(i=0; i<num_states; i++)  conv_counter[i] = 0;

for(iii=0; iii< num_states; iii++)
{
  for(ii=0; ii< num_inputs; ii++)
  {
      /*  Obtain the binary representation of the state of the encoder  */
      dec2bin(state, iii, k*m);


      /*  Initialize the encoder memory to the current state  */
      for(ll=0; ll<k; ll++)
        for(l=0; l<m; l++)
          mem[ll][l] = state[ll*m + l];


      /*  Shift the contents of memory  */
     for(i=0; i<k; i++)
       for(j= K-2; j>=0; j--)
         mem[i][j+1] = mem[i][j];



      /*  Obtain the binary representation of the encoder input  */
      dec2bin(input, ii, k);

      /*  Insert new bits into the encoder  */
      for(i=0; i<k; i++)  mem[i][0] = input[i];

      /*  Begin encoding process  */
     for(i=0; i<n; i++)
     {
       out[i] = 0;
       for(j=0; j<k; j++)
         for(l=0; l<K; l++)
           out[i]  ^=  g[i][j*K + l] & mem[j][l];
     }

     /*  Find out the new state of the encoder  */
     for(ll=0; ll<k; ll++)
        for(l=0; l<m; l++)
           new_state[ll*m+l] = mem[ll][l];


     a = bin2dec(new_state, k*m);

      prev_state[a][conv_counter[a]] = iii;
      branch_out[a][conv_counter[a]] = bin2dec(out,n);
      branch_in[iii][ii] = a;
      ++conv_counter[a];
    }
  }    /*  End obtaining the decoder information   */

   starting_state = 0;

   for(i=0; i< num_states; i++)  prev_metric[i] =  -99;
   prev_metric[starting_state] = 0;

   for(i=0; i<num_states; i++)
     for(j=0;j<trunc_length; j++)
       L[i][j] = 99999.0;
```

```
for(iii=0; iii<num+trunc_length; iii++)
{

  /*  Input the latest word  */

  for(i=0; i<n; i++) r[i] = v[iii*n+i];

  for(i=0; i< num_states; i++)
  {

    metric[i] = -99;
    m2 = -99;
    for(j=0; j< num_inputs; j++)
    {
      old_state[j] = prev_state[i][j];
      part_metric[j] = prev_metric[old_state[j]];
      for(l=0; l<n; l++)
        part_metric[j] +=  soft_metric[((branch_out[i][j] >> l)&1)][r[n-l-1]];

    }

    /*  Determine the maximum metric and the second maximum metric */

    for(j=0; j<num_inputs; j++)
    {
      if(part_metric[j] >= metric[i])
      {
        s1 = old_state[j];
        metric[i] = part_metric[j];
        a = j;
      }
    }

    for(j=0; j<num_inputs; j++)
    {
      if (j==a)  continue;
      if(part_metric[j] >= m2)
      {
        s2 = old_state[j];
        m2 = part_metric[j];
      }
    }

     /*  Now we have the old state and the metric, update the path info */

    /*  Calculate delta  */

    delta = (double)c*(metric[i] - m2);

    /*  Update the reliability information  */

    for(j=1; j<=trunc_length-m+1; j++)
    {
      if(path[s1][j] != path[s2][j])
     {
       if(L[s1][j] >= delta)
         L_next[i][j-1] = delta;
       else
         L_next[i][j-1] = L[s1][j];
     }
     else
       L_next[i][j-1] = L[s1][j];

   }

   for(j=trunc_length-m+2; j<trunc_length; j++)
     L_next[i][j-1] = L[s1][j];


   /* shift the path information */

   for (l=0; l<trunc_length-1; l++)
     path_next[i][l] = path[s1][l+1];


   /*  Insert the new branch input into the path  */

   for(l=0; l<num_inputs; l++)
     if(branch_in[s1][l] == i)
     {
       path_next[i][trunc_length-1] = l;
     }
   L_next[i][trunc_length-1] = 10000.0;

 }

/*  Obtain the output for this iteration if trunc_length bits have entered the buffer */

if(L_DEBUG)
{

    printf("\nReliabilities\n");

   for(l=0; l<num_states; l++)
   {
     printf("\n");
     for(i=0; i<trunc_length; i++)
     {
       printf("%3.3lf ", L_next[l][i]);
     }
   }

  if(PAUSE) getchar();

   printf("\nbits in paths\n");

   for(l=0; l<num_states; l++)
   {
     printf("\n");
     for(i=0; i<trunc_length; i++)
     {
       printf("%d ", path_next[l][i]);
     }
   }

  if(PAUSE) getchar();
 }

if(iii >= trunc_length-1)
{

  if((iii-trunc_length+1)%8 == 0)
  {
    max_metric = 0;

    /* Determine the path with the maximum metric */

    flag = 0;

    for(i=0; i< num_states; i++)
    {
      if(metric[i] > max_metric)
      {
        max_metric = metric[i];
        max_state = i;
        if(max_metric > 1000000) flag = 1;
      }
    }


    if(flag)
    {
      min_metric = max_metric;
      for(i=0; i< num_states; i++)
      {
        if(metric[i] < min_metric)
        {
          min_metric = metric[i];
        }
      }
      for(i=0; i<num_states; i++) metric[i] -= min_metric;
    }

   /*  Obtain the output from the path with the maximum metric  */

    for(j=0; j<8; j++)
    {

     for(i=1; i<= nlevels; i++)
     {
       if((L_next[max_state][j] >= (double)(i*L_max/nlevels)) &&
         (L_next[max_state][j] < (double)((i+1)*L_max)/nlevels))
       {
         if(path_next[max_state][j] == 1)  v[iii-trunc_length+1+j] = i;
```

```
             else v[iii-trunc_length+1+j] = -i;


             break;

            }
         }
         if(L_next[max_state][j] > L_max)
         {
           if(path_next[max_state][j] == 1)  v[iii-trunc_length+1+j] = nlevels;
           else v[iii-trunc_length+1+j] = -nlevels;
         }

          if(L_next[max_state][j] < L_max/nlevels)
         {
           if(path_next[max_state][j] == 1)  v[iii-trunc_length+1+j] = 1;
           else v[iii-trunc_length+1+j] = -1;
         }
       }
      }
    }

    for(i=0; i< num_states; i++)  prev_metric[i] = metric[i];

    for(i=0; i<num_states; i++)
      for(j=0; j<trunc_length; j++)
      {
        path[i][j] = path_next[i][j];
        L[i][j] = L_next[i][j];
      }
   }

  free(metric);
  free(prev_metric);
  free(conv_counter);
  free(input);
  free(out);
  free(state);
  free(new_state);
  free(r);
  free_2d_int_matrix(n, g);
  free_2d_int_matrix(k, mem);
  free_2d_int_matrix(num_states, prev_state);
  free_2d_int_matrix(num_states, branch_in);
  free_2d_int_matrix(num_states, branch_out);
  free_2d_int_matrix(num_states, path);
  free_2d_int_matrix(num_states, path_next);
  free_2d_double_matrix(num_states, L);
  free_2d_double_matrix(num_states, L_next);

  return(0);
}
```

B.8 Reed Solomon encoders and decoders

```
void make_GF_table(int m, int *GF_table, int *dec_table, int GF_poly)

/*  This procedure generates the Galois Field GF(2^m).  Information      */
/*  is stored in two lookup tables (dec_table and GF_table).             */
/*                                                                       */
/*  GF_table[i] = base ten number,  where 'i' is the power of alpha      */
/*  dec_table[i] = power of alpha, where 'i' is the base ten repersentation. */
/*                                                                       */
/*  For Example:   In GF(16)                                             */
/*                                                                       */
/*  a^6   =   0 0 1 1   =   3  (Base ten, decimal equivalent)            */
/*                                                                       */
/*      is the same as                                                   */
/*                                                                       */
/*   GF_talble[6] = 3          or       dec_table[3] = 6                 */
/*                                                                       */
/*  These two tables are used to transform a number back and fourth      */
/*  between the two representations (GF and decimal).                    */
/*   This is done in order to perform Galois Field arithmatic.  When     */
/*   adding, it is easier to use the decimal representation, because adding */
/*  two numbers, a and b, is   a^b, where '^' is exclusive-or.          */
/*  If a and b were in GF powers of alpha  representation, then adding   */
/*  the two numbers becomes:                                             */
/*                                                                       */
/*        GF_table[a] ^ GF_table[b]  = c                                 */
/*                                                                       */
/*  Multiplying two numbers is easier to use the powers of alpha         */
/*  representation.  For example,  in GF(16)                             */
/*                        alpha^4 x  alpha^5 = alpha^9                   */
/*  Multiplication is acheived by simply adding the powers of alpha (4+5)*/
/*  This addition is modulus q-1.                                        */
/*                    alpha^14 x alpha^7 = alpha^21 mod (15) = alpha^7   */
/*  Due to the fact that the element zero is not represented in the      */
/*  GF_table because in the powers of alpha representation, 0 is actually */
/*  equal to alpha^0 = 1.  So the GF representation for the element 0 is -1 */
/*  Care has to be taken when multiplying two elements.  The multiplying */
/*  can only take place if a or b is not equal to -1.  Else the result is equal */
/*  to zero.   So  dec_table[0] = -1 ,  but  GF_table[-1] does not equal */
/*  zero.  This returns an error because indixes of arrays can not have  */
/*  negative values.                                                     */
/*                                                                       */
/*  GF_poly is the irriducable (or primitave) polynomial that is used to */
/*  generate the field.  For example, for m = 4, the irriducable         */
/*  polynomial is:                                                       */
/*                                                                       */
/*              X^4  +  X  + 1 = 0     or     X^4  =  X + 1   =   1100  =  12   */
/*                                                                       */
/*  The X^4 = 12 is the way GF_poly is stored.  The field is created by  */
/*  repeated right shifts (multiplication by alpha).  If a '1' is right shifted */
/*  from the right most position (corresponding to alpha^3), we will have */
/*  alpha^4.  But alpha^4 is equal to alpha + 1 ( 1100 = 12), so 12 is   */
/*  'x-or' ed with the result.                                          */


{
  int i, q;

  GF_table[0] = pow(2, (m-1));
  q = pow(2, m);


  for( i=1; i<q-1; i++)
  {
    if (GF_table[i-1] & 1 == 1)
    {
      GF_table[i] = GF_table[i-1] >> 1;
      GF_table[i] = GF_table[i] ^ GF_poly;
    }
    else  GF_table[i] = GF_table[i-1] >> 1;
  }
  for(i=0; i<q-1; i++)   dec_table[GF_table[i]] = i;
  dec_table[0] = -1;
}

void  rs_encode (int *data, int n, int k, int m, int num)

/*  The following procedure encodes a data vector using a Reed Solomon  */
/*  code.  The parameters are defined as follows:                       */
/*                                                                      */
/*    m = number of data bits per symbol                                */
/*    q = 2^m  specifies the field    GF(q) = GF(2^m)                   */
/*    n = q-1  the block length                                         */
/*    t =  maximum number of errors that can be corrected               */
/*    k = n - 2*t    the number of information bits per block           */
/*    num = the total number of blocks to be encoded                    */
/*    data = the uncoded data vector                                    */
/*                                                                      */
/*  The field is first generated by calling the procedure 'make_GF_table'.  */
/*  Then the systematic encoding is accomplished by using feedback shift  */
/*  registers to generate the parity symbols.  The connections are specified  */
/*  by the generator polynomial g[].                                    */

{

  int GF_poly[11] = {0,0,0,6,12,20,48,72,184,272,576};
  int i, ii, j, l, c1, t, sum;
  int *u, *vv, *g, *data1, *GF_table, *dec_table;

  t =  (n-k)/2;

  GF_table = ivector(n+1);
  dec_table = ivector(n+1);
  g = ivector(n-k+1);
  u = ivector(k);
  vv = ivector(n);
  data1 = ivector(n*m*num);
```

```
make_GF_table (m, GF_table, dec_table, GF_poly[m]);

for (i=0; i<= n-k; i++) g[i] = 0;

g[0] = GF_table[1];
g[1] = GF_table[0];

/* Create the generator polynomial  g(X)  */
for (i=2; i<= n-k; i++)
{
  g[i] = GF_table[0];
  for (j=i-1; j>0; j--)
    if (g[j] != 0)   g[j] = g[j-1] ^ GF_table[ (dec_table[g[j]] + i) % n];
    else  g[j] = g[j-1];
  g[0] = GF_table[ (dec_table[g[0]] + i) % n];
}

/* Change g to GF representation  */
for(i=0; i<= n-k; i++)   g[i] = dec_table[g[i]];

for (ii = 0; ii <num; ii++)
{
  for(j = 0; j < k; j++)
  {
    u[j] = 0;
    for (l = 0; l < m; l++)
      u[j] += data[ii*k*m + j*m +l]*pow(2, m-1-l);
  }
  for (i=0; i< n-k; i++) vv[i] = 0;
  for (i=k-1; i>=0; i--)
  {
    c1 = dec_table[u[i]^vv[2*t-1]];
    if (c1 != -1)
    {
      for(j=2*t-1; j>0; j--)
        if (g[j] != -1)  vv[j] = vv[j-1]^GF_table[(g[j]+c1)%n];
        else   vv[j] = vv[j-1];
      vv[0] =  GF_table[(g[0] + c1)%n];
    }
    else
    {
      for (j= n-k-1; j>0; j--)
        vv[j] = vv[j-1];
      vv[0] = 0;
    }
  }
  for(i=0; i<k; i++)   vv[i+n-k] = u[i];

  for(i=0; i<n; i++)
  {
    sum = vv[i];
    for (j=0; j<m; j++)
    {
      c1 = pow(2, m-j-1);
```

```
        if (sum >= c1)
        {
          data1 [ii*n*m + i*m + j]  =  1;
          sum -= c1;
        }
        else  data1 [ii*n*m + i*m + j]  =  0;
      }
    }
  }
  for (i=0; i<n*m*num; i++) data[i] = data1[i];


  free(data1);
  free(GF_table);
  free(dec_table);
  free(vv);
  free(g);
  free(u);

}

void  rs_decode(int *data, int *counter, int n, int k, int q, int m, int num)

/*  The following procedure decodes a Reed Solomon encoded data         */
/*  vector.  This RS decoder has been modified to handle erasures.  An  */
/*  erased position is denoted '-2'.  The code can correct e errors and f */
/*  erasures if  2*e + f <=  dmin  where dmin is equal to n - k or 2*t.  */
/*  The parameters are defined as follows:                              */
/*                                                                      */
/*     m = number of data bits per symbol                               */
/*     q = 2^m  specifies the field    GF(q) = GF(2^m)                  */
/*     n = q-1  the block length                                        */
/*     t =  maximum number of errors that can be corrected              */
/*     k = n - 2*t    the number of information bits per block          */
/*     num = the total number of blocks to be encoded                   */
/*     data = the uncoded data vector                                   */
/*     lambda = The error locator polynomial                            */
/*     S = The syndrome polynomial                                      */
/*                                                                      */
/*  The decoding is done using the Berlekamp-Massey algorithm.  Details */
/*  of the algorithm can be found in "Theory and Practice of Error Control */
/*  Codes" by Richard E. Blahut.  This procedure can handle non-erasure */
/*  decoding also.                                                      */

{
  int  GF_poly[11] = {0,0,0,6,12,20,48,72,184,272,576};
  int  i, ii, j, l, t, rr, sum, L, c1, deg_lambda, num_erasure, delta_r;
  int  erasure, error, decode_flag, num_errors, numer, den;
  int *GF_table, *dec_table, *r, *U, *S, *lambda, *omega;
  int *B, *T, *tmp, *beta;


  t = (n-k)/2;
```

```
GF_table = ivector(q);
dec_table = ivector(q);
beta = ivector(n);
r = ivector(n);
U = ivector(n);
S = ivector(n);
T = ivector(n);
B = ivector(n);
tmp = ivector(n);
lambda = ivector(n);
omega = ivector(2*t+1);


/*  Create GF(2^m) field  */

make_GF_table (m, GF_table, dec_table, GF_poly[m]);


for (ii = 0; ii < num; ii ++)
{

  decode_flag = 0;
  num_erasure = 0;
  for(j = 0; j < n; j++)
  {
    r[j] = 0;
    sum = 0;
    erasure = 0;
    for (l = 0; l < m; l++)
    {
      if (data[ii*n*m + j*m +l] == -2)  /*  Erased bit  */
      {
        U[num_erasure] = j;
        if (erasure == 0)
        {
          num_erasure++;
          erasure = 1;
        }
        r[j] = -1;
        data[ii*n*m + j*m +l] = 0;
        break;
      }
      else    sum += data[ii*n*m + j*m +l]*pow(2, m-1-l);
    }
    if (r[j] != -1)   r[j] = dec_table[sum];
  }
  error = 0;

  /*  Compute the syndrome   */
  for(i=1; i<= n-k; i++)
  {
     S[i] = 0;
     for(j=0; j<n; j++)
        if(r[j] != -1)
          S[i] ^= GF_table[ (r[j] + i*j) % n];
  if (S[i] != 0)  error = 1; /*  If nonzero syndrome, there is an error*/
}
S[0] = 0;

/*  Convert to GF representation  */
for(i=1; i<=n-k; i++)   S[i] = dec_table[S[i]];


if (error)  /*  If the syndrome is equal to zero, no decoding nessasary  */
{
  for(i=0; i < n; i++)
  {
    lambda[i] = 0;
    B[i] = -1;
    T[i] = 0;
  }

  lambda[0] = GF_table[0];    /*   =1  */
  L = 0;
  deg_lambda = 0;
  B[0] = 0;      /*  = 1   */

  for(rr=1; rr <= 2*t; rr++)
  {
     if (rr <= num_erasure)
     {
        for (i=1; i <= deg_lambda+1; i++)
        {
           if (lambda[i-1] != 0)
             tmp[i] = GF_table[ (U[rr-1] + dec_table[ lambda[i-1] ] ) % n];
           else  tmp[i] = 0;
        }
        for (j=1; j <= deg_lambda+1; j++)   lambda[j] ^= tmp[j];
        deg_lambda++;
        for(j=0; j<= deg_lambda; j++)   B[j] = dec_table[lambda[j]];
        L ++;
     }

     else
     {
        /*  Compute the discrepancy  */
        delta_r = 0;
        for(j=0; j<=rr; j++)
          if (lambda[j] != 0  &&  S[rr-j] != -1)
            delta_r ^= GF_table[ (dec_table[lambda[j]] + S[rr-j]) % n];
       delta_r = dec_table[delta_r];

       if (delta_r != -1)
       {
         /*  T(x) <---  lambda(x)  +  delta_r * x * B(x)  */

         for (j=1; j <= deg_lambda +1; j++)
           if (B[j-1] != -1)
             T[j] = lambda[j] ^GF_table[ (delta_r + B[j-1]) % n];
         T[0] = lambda[0];
```

```
      ++deg_lambda ;

      if (2*L <= rr + num_erasure -1)
      {
        L = rr - L + num_erasure;

        /* B(x) <--- B(x) / delta_r */
        for(j=0; j<= deg_lambda; j++)
          if( lambda[j] != 0)
            B[j] = (n-delta_r + dec_table[lambda[j]]) % n ;
          else B[j] = -1;

        /* lambda(x) <---- T(x) */
        for(j=0; j<=deg_lambda+1; j++) lambda[j] = T[j];

      }
      else
      {
        /* lambda(x) <---- T(x) */
        for(j=0; j<n-k; j++) lambda[j] = T[j];

        /* B(x) <----- x * B(x) */
        tmp[0] = -1;
        for(j=1; j<=n-k; j++) tmp[j] = B[j-1];
        for(j=0; j<=n-k; j++) B[j] = tmp[j];
      }
    }

    else
    {
      /* B(x) <----- x * B(x) */
      tmp[0] = -1;
      for(j=1; j<=n-k; j++) tmp[j] = B[j-1];
      for(j=0; j<=n-k; j++) B[j] = tmp[j];
    }
  }
}

/* Change lambda(x) to GF representation */
for(i=0; i<n; i++) lambda[i] = dec_table[lambda[i]];

/* Compute the degree of lambda(x) */
deg_lambda = n;
for(i=n-1; i>=0; i--)
  if (lambda[i] != -1 && deg_lambda == n) deg_lambda = i;

if (deg_lambda <= 2*t) /* Below the capacity of the code */
{

  /* Comupte omega(X) = [1+S(X)] * lambda(X) */

  for(i=0; i<=n-k; i++) omega[i] = 0;
  for (i=0; i<=n-k; i++)
  {
    for (j=0; j<=n-k; j++)
  {
    if ((i+j) >= n-k+1)  continue;
    if (S[i] != -1 && lambda[j] != -1)
    omega[i+j] ^= GF_table[ (S[i] + lambda[j]) % n ];
  }
}

/*  Convert omega(x) to GF representation  */
for (i=0; i<=n-k; i++) omega[i] = dec_table[omega[i]];

/*  Find the roots of lambda(X).  The inverses of the roots gives us the   */
/*  location of the errors.                                                */

num_errors = 0;
for (i=0; i<q-1; i++)
{
  sum = 0;
  for (j=0; j<2*t; j++)
    if (lambda[j] != -1)
      sum ^= GF_table[ (lambda[j] + i*j) % (q-1) ];
  if (sum == 0)
  {
    beta[num_errors] = (n-i)%n;
    num_errors++;
  }
}

if(  (2*(num_errors-num_erasure) + num_erasure) <= 2*t)
{
  if (num_errors == deg_lambda)
  {

    /*  Convert r to base 10  representation */

    for(i=0; i<n; i++)
      if (r[i] != -1)  r[i] = GF_table[r[i]];
      else r[i] = 0;


    /*  Calculate the error values and correct the received vector  */

    for (i=0; i<num_errors; i++)
    {
      /*  Calculate the denominator   */
      den = 0;

      for (j=0; j<=2*t; j++)
      {
        if (j%2 == 0)  continue;
        if (lambda[j] != -1)
        {
          c1 = GF_table[ ( (q-1-beta[i])*(j-1) + lambda[j]) % n  ];
          den ^= c1;
        }
      }
```

```
          den = dec_table[den];

          /*  Calculate the numerator   */
          numer = 0;
          for (j=0; j<=2*t; j++)
            if (omega[j] != -1)  numer ^= GF_table[ ((q-1-beta[i])*j + omega[j])%n ];

          numer = dec_table[numer];

          /*   Correct the erroneous value   */
          if(numer != -1)
            r[beta[i]] ^=  GF_table[ (n + numer + beta[i] - den) % n];

        }

        /*  Change r back into the GF  representation   */

        for(i=0; i<n; i++)   r[i] = dec_table[r[i]];

      }     /*  end if  (num_errors == deg_lambda)  */
      else
      {
        decode_flag = 1;
        counter[3] += 1;
        /*printf("\n3");
        */
      }
    }     /*  end if (num_erasure + num errors <= dmin)  */
    else
    {
      decode_flag = 1;
      counter[2] += 1;
      /*printf("\n2");
      */

    }
  }     /*  end if (deg_lambda <= 2*t)   */
  else
  {
    decode_flag = 1;
    counter[1] += 1;
    /*printf("\n1");
    */

  }
}     /*  end if (error)  */
else
{
  decode_flag = 0;
  counter[0] += 1;
  /*printf("\n1");
  */

}
if (decode_flag)
```

```
    {
      counter[5] += 1;
      /*printf("\n5");
      */

      for(j=0; j<k; j++)
      {
        for(l=0; l<m; l++)
          data[ii*k*m + j*m + l] = data[ii*n*m + (n-k+j)*m + l];
      }
    }
    else
    {
      counter[4] += 1;
      /*printf("\n4");
      */

      for(j=0 ; j<k; j++)
      {
        if (r[n-k+j] == -1)  sum = 0;
        else  sum = GF_table[r[n-k+j]];
        for (l=0; l<m; l++)
        {
          c1 = pow(2, m-l-1);
          if (sum >= c1)
          {
            data [ii*k*m + j*m + l]  =  1;
            sum -= c1;
          }
          else  data [ii*k*m + j*m + l]  =  0;
        }
      }
    }

  }

  free(GF_table);
  free(dec_table);
  free(beta);
  free(r);
  free(U);
  free(S);
  free(T);
  free(B);
  free(tmp);
  free(lambda);
  free(omega);

}


void  rs_decode_erasure_method_1(int *data, int n, int k, int q, int m, int num,
          int *erasure_counter, int *counter)

/*  The Reed Solomon deocder is modified to use erasure Method 1.  This is to be */
/* used with the SOVA.  The bit reliabilities are first converted into symbol        */
/* reliabilities.  A table of the least reliabile symbols are compiled for each       */
/* codeword.  Decoding is attempted using no erasures.  If successful, decoding    */
/* stops.  If unsuccessful, then decoding is reattemped using 2 erasures.  For each */
/*  unsuccessful decoding trial, two more erasures are added until either a          */
/* successful decoding takes place, or a maximum number of erasures has been     */
/* reached (16 for this simulation)                                                              */

{
  int  GF_poly[11] = {0,0,0,6,12,20,48,72,184,272,576};
  int  i, ii, j, l, t, rr, sum, L, c1, deg_lambda, delta_r;
  int  error, decode_flag, num_errors, numer, den;
  int num_erasures, temp, max_eras, *reliability;
  int *GF_table, *dec_table, *r, *U,  *S;
  int *lambda, *omega, *B, *T, *tmp, *beta, *min;

  t = (n-k)/2;

  max_eras = t;


  GF_table = ivector(n+1);
  dec_table = ivector(n+1);
  beta = ivector(n);
  r = ivector(n);
  U = ivector(2*t);
  S = ivector(n);
  T = ivector(n);
  B = ivector(n);
  tmp = ivector(n);
  lambda = ivector(n);
  omega = ivector(2*t+1);
  reliability = ivector(n);
  min = ivector(2*t);


  /* Create GF(2^m) field  */

  make_GF_table (m, GF_table, dec_table, GF_poly[m]);


  for (ii = 0; ii < num; ii ++)
  {

    for(j = 0; j < n; j++)
    {
      r[j] = 0;
      reliability[j] = 500;
      sum = 0;
      for (l = 0; l < m; l++)
      {
        if( reliability[j] > abs(data[ii*n*m + j*m +l]))
          reliability[j] = abs(data[ii*n*m + j*m +l]);

        if(data[ii*n*m + j*m +l] <= 0)
      data[ii*n*m + j*m +l] = 0;
    else
      data[ii*n*m + j*m +l] = 1;
    sum += data[ii*n*m + j*m +l]*pow(2, m-1-l);
  }
  r[j] = dec_table[sum];

}

/*  Find the minimum reliabilities  */

for(i=0; i<max_eras; i++)
{
  min[i] = reliability[i];
  U[i] = i;
}

/*  Do the initial sorting.  Place the minimum in min[0] and the */
/*  maximum in min[max_eras - 1]                                 */

for(i=0; i<max_eras; i++)
{
  for(j=0; j<max_eras -1; j++)
  {
    if(min[j] > min[j+1])
    {
      temp = min[j];
      min[j] = min[j+1];
      min[j+1] = temp;
      temp = U[j];
      U[j] = U[j+1];
      U[j+1] = temp;
    }
  }
}


for(i=max_eras; i<n; i++)
{

 if(reliability[i] < min[max_eras - 1])
 {
   min[max_eras-1] = reliability[i];
   U[max_eras-1] = i;
   for(j=max_eras-1; j>=1; j--)
   {
     if(min[j] < min[j-1])
     {
       temp = min[j];
       min[j] = min[j-1];
       min[j-1] = temp;
       temp = U[j];
       U[j] = U[j-1];
       U[j-1] = temp;
     }
```

```
      else break;
    }


  }
}

/*  The symbols with the minimum reliabilities are in positions  */
/*  U[0] ... U[num_erasures-1].  These symbols will be erased.   */
/*  Set these symbols equal to 0 before computing the syndrome   */


decode_flag = 1;

for(num_erasures=0; num_erasures <= max_eras; num_erasures +=2)
{

  if(decode_flag)
  {

    for(i=0; i<num_erasures; i++)  r[U[i]] = -1;  /*  =0  */

    error = 0;

    /*  Compute the syndrome   */
    for(i=1; i<= n-k; i++)
    {
      S[i] = 0;
      for(j=0; j<n; j++)
        if(r[j] != -1)
          S[i] ^= GF_table[ (r[j] + i*j) % n];
      if (S[i] != 0)  error = 1; /*  If nonzero syndrome, there is an error*/
    }
    S[0] = 0;

    /*  Convert to GF representation  */
    for(i=1; i<=n-k; i++)   S[i] = dec_table[S[i]];

    if (error)   /*  If the syndrome is equal to zero, no decoding nessasary  */
    {
      for(i=0; i < n; i++)
      {
        lambda[i] = 0;
        B[i] = -1;
        T[i] = 0;
      }

      lambda[0] = GF_table[0];    /*   =1  */
      L = 0;
      deg_lambda = 0;
      B[0] = 0;      /*  = 1   */

      for(rr=1; rr <= 2*t; rr++)
      {
        if (rr <= num_erasures)
```

```
{
  for (i=1; i <= deg_lambda+1; i++)
  {
    if (lambda[i-1] != 0)
      tmp[i] = GF_table[(U[rr-1]+dec_table[lambda[i-1]])%n];
    else  tmp[i] = 0;
  }
  for (j=1; j <= deg_lambda+1; j++)   lambda[j] ^= tmp[j];
  deg_lambda++;
  for(j=0; j<= deg_lambda; j++)   B[j] = dec_table[lambda[j]];
  L ++;
}

else
{
  /* Compute the discrepancy */
  delta_r = 0;
  for(j=0; j<=rr; j++)
    if (lambda[j] != 0  &&  S[rr-j] != -1)
      delta_r ^= GF_table[(dec_table[lambda[j]]+S[rr-j])%n];
  delta_r = dec_table[delta_r];

  if (delta_r != -1)
  {
    /*  T(x) <---  lambda(x)  +  delta_r * x * B(x)  */

    for (j=1; j <= deg_lambda +1; j++)
      if (B[j-1] != -1)
        T[j] = lambda[j] ^GF_table[ (delta_r + B[j-1]) % n];
    T[0] = lambda[0];
    ++deg_lambda;

    if (2*L <= rr + num_erasures-1)
    {
      L = rr - L + num_erasures;

      /* B(x) <---  B(x) /  delta_r  */
      for(j=0; j<= deg_lambda; j++)
        if(  lambda[j] != 0)
          B[j] = (n-delta_r+dec_table[lambda[j]])%n;
        else  B[j] = -1;

      /* lambda(x) <----  T(x)   */
      for(j=0; j<=deg_lambda+1; j++)  lambda[j] = T[j];

    }
    else
    {
      /* lambda(x) <----  T(x)   */
      for(j=0; j<n-k; j++)   lambda[j] = T[j];

      /* B(x)  <-----  x * B(x)   */
      tmp[0] = -1;
      for(j=1; j<=n-k; j++)   tmp[j] = B[j-1];
      for(j=0; j<=n-k; j++)   B[j] = tmp[j];
      }
    }

    else
    {
      /* B(x) <----- x * B(x)  */
      tmp[0] = -1;
      for(j=1; j<=n-k; j++)  tmp[j] = B[j-1];
      for(j=0; j<=n-k; j++)  B[j] = tmp[j];
    }
  }
}

/* Change lambda(x) to GF representation */
for(i=0; i<n; i++)  lambda[i] = dec_table[lambda[i]];

/* Compute the degree of lambda(x)  */
deg_lambda = n;
for(i=n-1; i>=0; i--)
  if (lambda[i] != -1 && deg_lambda == n) deg_lambda = i;

if (deg_lambda <= 2*t)  /* Below the capacity of the code  */
{

  /*   Comupte omega(X) = [1+S(X)] * lambda(X)   */

  for(i=0; i<=n-k; i++)  omega[i] = 0;
  for (i=0; i<=n-k; i++)
  {
    for (j=0; j<=n-k; j++)
    {
      if ((i+j) >= n-k+1)  continue;
      if (S[i] != -1 && lambda[j] != -1)
      omega[i+j] ^= GF_table[ (S[i] + lambda[j]) % n ];
    }
  }

  /*  Convert omega(x) to GF representation */
  for (i=0; i<=n-k; i++) omega[i] = dec_table[omega[i]];

  /*  Find the roots of lambda(X). The inverses of the roots gives us the   */
  /*  location of the errors.                                               */

  num_errors = 0;
  for (i=0; i<q-1; i++)
  {
    sum = 0;
    for (j=0; j<2*t; j++)
      if (lambda[j] != -1)
        sum ^= GF_table[ (lambda[j] + i*j) % (q-1) ];
    if (sum == 0)
    {
      beta[num_errors] = (n-i)%n;
      num_errors++;
    }
```

```
}

if(  (2*(num_errors-num_erasures) + num_erasures) <= 2*t)
{
  if (num_errors == deg_lambda)
  {

    /*  Correct Decoding.  record the umber of erasures required */

    erasure_counter[num_erasures] += 1;

    decode_flag = 0;

    /* Convert r to base 10 representation */

    for(i=0; i<n; i++)
      if (r[i] != -1)  r[i] = GF_table[r[i]];
      else r[i] = 0;

    /* Calculate the error values and correct the received vector  */

    for (i=0; i<num_errors; i++)
    {
      /*   Calculate the denominator   */
      den = 0;

      for (j=0; j<=2*t; j++)
      {
        if (j%2 == 0)  continue;
        if (lambda[j] != -1)
        {
          c1 = GF_table[ ( (q-1-beta[i])*(j-1) + lambda[j]) % n  ];
          den ^= c1;
        }
      }

      den = dec_table[den];

      /*   Calculate the numerator   */
      numer = 0;
      for (j=0; j<=2*t; j++)
        if (omega[j] != -1)  numer ^= GF_table[ ((q-1-beta[i])*j + omega[j])%n ];

      numer = dec_table[numer];

      /*    Correct the erroneous value   */
      if(numer != -1)
        r[beta[i]] ^= GF_table[(n+numer+beta[i]-den)%n];

    }

    /*   Change r back into the GF  representation   */

    for(i=0; i<n; i++)   r[i] = dec_table[r[i]];
```

```
          }     /*  end if  (num_errors == deg_lambda)  */
          else
          {
            decode_flag = 1;
            if(num_erasures == max_eras) counter[3] += 1;

          }
        }     /*  end if (num_erasures + num errors <= dmin)  */
        else
        {
          decode_flag = 1;
          if(num_erasures == max_eras) counter[2] += 1;

        }
      }     /*  end if (deg_lambda <= 2*t)    */
      else
      {
        decode_flag = 1;
        if(num_erasures == max_eras) counter[1] += 1;
      }
    }     /*  end if (error)  */
    else
    {
      erasure_counter[num_erasures] += 1;
      decode_flag = 0;
      counter[0] += 1;
    }

  }  /* end if decode flag  */
}  /* end for num_erasures = 0 to max_eras  */


if (decode_flag)
{
  /*  Incorrect decoding.  */

  counter[5] += 1;

   for(j=0; j<k; j++)
   {
     for(l=0; l<m; l++)
        data[ii*k*m + j*m + l] = data[ii*n*m + (n-k+j)*m + l];
   }
}
else
{
  counter[4] += 1;

  /*  Correct decoding  */

  for(j=0 ; j<k; j++)
  {
    if (r[n-k+j] == -1)  sum = 0;
    else  sum = GF_table[r[n-k+j]];
    for (l=0; l<m; l++)
```

```
        {
          c1 = pow(2, m-l-1);
          if (sum >= c1)
          {
            data [ii*k*m + j*m + l]  =  1;
            sum -= c1;
          }
          else  data [ii*k*m + j*m + l]  =  0;
        }
      }
    }

  }

  free(GF_table);
  free(dec_table);
  free(reliability);
  free(min);
  free(beta);
  free(r);
  free(U);
  free(S);
  free(T);
  free(B);
  free(tmp);
  free(lambda);
  free(omega);
}


void  rs_decode_erasure_method_2(int *data, int *v2, int *counter, int n, int k, int q, int m, int num,
          int *erasure_counter, int I, int *decoded_word,
          int *frame_failure, int *error_stat1, int *error_stat2)

/*  Reed Solomon decoding is performed using erasure method 2.  In this method, the deinterleaving */
/*  frame is reconstructed.  Decoding of each RSW is initially attempted using erasure method 1.     */
/*  The flags for each undecoded RSW are updated using information provided by the successfully     */
/*  decoded RSW in the frame.                                                                        */
{
  int  GF_poly[11] = {0,0,0,6,12,20,48,72,184,272,576};
  int  i, ii, j, l, ll, t, rr, sum, sum2, L, c1, deg_lambda, delta_r;
  int  error, decode_flag, num_errors, numer, den, I_flag;
  int num_erasures, temp, max_eras, max_eras1, cc, failure, ne;
  int *r, *r2, *U, *S, *lambda, *omega, *B, *T, *tmp, *beta;

  struct mat **RSM;

  int jj, kk, reli, GEBE, offset, GE_count, r_count, max_iter, id, id1;
  int *GF_table, *dec_table;

  GF_table = ivector(n+1);
  dec_table = ivector(n+1);

  t = (n-k)/2;
```

```
beta = ivector(n);
r = ivector(n);
r2 = ivector(n);

U = ivector(2*t);
S = ivector(n);
T = ivector(n);
B = ivector(n);
tmp = ivector(n);
lambda = ivector(n);
omega = ivector(2*t+1);
counter = ivector(20);

max_eras1 = 16;
max_iter = I;

/*  Create GF(2^m) field  */

make_GF_table (m, GF_table, dec_table, GF_poly[m]);

 RSM = (struct mat **)calloc(n*m, sizeof(struct mat *));
 for(i=0;i<n*m;i++)
 {
        RSM[i] = (struct mat *)calloc(I, sizeof(struct mat));
 }


for (ii = 0; ii < num/I; ii ++)
{

  /*  Enter the data into the matrix  */

  counter[13] ++;  /* Total number of frames */

  I_flag = 0;

  for(i=0; i<I; i++)
  {
    id = 0;
    decoded_word[i] = 0;
    ne = 0;
    counter[0]++;  /* number of RSW */

    for(j = 0; j < n; j++)
    {
      RSM[j][i].r = 0;
      RSM[j][i].r2 = 0;
      RSM[j][i].reli = 500;
      RSM[j][i].flag = 0;
      sum = 0;
      sum2 = 0;
      for (l = 0; l < m; l++)
      {
        if( RSM[j][i].reli > abs(data[ii*I*n*m + i*n*m + j*m +l]))
          RSM[j][i].reli = abs(data[ii*I*n*m + i*n*m + j*m +l]);
        if(data[ii*I*n*m + i*n*m + j*m +l] <= 0)
          data[ii*I*n*m + i*n*m + j*m +l] = 0;
        else
          data[ii*I*n*m + i*n*m + j*m +l] = 1;

        sum += data[ii*I*n*m + i*n*m + j*m +l]*pow(2, m-1-l);
        sum2 += v2[ii*I*n*m + i*n*m + j*m +l]*pow(2, m-1-l);
      }
      RSM[j][i].r = dec_table[sum];
      RSM[j][i].r2 = dec_table[sum2];

      ++counter[11]; /*  Total number of bytes  */

      if(sum != sum2)
      {
        ++ne;
        counter[12] ++; /* Number of byte errors  */
      }

    }
    if(ne >32) ne = 32;

    if(ne <=16) ne = 16;
    else  I_flag = 1;


    ++ error_stat1[ne-16];
  }

  if(I_flag) counter[14] ++; /* Number of frame errors */


  /*  If GEN_STAT = 1, then the probability of flagging a BE given a GE */
  /*  and the probability of flagging a GE given a BE  */

  if(GEN_STAT)
  {
    for(i=0; i<I; i++)
    {
      for(j=0; j<n; j++)
      {
        reli = RSM[j][i].reli;
        if(reli >= 255) continue;

        if(RSM[j][i].r == RSM[j][i].r2) GEBE = 2;
        else GEBE = 1;

        kk = j;
        offset = 0;

        /*  loop forward looking for identical reliabilities  */

        for(l=0; l<I; l++)
        {
```

```
  if(i+l>=I && kk == n-1) break;
  if(i+l>=I)
  {
    kk = j+1;
    offset = I;
  }
  if(RSM[kk][i+l-offset}.reli == reli)
  {
    if(GEBE == 1)
    {
      counter[6]++;
      if(RSM[kk][i+l-offset].r == RSM[kk][i+l-offset].r2)
        counter[5]++;
      else counter[4]++;
    }
    if(GEBE == 2)
    {
      counter[9]++;
      if(RSM[kk][i+l-offset].r == RSM[kk][i+l-offset].r2)
        counter[7]++;
      else counter[8]++;
    }
  }
  else break;
}

/*  loop backwards looking for identical identities  */

kk = j;
offset = 0;

for(l=0; l<I; l++)
{
  if(i-l < 0 && kk == 0)  break;
  if(i-l < 0)
  {
    kk = j-1;
    offset = I;
  }
  if(RSM[kk][i-l+offset].reli == reli)
  {
    if(GEBE == 1)
    {
      counter[6]++;
      if(RSM[kk][i-l+offset].r == RSM[kk][i-l+offset].r2)
        counter[5]++;
      else counter[4]++;
    }
    if(GEBE == 2)
    {
      counter[9]++;
      if(RSM[kk][i-l+offset].r == RSM[kk][i-l+offset].r2)
        counter[7]++;
      else counter[8]++;
    }
```

```
        }
        else break;
      }
    }  /*  end loop j = 0 to n */
  }  /*  end loop i = 0 to I */
}  /*  end if(GEN_STAT)  */


if(DEBUG_DUMP)
{
  printf("\nDecoded Word\n\n");
  for(i=0; i<I; i++) printf("%d\t",decoded_word[i]);

  printf("\n");
  for(i=0; i<n; i++)
  {
    printf("\n%d  ",i);
    for(j=0; j<I; j++)
    {
      if(RSM[i][j].r != RSM[i][j].r2)
        printf("(%d, GE)\t",RSM[i][j].reli);
      else
        printf("(%d, BE)\t",RSM[i][j].reli);
    }
  }
 if(PAUSE) getchar();
}


if(DEBUG1)
{

  printf("\nReceived word\n");

  for(i=0; i<I; i++)
  {
    printf("\n");
    for(j=0; j<n; j++)
    {
      printf(" %d",RSM[j][i].r);
    }
  }

 if(PAUSE) getchar();


  printf("\nCorrect word\n");

  for(i=0; i<I; i++)
  {
    printf("\n");
    for(j=0; j<n; j++)
    {
      printf(" %d",RSM[j][i].r2);
    }
    }

  if(PAUSE) getchar();


   printf("\nReliability\n");

   for(i=0; i<I; i++)
   {
     printf("\n");
     for(j=0; j<n; j++)
     {
       printf(" %d",RSM[j][i].reli);
     }
   }

  if(PAUSE) getchar();


   printf("\nError flag\n");

   for(i=0; i<I; i++)
   {
     printf("\n");
     for(j=0; j<n; j++)
     {
       printf(" %d",RSM[j][i].flag);
     }
   }

  if(PAUSE) getchar();
} /* end if (DEBUG1) */

id1 = 0;
id = 2;

/* Attempt the iteritive decoding */

for(jj=0; jj<max_iter; jj++)
{

  /* Obtain the reliability info from the enibhoring codewords from */
  /* the interleaver and/or SOVA */

  if(jj==0) max_eras = 16;
  else max_eras = max_eras1;

  if(id1==id) break;

  id = id1;

  if(DEBUG)
  {
    printf("\ndecoded word = ");
    for(i=0; i<I; i++) printf("%d ", decoded_word[i]);
```

```
}

if(id == I) break;

for(i=0; i<I; i++)
{

  if(decoded_word[i] != 1) continue;

  for(j=0; j<n; j++)
  {
    reli = RSM[j][i].reli;
    if(reli >= 255) continue;
    GEBE = RSM[j][i].flag;

    kk = j;
    offset = 0;

    /*  loop forward looking for identical reliabilities  */

    for(l=0; l<I; l++)
    {
      if(i+l>=I && kk == n-1) break;
      if(i+l>=I)
      {
        kk = j+1;
        offset = I;
      }
      if(RSM[kk][i+l-offset].reli == reli)
      {
        if(decoded_word[i+l-offset] == 0)
          RSM[kk][i+l-offset].flag = GEBE;
      }
      else break;
    }

    /*  loop backwards looking for identical identities  */

    kk = j;
    offset = 0;

    for(l=0; l<I; l++)
    {
      if(i-l < 0 && kk == 0)  break;
      if(i-l < 0)
      {
        kk = j-1;
        offset = I;
      }
      if(RSM[kk][i-l+offset].reli == reli)
      {
        if(decoded_word[i-l+offset] == 0)
          RSM[kk][i-l+offset].flag = GEBE;

      }
      else break;
    }
  } /*  end loop j = 0 to n */

  decoded_word[i] = 2;  /*  flag this word as already giving GEBE info  */
}

/*  Done revising reliability info  */

/*  Start prioritising the reliability info, and then begin decoding */

for(ll = 0; ll<I; ll++)  /*  Find good erasures (GE) first */
{
  GE_count = 0;

  if(DEBUG)
  {

    printf("\ndecoded word =  ");
    for(i=0; i<I; i++) printf("%d ", decoded_word[i]);
  }

  if(decoded_word[ll] != 0) continue;

  for(i=0; i<n; i++)
  {
    if(RSM[i][ll].flag == 1)
    {
      U[GE_count] = i;
      GE_count ++;
    }
  }

  if(GE_count < max_eras)
  {
    r_count = 0;
    for(i=0; i<n; i++)
    {
      if(GE_count + r_count < max_eras)
      {
        if(RSM[i][ll].flag == 0)
        {
          U[GE_count + r_count] = i;
          r_count++;
        }
      }

      if(GE_count + r_count == max_eras)
      {
        for(j=GE_count; j<max_eras; j++)
        {
          for(l=GE_count; l<max_eras-1; l++)
          {
            if(RSM[U[l]][ll].reli > RSM[U[l+1]][ll].reli)
            {
```

```
            temp = U[l];
            U[l] = U[l+1];
            U[l+1] = temp;
          }
        }
      }
      r_count ++;
    }

    if(GE_count + r_count > max_eras)
    {
      if(RSM[U[max_eras-1]][ll].reli > RSM[i][ll].reli
        && RSM[i][ll].flag == 0)
      {

        U[max_eras-1] = i;

        for(j=max_eras-1; j>=GE_count+1; j--)
        {
          if(RSM[U[j]][ll].reli < RSM[U[j-1]][ll].reli)
          {
            temp = U[j];
            U[j] = U[j-1];
            U[j-1] = temp;
          }
          else break;
        }
      }
    } /*  End if(GE_count + r_count > max_eras)  */

  } /*  End loop  i = 0 .. n  */
} /*  End loop if(GE_count < max_eras)  */

/*  Now we have reliabilities try to decode  */


if(DEBUG)
{
  printf("\nMinimum reliabilities and positions\n");
  for(i=0; i<max_eras; i++) printf("(%d, %d)",U[i], RSM[U[i]][ll].reli);
  printf("\n");
  printf("\nnumber of GE = %d", GE_count);
  cc = 0;
  for(i=0; i<n; i++)
  {
    if(RSM[i][ll].r != RSM[i][ll].r2)
    {
      cc += 1;
      printf("\n%d error at %d  Reliability = %d", cc, i, RSM[i][ll].reli);
      for(j=0; j<max_eras; j++)
      {
        if(i == U[j])  printf(" min %d", j);
      }
    }
```

```
   }
  if(PAUSE) getchar();
 } /*  end if (DEBUG)  */

for(i=0; i<n; i++) r[i] = RSM[i][ll].r;

/*  Begin decoding.  Try with 0 erasures, and keep on adding 2 */
/*  until decodes properly, or until maximum is reached        */


/*  The symbols with the minimum reliabilities are in positions  */
/*  U[0] ... U[num_erasures-1].  These symbols will be erased.  */
/*  Set these symbols equal to 0 before computing the syndrome   */


decode_flag = 1;

for(num_erasures=0; num_erasures <= max_eras; num_erasures +=2)
{


  if(decode_flag)
  {

           counter[15] ++;  /* Number of decoding trials  */

    for(i=0; i<num_erasures; i++)  r[U[i]] = -1;  /*  =0  */

    error = 0;

    /*  Compute the syndrome   */
    for(i=1; i<= n-k; i++)
    {
      S[i] = 0;
      for(j=0; j<n; j++)
        if(r[j] != -1)
          S[i] ^= GF_table[ (r[j] + i*j) % n];
      if (S[i] != 0)  error = 1; /*  If nonzero syndrome, there is an error*/
    }
    S[0] = 0;

    /*  Convert to GF representation  */
    for(i=1; i<=n-k; i++)   S[i] = dec_table[S[i]];

    if (error)   /*  If the syndrome is equal to zero, no decoding nessasary  */
    {
      for(i=0; i < n; i++)
      {
        lambda[i] = 0;
        B[i] = -1;
        T[i] = 0;
      }

      lambda[0] = GF_table[0],    /*   =1  */
      L = 0;
```

```
deg_lambda = 0;
B[0] = 0;     /* = 1   */

for(rr=1; rr <= 2*t; rr++)
{
  if (rr <= num_erasures)
  {
    for (i=1; i <= deg_lambda+1; i++)
    {
      if (lambda[i-1] != 0)
        tmp[i] = GF_table[(U[rr-1]+dec_table[lambda[i-1]])%n];
      else  tmp[i] = 0;
    }
    for (j=1; j <= deg_lambda+1; j++)   lambda[j] ^= tmp[j];
    deg_lambda++;
    for(j=0; j<= deg_lambda; j++)   B[j] = dec_table[lambda[j]];
    L ++;
  }

  else
  {
    /*  Compute the discrepancy  */
    delta_r = 0;
    for(j=0; j<=rr; j++)
      if (lambda[j] != 0  &&  S[rr-j] != -1)
        delta_r ^= GF_table[(dec_table[lambda[j]]+S[rr-j])%n];
    delta_r = dec_table[delta_r];

    if (delta_r != -1)
    {
      /*   T(x) <---  lambda(x)  +  delta_r * x * B(x)  */

      for (j=1; j <= deg_lambda +1; j++)
        if (B[j-1] != -1)
          T[j] = lambda[j] ^GF_table[ (delta_r + B[j-1]) % n];
      T[0] = lambda[0];
      ++deg_lambda;

      if (2*L <= rr + num_erasures-1)
      {
        L = rr - L + num_erasures;

        /*  B(x) <---  B(x) /  delta_r  */
        for(j=0; j<= deg_lambda; j++)
          if(  lambda[j] != 0)
            B[j] = (n-delta_r+dec_table[lambda[j]])%n;
          else  B[j] = -1;

        /*  lambda(x) <----  T(x)   */
        for(j=0; j<=deg_lambda+1; j++)  lambda[j] = T[j];

      }
      else
      {
        /*  lambda(x) <----  T(x)   */
        for(j=0; j<n-k; j++)   lambda[j] = T[j];

        /*  B(x)  <-----  x * B(x)   */
        tmp[0] = -1;
        for(j=1; j<=n-k; j++)   tmp[j] = B[j-1];
        for(j=0; j<=n-k; j++)   B[j] = tmp[j];
      }
    }

    else
    {
      /*  B(x)  <-----  x * B(x)   */
      tmp[0] = -1;
      for(j=1; j<=n-k; j++)   tmp[j] = B[j-1];
      for(j=0; j<=n-k; j++)   B[j] = tmp[j];
    }
  }
}

/*  Change lambda(x) to GF representation  */
for(i=0; i<n; i++)  lambda[i] = dec_table[lambda[i]];

/*  Compute the degree of lambda(x)   */
deg_lambda = n;
for(i=n-1; i>=0; i--)
  if (lambda[i] != -1 && deg_lambda == n) deg_lambda = i;

if (deg_lambda <= 2*t)  /*  Below the capacity of the code  */
{

  /*    Comupte omega(X) = [1+S(X)] * lambda(X)   */

  for(i=0; i<=n-k; i++)  omega[i] = 0;
  for (i=0; i<=n-k; i++)
  {
    for (j=0; j<=n-k; j++)
    {
      if ((i+j) >= n-k+1)  continue;
      if (S[i] != -1 && lambda[j] != -1)
      omega[i+j] ^= GF_table[ (S[i] + lambda[j]) % n ];
    }
  }

  /*  Convert omega(x) to GF representation  */
  for (i=0; i<=n-k; i++) omega[i] = dec_table[omega[i]];

  /*  Find the roots of lambda(X). The inverses of the roots gives us the  */
  /*  location of the errors.                                               */

  num_errors = 0;
  for (i=0; i<q-1; i++)
  {
    sum = 0;
    for (j=0; j<2*t; j++)
      if (lambda[j] != -1)
      sum ^= GF_table[ (lambda[j] + i*j) % (q-1) ];
  if (sum == 0)
  {
     beta[num_errors] = (n-i)%n;
    num_errors++;
  }
}

if(  (2*(num_errors-num_erasures) + num_erasures) <= 2*t)
{
  if (num_errors == deg_lambda)
  {

    /*  Correct Decoding.  record the umber of erasures required */

    erasure_counter[num_erasures] += 1;

    decode_flag = 0;

    /* Convert r to base 10 representation */

    for(i=0; i<n; i++)
      if (r[i] != -1)  r[i] = GF_table[r[i]];
      else r[i] = 0;

    /* Calculate the error values and correct the received vector  */

    for (i=0; i<num_errors; i++)
    {
      /*   Calculate the denominator   */
      den = 0;

      for (j=0; j<=2*t; j++)
      {
        if (j%2 == 0)  continue;
        if (lambda[j] != -1)
        {
          c1 = GF_table[ ( (q-1-beta[i])*(j-1) + lambda[j]) % n  ];
          den ^= c1;
        }
      }

      den = dec_table[den];

      /*   Calculate the numerator   */
      numer = 0;
      for (j=0; j<=2*t; j++)
        if (omega[j] != -1)  numer ^= GF_table[ ((q-1-beta[i])*j + omega[j])%n ];

      numer = dec_table[numer];

      /*    Correct the erroneous value   */
      if(numer != -1)
        r[beta[i]] ^= GF_table[(n+numer+beta[i]-den)%n];
```

```
            }

           /*   Change r back into the GF  representation   */

            for(i=0; i<n; i++)   r[i] = dec_table[r[i]];

          }     /*  end if  (num_errors == deg_lambda)  */
          else
          {
            decode_flag = 1;

          }
        }     /*  end if (num_erasures + num errors <= dmin)  */
        else
        {
          decode_flag = 1;

        }
      }     /*  end if (deg_lambda <= 2*t)   */
      else
      {
        decode_flag = 1;
      }
    }     /*  end if (error)  */
    else
    {
      erasure_counter[num_erasures] += 1;
      decode_flag = 0;
     }

  } /* end if decode flag  */


} /* end for num_erasures = 0 to max_eras  */


if (!decode_flag)
{
  /*  Correct decoding.  */


   decoded_word[ll] = 1;
   id1 +=1;

   for(i=0; i<n; i++)
   {
     if(RSM[i][ll].r != RSM[i][ll].r2) RSM[i][ll].flag = 1; /* GE */
     else RSM[i][ll].flag = 2;
   }

   if(DEBUG)
   {
     printf("\nWord %d decoded correctly", ll);
```

```
          if(PAUSE) getchar();
        }

        failure = 0;
        for(i=0; i<n; i++)
        {
          RSM[i][ll].r = r[i];
          if(RSM[i][ll].r != RSM[i][ll].r2) failure = 1;
        }

        if(failure) counter[10]++;

      }
    }  /*  End ll = 0 ... I  */
  }  /*  End jj = 0 .. max_iter  */


  ne = 0;
  for(jj=0; jj<I; jj++) if(decoded_word[jj]!=0) ++ne;

  ++frame_failure[ne];

  /*  Send the result to the output */

  for(jj=0; jj<I; jj++)
  {
    if(decoded_word[jj] == 0)
    {
      ++counter[2];  /*  Number of incorrect RSW  */
      ne = 0;
      for(j=0; j<n; j++)
        if(RSM[j][jj].r != RSM[j][jj].r2) ne++;
      ++error_stat2[ne-16];
    }

    else ++counter[1];  /*  Number of correct RSW  */

    for(j=0 ; j<k; j++)
    {
      if (RSM[n-k+j][jj].r == -1)  sum = 0;
      else  sum = GF_table[RSM[n-k+j][jj].r];
      for (l=0; l<m; l++)
      {
        c1 = pow(2, m-l-1);
        if (sum >= c1)
        {
          data [ii*I*k*m + +jj*m*k + j*m + l]  =  1;
          sum -= c1;
        }
        else  data [ii*I*k*m + +jj*m*k + j*m + l]  =  0;
      }
    }
  }
}
```

```
  for(i=0;i<n*m;i++)
    free(*RSM);
        free(RSM);

  free(beta);
  free(r);
  free(r2);
  free(U);
  free(S);
  free(T);
  free(B);
  free(tmp);
  free(lambda);
  free(omega);
  free(counter);

}


void rs_decode_erasure_method_3(int *data, int *v2, int *counter, int n, int k, int q,
        int m, int num,  int *erasure_counter, int I, int *decoded_word,
        int *frame_failure, int *error_stat1, int *error_stat2)

{
  int GF_poly[11] = {0,0,0,6,12,20,48,72,184,272,576};
  int i, ii, j, l, ll, t, rr, sum, sum2, L, c1, deg_lambda, delta_r;
  int error, decode_flag, num_errors, numer, den, I_flag;
  int num_erasures, temp, max_eras, max_eras1, cc, failure, ne;
  int *r, *r2, *U, *S, *lambda, *omega, *B, *T, *tmp, *beta;
  int jj, kk, reli, GEBE, offset, GE_count, r_count, max_iter, id, id1;
  int redecode_flag, new_info_flag;
  int *GF_table, *dec_table;

  struct mat **RSM;

  GF_table = ivector(n+1);
  dec_table = ivector(n+1);

  t = (n-k)/2;

  beta = ivector(n);
  r = ivector(n);
  r2 = ivector(n);
  U = ivector(2*t);
  S = ivector(n);
  T = ivector(n);
  B = ivector(n);
  tmp = ivector(n);
  lambda = ivector(n);
  omega = ivector(2*t+1);


  max_eras1 = 16;
  max_iter = I;
```

```
/*  Create GF(2^m) field  */

make_GF_table (m, GF_table, dec_table, GF_poly[m]);


for (ii = 0; ii < num/I; ii ++)
{

  /*  Enter the data into the matrix  */

  counter[13] ++;  /* Total number of frames */

  I_flag = 0;

  for(i=0; i<I; i++)
  {
    id = 0;
    decoded_word[i] = 0;
    ne = 0;
    counter[0]++;  /* number of RSW */

    for(j = 0; j < n; j++)
    {
      RSM[j][i].r = 0;
      RSM[j][i].r2 = 0;
      RSM[j][i].reli = 500;
      RSM[j][i].flag = 0;
      sum = 0;
      sum2 = 0;
      for (l = 0; l < m; l++)
      {
        if( RSM[j][i].reli > abs(data[ii*I*n*m + i*n*m + j*m +l]))
          RSM[j][i].reli = abs(data[ii*I*n*m + i*n*m + j*m +l]);

        if(data[ii*I*n*m + i*n*m + j*m +l] <= 0)
         data[ii*I*n*m + i*n*m + j*m +l] = 0;
        else
         data[ii*I*n*m + i*n*m + j*m +l] = 1;

        sum += data[ii*I*n*m + i*n*m + j*m +l]*pow(2, m-1-l);
        sum2 += v2[ii*I*n*m + i*n*m + j*m +l]*pow(2, m-1-l);
      }
      RSM[j][i].r = dec_table[sum];
      RSM[j][i].r2 = dec_table[sum2];

      ++counter[11];  /*  Total number of bytes  */

      if(sum != sum2)
      {
        ++ne;
        counter[12] ++;  /* Number of byte errors  */
      }

    }
    if(ne >trunc_length) ne = 32;
  if(ne <=16) ne = 16;
  else  I_flag = 1;


  ++ error_stat1[ne-16];
}

if(I_flag) counter[14] ++;  /* Number of frame errors */


/*  If GEN_STAT = 1, then the probability of flagging a BE given a GE */
/*  and the probability of flagging a GE given a BE  */

if(GEN_STAT)
{
   for(i=0; i<I; i++)
   {
     for(j=0; j<n; j++)
     {
       reli = RSM[j][i].reli;
       if(reli >= 255) continue;

       if(RSM[j][i].r == RSM[j][i].r2) GEBE = 2;
       else GEBE = 1;

       kk = j;
       offset = 0;

       /*  loop forward looking for identical reliabilities  */

       for(l=0; l<I; l++)
       {
         if(i+l>=I && kk == n-1) break;
         if(i+l>=I)
         {
           kk = j+1;
           offset = I;
         }
         if(RSM[kk][i+l-offset].reli == reli)
         {
           if(GEBE == 1)
           {
             counter[6]++;
             if(RSM[kk][i+l-offset].r == RSM[kk][i+l-offset].r2)
               counter[5]++;
             else counter[4]++;
           }
           if(GEBE == 2)
           {
             counter[9]++;
             if(RSM[kk][i+l-offset].r == RSM[kk][i+l-offset].r2)
               counter[7]++;
             else counter[8]++;
           }
```

```
        }
        else break;
      }

      /*  loop backwards looking for identical identities  */

      kk = j;
      offset = 0;

      for(l=0; l<I; l++)
      {
        if(i-l < 0 && kk == 0)  break;
        if(i-l < 0)
        {
          kk = j-1;
          offset = I;
        }
        if(RSM[kk][i-l+offset].reli == reli)
        {
          if(GEBE == 1)
          {
            counter[6]++;
            if(RSM[kk][i-l+offset].r == RSM[kk][i-l+offset].r2)
              counter[5]++;
            else counter[4]++;
          }
          if(GEBE == 2)
          {
            counter[9]++;
            if(RSM[kk][i-l+offset].r == RSM[kk][i-l+offset].r2)
              counter[7]++;
            else counter[8]++;
          }
        }
        else break;
      }
    }  /*  end loop j = 0 to n */
  }  /*  end loop i = 0 to I */
}  /*  end if(GEN_STAT)  */


if(DEBUG_DUMP)
{
  printf("\nDecoded Word\n\n");
  for(i=0; i<I; i++) printf("%d\t",decoded_word[i]);

  printf("\n");
  for(i=0; i<n; i++)
  {
    printf("\n%d  ",i);
    for(j=0; j<I; j++)
    {
      if(RSM[i][j].r != RSM[i][j].r2)
        printf("(%d, GE)\t",RSM[i][j].reli);
      else
        printf("(%d, BE)\t",RSM[i][j].reli);
    }
  }
  if(PAUSE) getchar();
}


if(DEBUG1)
{

  printf("\nReceived word\n");

  for(i=0; i<I; i++)
  {
    printf("\n");
    for(j=0; j<n; j++)
    {
      printf(" %d",RSM[j][i].r);
    }
  }

  if(PAUSE) getchar();


  printf("\nCorrect word\n");

  for(i=0; i<I; i++)
  {
    printf("\n");
    for(j=0; j<n; j++)
    {
      printf(" %d",RSM[j][i].r2);
    }
  }

  if(PAUSE) getchar();


  printf("\nReliability\n");

  for(i=0; i<I; i++)
  {
    printf("\n");
    for(j=0; j<n; j++)
    {
      printf(" %d",RSM[j][i].reli);
    }
  }

  if(PAUSE) getchar();


  printf("\nError flag\n");
  for(i=0; i<I; i++)
  {
    printf("\n");
    for(j=0; j<n; j++)
    {
      printf(" %d",RSM[j][i].flag);
    }
  }

 if(PAUSE) getchar();
} /*  end if (DEBUG1) */

id1 = 0;
id = 2;

/*  Attempt the iteritive decoding  */

for(jj=0; jj<max_iter; jj++)
{

  /*  Obtain the reliability info from the enibhoring codewords from */
  /*  the interleaver and/or SOVA  */

  if(jj==0) max_eras = 16;
  else max_eras = max_eras1;

  if(id1==id) break;

  id = id1;

  if(DEBUG)
  {
    printf("\ndecoded word =  ");
    for(i=0; i<I; i++) printf("%d ", decoded_word[i]);
  }

  if(id == I) break;

  if(jj==0) redecode_flag = 1;
  else redecode_flag = 0;

  for(i=I-1; i>=1; i--)
    if(decoded_word[i] !=0 && decoded_word[i-1] == 0)
      redecode_flag = 1;

  if(!redecode_flag) break;

  /*  Start prioritising the reliability info, and then begin decoding */

  for(ll = 0; ll<I; ll++)  /*  Find good erasures (GE) first */
  {
    GE_count = 0;

    if(DEBUG)
    {
```

```
  printf("\ndecoded word =  ");
  for(i=0; i<I; i++) printf("%d ", decoded_word[i]);
}

if(decoded_word[ll] != 0) continue;

for(i=0; i<n; i++)
{
  if(RSM[i][ll].flag == 1)
  {
    U[GE_count] = i;
    GE_count ++;
  }
}

if(GE_count < max_eras)
{
  r_count = 0;
  for(i=0; i<n; i++)
  {
    if(GE_count + r_count < max_eras)
    {
      if(RSM[i][ll].flag == 0)
      {
        U[GE_count + r_count] = i;
        r_count++;
      }
    }

    if(GE_count + r_count == max_eras)
    {
      for(j=GE_count; j<max_eras; j++)
      {
        for(l=GE_count; l<max_eras-1; l++)
        {
          if(RSM[U[l]][ll].reli > RSM[U[l+1]][ll].reli)
          {
            temp = U[l];
            U[l] = U[l+1];
            U[l+1] = temp;
          }
        }
      }
      r_count ++;
    }

    if(GE_count + r_count > max_eras)
    {
      if(RSM[U[max_eras-1]][ll].reli > RSM[i][ll].reli
        && RSM[i][ll].flag == 0)
      {

        U[max_eras-1] = i;
```

```
         for(j=max_eras-1; j>=GE_count+1; j--)
         {
           if(RSM[U[j]][ll].reli < RSM[U[j-1]][ll].reli)
           {
             temp = U[j];
             U[j] = U[j-1];
             U[j-1] = temp;
           }
           else break;
         }
       }
     } /* End if(GE_count + r_count > max_eras) */

   } /* End loop i = 0 .. n */
 } /* End loop if(GE_count < max_eras) */

 /* Now we have reliabilities try to decode */



 if(DEBUG)
 {
   printf("Number of trials = %d\n\nMinimum reliabilities and positions\n", counter[15]);
   for(i=0; i<max_eras; i++) printf("(%d, %d)",U[i], RSM[U[i]][ll].reli);
   printf("\n");
   printf("\nnumber of GE = %d", GE_count);
   cc = 0;
   for(i=0; i<n; i++)
   {
     if(RSM[i][ll].r != RSM[i][ll].r2)
     {
       cc += 1;
       printf("\n%d error at %d  Reliability = %d", cc, i, RSM[i][ll].reli);
       for(j=0; j<max_eras; j++)
       {
         if(i == U[j]) printf(" min %d", j);
       }

       if (RSM[i][ll].flag == 1) printf(" GE");
     }
   }
  if(PAUSE) getchar();
 } /* end if (DEBUG) */

for(i=0; i<n; i++) r[i] = RSM[i][ll].r;

decode_flag = 1;

if(GE_count >= max_eras1) max_eras = GE_count;
else max_eras = max_eras1;

for(num_erasures=GE_count; num_erasures <= max_eras; num_erasures +=2)
{
```

```
if(decode_flag)
{
          counter[15] ++;  /* Number of decoding trials  */

  for(i=0; i<num_erasures; i++)  r[U[i]] = -1;  /*  =0  */

  error = 0;

  /*  Compute the syndrome   */
  for(i=1; i<= n-k; i++)
  {
     S[i] = 0;
     for(j=0; j<n; j++)
       if(r[j] != -1)
         S[i] ^= GF_table[ (r[j] + i*j) % n];
     if (S[i] != 0)  error = 1; /*  If nonzero syndrome, there is an error*/
  }
  S[0] = 0;

  /*  Convert to GF representation  */
  for(i=1; i<=n-k; i++)   S[i] = dec_table[S[i]];

  if (error)   /*  If the syndrome is equal to zero, no decoding nessasary  */
  {
    for(i=0; i < n; i++)
    {
      lambda[i] = 0;
      B[i] = -1;
      T[i] = 0;
    }

    lambda[0] = GF_table[0];    /*   =1  */
    L = 0;
    deg_lambda = 0;
    B[0] = 0;      /*  = 1   */

    for(rr=1; rr <= 2*t; rr++)
    {
      if (rr <= num_erasures)
      {
        for (i=1; i <= deg_lambda+1; i++)
        {
          if (lambda[i-1] != 0)
            tmp[i] = GF_table[(U[rr-1]+dec_table[lambda[i-1]])%n];
          else  tmp[i] = 0;
        }
        for (j=1; j <= deg_lambda+1; j++)   lambda[j] ^= tmp[j];
        deg_lambda++;
        for(j=0; j<= deg_lambda; j++)   B[j] = dec_table[lambda[j]];
        L ++;
      }

      else
      {
```

```
    /*  Compute the discrepancy  */
    delta_r = 0;
    for(j=0; j<=rr; j++)
      if (lambda[j] != 0  &&  S[rr-j] != -1)
        delta_r ^= GF_table[(dec_table[lambda[j]]+S[rr-j])%n];
    delta_r = dec_table[delta_r];

    if (delta_r != -1)
    {
      /*   T(x) <---  lambda(x)  +  delta_r * x * B(x)  */

      for (j=1; j <= deg_lambda +1; j++)
        if (B[j-1] != -1)
          T[j] = lambda[j] ^GF_table[ (delta_r + B[j-1]) % n];
      T[0] = lambda[0];
      ++deg_lambda;

      if (2*L <= rr + num_erasures-1)
      {
        L = rr - L + num_erasures;

        /*  B(x) <---  B(x) /  delta_r  */
        for(j=0; j<= deg_lambda; j++)
          if(  lambda[j] != 0)
            B[j] = (n-delta_r+dec_table[lambda[j]])%n;
          else  B[j] = -1;

        /*  lambda(x) <----  T(x)   */
        for(j=0; j<=deg_lambda+1; j++)  lambda[j] = T[j];

      }
      else
      {
        /*  lambda(x) <----  T(x)   */
        for(j=0; j<n-k; j++)   lambda[j] = T[j];

        /*  B(x)  <-----  x * B(x)   */
        tmp[0] = -1;
        for(j=1; j<=n-k; j++)   tmp[j] = B[j-1];
        for(j=0; j<=n-k; j++)   B[j] = tmp[j];
      }
    }

    else
    {
      /*  B(x)  <-----  x * B(x)   */
      tmp[0] = -1;
      for(j=1; j<=n-k; j++)   tmp[j] = B[j-1];
      for(j=0; j<=n-k; j++)   B[j] = tmp[j];
    }
  }
}

/*  Change lambda(x) to GF representation  */
for(i=0; i<n; i++)  lambda[i] = dec_table[lambda[i]];
```

```
/*  Compute the degree of lambda(x)   */
deg_lambda = n;
for(i=n-1; i>=0; i--)
  if (lambda[i] != -1 && deg_lambda == n) deg_lambda = i;

if (deg_lambda <= 2*t)  /*  Below the capacity of the code  */
{

  /*    Comupte omega(X) = [1+S(X)] * lambda(X)    */

  for(i=0; i<=n-k; i++)  omega[i] = 0;
  for (i=0; i<=n-k; i++)
  {
    for (j=0; j<=n-k; j++)
    {
      if ((i+j) >= n-k+1)  continue;
      if (S[i] != -1 && lambda[j] != -1)
      omega[i+j] ^= GF_table[ (S[i] + lambda[j]) % n ];
    }
  }

  /*   Convert omega(x) to GF representation  */
  for (i=0; i<=n-k; i++) omega[i] = dec_table[omega[i]];

  /*   Find the roots of lambda(X).  The inverses of the roots gives us the   */
  /*   location of the errors.                                                 */

  num_errors = 0;
  for (i=0; i<q-1; i++)
  {
    sum = 0;
    for (j=0; j<2*t; j++)
      if (lambda[j] != -1)
        sum ^= GF_table[ (lambda[j] + i*j) % (q-1) ];
    if (sum == 0)
    {
       beta[num_errors] = (n-i)%n;
      num_errors++;
    }
  }

  if(  (2*(num_errors-num_erasures) + num_erasures) <= 2*t)
  {
    if (num_errors == deg_lambda)
    {

      /*  Correct Decoding.  record the umber of erasures required */

      erasure_counter[num_erasures] += 1;

      decode_flag = 0;

      /* Convert r to base 10 representation */
      for(i=0; i<n; i++)
        if (r[i] != -1)  r[i] = GF_table[r[i]];
        else r[i] = 0;

      /* Calculate the error values and correct the received vector  */

      for (i=0; i<num_errors; i++)
      {
        /*  Calculate the denominator   */
        den = 0;

        for (j=0; j<=2*t; j++)
        {
          if (j%2 == 0)  continue;
          if (lambda[j] != -1)
          {
            c1 = GF_table[ ( (q-1-beta[i])*(j-1) + lambda[j]) % n  ];
            den ^= c1;
          }
        }

        den = dec_table[den];

        /*  Calculate the numerator   */
        numer = 0;
        for (j=0; j<=2*t; j++)
          if (omega[j] != -1)  numer ^= GF_table[ ((q-1-beta[i])*j + omega[j])%n ];

        numer = dec_table[numer];

        /*   Correct the erroneous value   */
        if(numer != -1)
          r[beta[i]] ^= GF_table[(n+numer+beta[i]-den)%n];

      }

      /*  Change r back into the GF  representation   */

      for(i=0; i<n; i++)   r[i] = dec_table[r[i]];

    }     /* end if  (num_errors == deg_lambda)  */
    else
    {
      decode_flag = 1;

    }
  }     /* end if (num_erasures + num errors <= dmin)  */
  else
  {
    decode_flag = 1;

  }
}     /* end if (deg_lambda <= 2*t)   */
else
{
```

```
          decode_flag = 1;
        }
      }     /* end if (error)  */
      else
      {
        erasure_counter[num_erasures] += 1;
        decode_flag = 0;
       }

     }  /* end if decode flag  */



   }  /* end for num_erasures = 0 to max_eras  */



   if (!decode_flag)
   {
     /*  Correct decoding.  */


      decoded_word[ll] = 1;
      id1 +=1;

      for(i=0; i<n; i++)
      {
        if(RSM[i][ll].r != RSM[i][ll].r2) RSM[i][ll].flag = 1; /* GE */
        else RSM[i][ll].flag = 2;
      }

      if(DEBUG)
      {
        printf("\nWord %d decoded correctly", ll);
       if(PAUSE) getchar();
      }

      failure = 0;
      for(i=0; i<n; i++)
      {
        RSM[i][ll].r = r[i];
        if(RSM[i][ll].r != RSM[i][ll].r2) failure = 1;
      }

      if(failure) counter[10]++;

/*  Begin updating the flags */


  for(j=0; j<n; j++)
  {
    reli = RSM[j][ll].reli;
    if(reli >= 255) continue;
    GEBE = RSM[j][ll].flag;
```

```
     kk = j;
     offset = 0;

     /*  loop forward looking for identical reliabilities  */

     for(l=0; l<I; l++)
     {
       if(ll+l>=I && kk == n-1) break;
       if(ll+l>=I)
       {
         kk = j+1;
         offset = I;
       }
       if(RSM[kk][ll+l-offset].reli == reli)
       {
         if(decoded_word[ll+l-offset] == 0)
           RSM[kk][ll+l-offset].flag = GEBE;
       }
       else break;
     }

     /*  loop backwards looking for identical identities  */

     kk = j;
     offset = 0;

     for(l=0; l<I; l++)
     {
       if(ll-l < 0 && kk == 0)  break;
       if(ll-l < 0)
       {
         kk = j-1;
         offset = I;
       }
       if(RSM[kk][ll-l+offset].reli == reli)
       {
         if(decoded_word[ll-l+offset] == 0)
           RSM[kk][ll-l+offset].flag = GEBE;

       }
       else break;
     }
   }  /*  end loop j = 0 to n */

   decoded_word[i] = 2;  /*  flag this word as already giving GEBE info  */


 /*  Done revising reliability info  */



    }
  }  /*  End ll = 0 ... I  */
}  /*  End jj = 0 .. max_iter  */
```

```
    ne = 0;
    for(jj=0; jj<I; jj++) if(decoded_word[jj]!=0) ++ne;

    ++frame_failure[ne];

    /*  Send the result to the output */

    for(jj=0; jj<I; jj++)
    {
      if(decoded_word[jj] == 0)
      {
        ++counter[2];  /*  Number of incorrect RSW  */
        ne = 0;
        for(j=0; j<n; j++)
          if(RSM[j][jj].r != RSM[j][jj].r2) ne++;
        ++error_stat2[ne-16];
      }

      else ++counter[1];  /*  Number of correct RSW  */

      for(j=0 ; j<k; j++)
      {
        if (RSM[n-k+j][jj].r == -1)  sum = 0;
        else  sum = GF_table[RSM[n-k+j][jj].r];
        for (l=0; l<m; l++)
        {
          c1 = pow(2, m-l-1);
          if (sum >= c1)
          {
            data [ii*I*k*m + +jj*m*k + j*m + l]  =  1;
            sum -= c1;
          }
          else  data [ii*I*k*m + +jj*m*k + j*m + l]  =  0;
        }
      }
    }

   }

  for(i=0;i<n*m;i++)
    free(*RSM);
      free(RSM);

  free(beta);
  free(r);
  free(r2);
  free(U);
  free(S);
  free(T);
  free(B);
  free(tmp);
  free(lambda);
  free(omega);
}
```

B.8 Main program for real system using erasure Method 1

```
# include <stdio.h>
# include <math.h>
# include <stdlib.h>
# include <stddef.h>
# include "mmt31.h"

# define L_DEBUG 0
# define PAUSE 0

# define Nss 16      /* Number of samples per symbol */
# define Rb 1   /* The bit rate */

# define Tb (1.0/Rb)
# define Eb 1.0      /* The Energy per bit */

# define Pi 3.14159265359

/* Defining the random number generator constants */

# define IM1 2147483563
# define IM2 2147483399
# define AM (1.0/IM1)
# define IMM1 (IM1-1)
# define IA1 40014
# define IA2 40692
# define IQ1 53668
# define IQ2 52774
# define IR1 12211
# define IR2 3791
# define NTAB 32
# define NDIV (1+IMM1/NTAB)
# define EPS 1.2e-7
# define RNMX (1.0-EPS)


void main(int argc, char *argv[])
{
  int *v, *u, i, ii, errors, M1, M2, Q;
  long *idum;
  long idum_cell;
  long num_samples, num_bits, N, N_total;
  int rs_m, rs_t, rs_n, rs_k, rs_q, rs_num;
  int conv_k, conv_n, conv_m, conv_num;
  double *data, *Tx, *Rx, Pe;
  double Psig, EbNo, gain;
  int max_repeat, I, rows, I_num, a;

  double *x1, *x2;
```

```
 FILE   *inpf0, *inpf1, *inpf2,  *inpf3, *outf0;

 int  **soft_metric, *erasure_counter, *rs_counter;
 int  num_erasures;
 char cFilename[80];

 idum=&idum_cell;

 EbNo=atof(argv[1]);
 N = atol(argv[2]);
 I = atol(argv[3]);
 max_repeat = atol(argv[4]);


 sprintf(cFilename,"New_Real_M1_erasure_%2.2fEbNo_%dI_32delta_8soft.output",EbNo,I);

 inpf0 = fopen("m1.dat","r");
 inpf1 = fopen("m2.dat","r");
 inpf2 = fopen("tx.dat","r");
 inpf3 = fopen("rx.dat","r");


 /* obtain the number of single sided filter */
 /* coefficents for Tx (M1) and Rx (M2)  */

 fscanf(inpf0, "%d", &M1);
 fscanf(inpf1, "%d", &M2);

 Tx = dvector(2*M1);
 Rx = dvector(2*M2);

 x1 = dvector(2*M1+1);
 x2 = dvector(2*M2+1);

 /*  Read in the filter coefficients from data files  */

 for(i=0; i<2*M1; i++)
   fscanf(inpf2, "%lf", &Tx[i]);

 for(i=0; i<2*M2; i++)
   fscanf(inpf3, "%lf", &Rx[i]);


/*  The Reed Solomon coding parameters  */

 rs_m = 8;
 rs_t = 16;
 rs_q = pow(2, rs_m);
 rs_n = rs_q-1;
 rs_k = rs_n - 2*rs_t;

/*  Interleaving parameters  */


 rows = rs_m*rs_n;
```

```
/*  The Convolutional coding parameters  */

conv_k = 1;
conv_n = 2;
conv_m = 6;

Q = 8;  /*  Number of soft decision levels  */

rs_num = N/(rs_k*rs_m);

if( N % (rs_k*rs_m) != 0)
{
  ++rs_num;
  N = rs_num*rs_k*rs_m;
}

I_num = rs_num/I;


if(rs_num%I != 0)
{
  ++ I_num;
  rs_num = I_num*I;
  N = rs_num*rs_k*rs_m;
}


gain = (double)(conv_k*rs_k)/(conv_n*rs_n);

 a = 10.0;

conv_num = (rs_num*rs_n*rs_m + 50)/conv_k;
num_bits = conv_n*(conv_num + 100);
num_samples = num_bits*Nss;
errors = 0;
erasure_counter = ivector(2*rs_t);
rs_counter = ivector(rs_t);

soft_metric = int_matrix_2d(2, Q);

for(i=0; i<6; i++)  rs_counter[i] = 0;

N_total = 0;

data = dvector(num_samples+3*M1);


v = ivector(num_bits);
u = ivector(N);


*idum = -10;
```

```
for(ii=0; ii<max_repeat; ii++)
{
  for (i=0; i<num_bits; i++) v[i] = 0;



    for (i=0; i<N; i++)
    {
      v[i] = bitgen(idum);
      u[i] = v[i];
    }

    rs_encode(v, rs_n, rs_k, rs_m, rs_num);

    interleave(v, rows, rs_n, rs_m, I, I_num);

    conv_encode(v, conv_n, conv_k, conv_m, conv_num);

    modulate(data, v, num_bits);


    filter(data, M1, Tx, x1, num_samples);


    /*  Adjust for delay introduced by filter */

    for(i=0; i<num_samples; i++)
      data[i] = data[i+M1];

    Psig = calc_power(data, num_samples);


    add_noise(idum, data, num_samples, Psig, EbNo, gain);

    filter(data, M2, Rx, x2, num_samples);

    /*  Adjust for delay introduced by filter */

    for(i=0; i<num_samples; i++)
      data[i] = data[i+M2];

    demodsoft(data, v, num_bits, soft_metric, Q, a);

    conv_decode(v, conv_n, conv_k, conv_m,  conv_num, soft_metric, a);

    deinterleave(v, rows, rs_n, rs_m, I, I_num);

    rs_decode(v, rs_n, rs_k, rs_q, rs_m, rs_num, erasure_counter, rs_counter);

    for (i=0; i<N; i++)
      if (v[i] != u[i])
      {
         errors += 1;
      }
```

```
        N_total += N;

        outf0=fopen(cFilename,"at+");
        for (i=0; i<=rs_t; i+=2)fprintf(outf0,"%de= %d ",i, erasure_counter[i]);

        fprintf(outf0,"\nnum = %d\n\nRS counter =  ",rs_num);

        for(i=0; i<6; i++)    fprintf(outf0,"  %d",rs_counter[i]);

        fprintf(outf0,"\n\n1.  S = 0.  No Decoding\n2.  deg_lambda > 2*t");
        fprintf(outf0,"\n3.  2*e + f  >  2*t\n4.  num_errors != deg_lambda");
        fprintf(outf0,"\n5.  Correct decoding\n5.  Incorrect Decoding");


        fprintf(outf0,"\n\nEb/No = %2.2f  dB",EbNo);
        fprintf(outf0,"\n(%d, %d, %d)  Reed Solomon Code",rs_n, rs_k, rs_t);
        fprintf(outf0,"\ninterleaving depth I = %d",I);
        fprintf(outf0,"\n(%d, %d, %d) convolutional code", conv_n, conv_k, conv_m);
        fprintf(outf0,"\niteration number = %d\n# errors = %d\n N total = %ld", ii+1, errors, N_total);
        Pe = (float)errors/N_total;
        fprintf(outf0,"\nPe = %2.10f\n\n",Pe);
        fclose(outf0);

    }

    outf0=fopen(cFilename,"at+");
    fprintf(outf0,"\nerasure counter = ");
    for (i=0; i<=rs_t; i+=2) fprintf(outf0,"%de= %d ",i, erasure_counter[i]);

    fprintf(outf0,"\nnum = %d\n\nRS counter =  ",rs_num);

    for(i=0; i<6; i++)    fprintf(outf0,"  %d",rs_counter[i]);

    fprintf(outf0,"\n\n1.  S = 0.  No Decoding\n2.  deg_lambda > 2*t");
    fprintf(outf0,"\n3.  2*e + f  >  2*t\n4.  num_errors != deg_lambda");
    fprintf(outf0,"\n5.  Correct decoding\n5.  Incorrect Decoding");


    fprintf(outf0,"\n\nEb/No = %2.2f  dB",EbNo);
    fprintf(outf0,"\nNumber of erasures = %d", num_erasures);
    fprintf(outf0,"\n(%d, %d, %d)  Reed Solomon Code",rs_n, rs_k, rs_t);
    fprintf(outf0,"\ninterleaving depth I = %d",I);
    fprintf(outf0,"\n(%d, %d, %d) convolutional code", conv_n, conv_k, conv_m);
    fprintf(outf0,"\niteration number = %d\n# errors = %d\n N total = %ld", ii+1, errors, N_total);
    Pe = (float)errors/N_total;
    fprintf(outf0,"\nPe = %2.10f\n\n",Pe);

    fclose(outf0);
    fclose(inpf1);
    fclose(inpf2);
    fclose(inpf3);

    free(x1);
    free(x2);
    free(data);
```

```
  free(v);
  free(u);
  free(Rx);
  free(Tx);
  free_2d_int_matrix(2, soft_metric);
}
```

B.10 Main program for ideal system using erasure Method 2

```
# include <stdio.h>
# include <math.h>
# include <stdlib.h>
# include <stddef.h>
# include "mmt33.h"

# define FILE_PRINT 1  /* Prints either to file(1) or screen(0) */
                       /* Note: All DEBUG flags must be 0 */

# define GEN_STAT 1

# define PAUSE 0  /* uses getchar(); to pause the output displayed to the screen */
# define L_DEBUG 0  /* prints out reliability values in SOVA */
# define DEBUG 0
# define DEBUG1 0
# define DEBUG_DUMP 0  /* Shows the whole deinterleaving frame with reliability values */

# define Nss 16        /*  Number of samples per symbol */
# define Rb 1    /*  The bit rate */

# define Tb (1.0/Rb)
# define Eb 1.0        /*  The Energy per bit */

struct mat
        {
         int r;
         int r2;
         int reli;
         int flag;
        };

/*  Defining the random number generator constants */

# define IM1 2147483563
# define IM2 2147483399
# define AM (1.0/IM1)
# define IMM1 (IM1-1)
# define IA1 40014
# define IA2 40692
# define IQ1 53668
# define IQ2 52774
# define IR1 12211
# define IR2 3791
# define NTAB 32
# define NDIV (1+IMM1/NTAB)
# define EPS 1.2e-7
# define RNMX (1.0-EPS)

int main(int argc, char *argv[])
{
  int *v, *v2, *u, i, ii, j, errors,  Q;
  long  *idum;
  long idum_cell;
  long   num_bits, N, N_total;
  int rs_m, rs_t, rs_n, rs_k, rs_q, rs_num;
  int conv_k, conv_n, conv_m, conv_num;
  double  Pe, x, var, sd, A, amp, b, a;
  double  EbNo, gain, *prob;
  int  max_repeat, I, rows, I_num, level, frame_error;

  FILE *outf0;

  int  **soft_metric;
  int  *erasure_counter, *counter, *dccoded_word;
  int *frame_failure, *error_stat1, *error_stat2, **num_bits_per_level;

  char cFilename[80];

  idum=&idum_cell;

  EbNo=atof(argv[1]);
  N = atol(argv[2]);
  I = atol(argv[3]);
  max_repeat = atol(argv[4]);


  if(FILE_PRINT) sprintf(cFilename,"B_Method2_%2.3fEbNo%dI.output",EbNo,I);


  /* The Reed Solomon coding parameters  */

  rs_m = 8;
  rs_t = 16;
  rs_q = pow(2, rs_m);
  rs_n = rs_q-1;
  rs_k = rs_n - 2*rs_t;


  /* Interleaving parameters  */

  rows = rs_m*rs_n;

  /* The Convolutional coding parameters   */

  conv_k = 1;
  conv_n = 2;
  conv_m = 6;

  Q = 256;  /* Number of soft decision levels  */
  rs_num = N/(rs_k*rs_m);

  if( N % (rs_k*rs_m) != 0)
  {
    ++rs_num;
    N = rs_num*rs_k*rs_m;
  }

  I_num = rs_num/I;

  if(rs_num%I != 0)
  {
    ++ I_num;
    rs_num = I_num*I;
    N = rs_num*rs_k*rs_m;
  }

  gain = (double)(conv_k*rs_k)/(conv_n*rs_n);

  a = 10.0;

  conv_num = (rs_num*rs_n*rs_m +50)/conv_k;
  num_bits = conv_n*(conv_num + 100);
  errors = 0;

  soft_metric = int_matrix_2d(2, Q);
  num_bits_per_level = int_matrix_2d(2, Q);
  counter = ivector(7);


  prob = dvector(Q);

/*  stuff from RS encode */

  frame_failure = ivector(I+1);
  error_stat1 = ivector(32);  /* Keeps track of how many RSW contain x number of errors */
  error_stat2 = ivector(32);

  decoded_word = ivector(I);

  for(i=0; i<7; i++)  counter[i] = 0;

  amp = 1.0;

  N_total = 0;

  v = ivector(num_bits);
  u = ivector(N);
  v2 = ivector(num_bits);

  *idum = -10;   /*  Random number initial seed  */

  A = gain*pow(10.0, (EbNo/10.0));
```

```
var =  1.0/(2*A);

sd = sqrt(var);

for(ii=0; ii<max_repeat; ii++)
{
  for (i=0; i<num_bits; i++) v[i] = 0;

    for (i=0; i<N; i++)
    {
      v[i] = bitgen(idum);
      u[i] = v[i];
    }

    rs_encode(v, rs_n, rs_k, rs_m, rs_num);

    for(i=0; i<num_bits; i++)  v2[i] = v[i];

    interleave(v, rows, rs_n, rs_m, I, I_num);

    conv_encode(v, conv_n, conv_k, conv_m, conv_num);

    for (i=0; i<num_bits; i++)
    {
      if(v[i] == 1)  x = 1.0 + sd*gasdev2(idum);
      else   x = -1.0 + sd*gasdev2(idum);

      /*  Demod  */

      for(j=0; j<Q; j++)
        if(x>=amp*(2*j-Q)/Q && x< amp*(2*(j+1)-Q)/Q)
          level = j;

      if(x >= amp) level = Q-1;
      if(x <= -amp)  level = 0;

      num_bits_per_level[v[i]][level] += 1;

      v[i] = level;
    }

     for(i=0; i<Q; i++)
      prob[i] = (double)(num_bits_per_level[0][i] + num_bits_per_level[1][Q-i-1] + 1)/(2*num_bits +Q);

    b = -log(prob[Q-1])/log(2.0);

    for(i=0; i<Q; i++)
    {
      soft_metric[0][i] = (int)(floor)(a*(log(prob[i])/log(2.0) + b));
      soft_metric[1][Q-i-1] = soft_metric[0][i];
    }

    conv_decode(v, conv_n, conv_k, conv_m,  conv_num, soft_metric, a);
```

```
      deinterleave(v, rows, rs_n, rs_m, I, I_num);

      rs_decode_erasure_method_2(v, v2, counter, rs_n, rs_k, rs_q, rs_m, rs_num,
             erasure_counter, I, decoded_word, frame_failure, error_stat1, error_stat2);

      for (i=0; i<N; i++)
        if (v[i] != u[i])
           errors += 1;

      N_total += N;

      if(!FILE_PRINT)
      {

        printf("\n\n");
        printf("Erasure Counter\n");
        for (i=0; i<=2*rs_t-4; i+=2) printf("%de= %d ",i, erasure_counter[i]);

        printf("\n\nError Counter 1 (number of times x amount of errors occoured)\n");

        for(i=0; i<16; i++)
        {
          if(i==7) printf("\n");
          printf("%derr = %d ", 16+i, error_stat1[i]);
        }

        printf("\n\nError Counter 2 (# of errors for non-decoded word)\n");

        for(i=0; i<16; i++)
        {
          if(i==7) printf("\n");
          printf("%derr = %d ", 16+i, error_stat2[i]);
        }

        printf("\n\nFrame decoded with x correct\n");
        for(i=0; i<=I; i++) printf("%dcct = %d ",i, frame_failure[i]);


        if(GEN_STAT)
        {
          printf("\n\nNumber of GE flagged GE = %d  Prob = %1.3f\n", counter[4],
(float)counter[4]/counter[6]);
          printf("Number of GE flagged BE = %d  Prob = %1.3f\n",
counter[5],(float)counter[5]/counter[6]);
          printf("Total number of GE declaired = %d\n",counter[6]);

          printf("\nNumber of BE flagged BE = %d  Prob = %1.3f\n", counter[7],
(float)counter[7]/counter[9]);
          printf("Number of BE flagged GE = %d  Prob = %1.3f\n", counter[8],
(float)counter[8]/counter[9]);
          printf("Total number of BE declaired = %d\n",counter[9]);
        }


        printf("\n\nBefore Decoding, no erasures used:\n");
```

```
  printf("\nTotal number of bytes = %d", counter[11]);
  printf("\nNumber of byte errors = %d", counter[12]);
  printf("\nProb of byte error = %2.5lf",(double)counter[12]/counter[11]);

  printf("\n\nTotal number of frames = %d", counter[13]);
  printf("\nNumber of frames in error = %d", counter[14]);
  printf("\nProb of frame error = %2.5lf", (double)counter[14]/counter[13]);

  printf("\n\nNumber of RSW = %d",counter[0]);
  printf("\nNumber of RSW correct = %d",error_stat1[0]);
  printf("\nNumber of RSW incorrect = %d", counter[0]-error_stat1[0]);
  printf("\nProb of RSW incorrect = %2.5lf",(double)(counter[0]-error_stat1[0])/counter[0]);

  printf("\n\nAfter decoding:\n");

  frame_error = 0;
  for(i=0; i<I; i++) frame_error += frame_failure[i];

  printf("\nTotal number of frames = %d", counter[13]);
  printf("\nNumber of frames in error = %d", frame_error);
  printf("\nProb of frame error = %2.5lf",(double)frame_error/counter[13]);


  printf("\n\nNumber of RSW = %d",counter[0]);
  printf("\nNumber of RSW correct = %d",counter[1]);
  printf("\nNumber of RSW incorrect = %d", counter[2]);
  printf("\nProb of RSW incorrect = %2.5lf",(double)counter[2]/counter[0]);

  printf("\n\nTotal number of decoding trials = %d",counter[15]);
  printf("\nAverage number of decoding trials = %3.3lf", (double)counter[15]/counter[0]);

  printf("\nNumber of decoding failures = %d\n",counter[10]);



  printf("\n\nEb/No = %2.2f  dB",EbNo);
  printf("\n(%d, %d, %d)  Reed Solomon Code",rs_n, rs_k, rs_t);
  printf("\ninterleaving depth I = %d",I);
  printf("\n(%d, %d, %d) convolutional code", conv_n, conv_k, conv_m);
  printf("\niteration number = %d\n# errors = %d\n N total = %ld", ii+1, errors, N_total);
  Pe = (float)errors/N_total;
  printf("\nPe = %2.10f\n\n",Pe);
}

if(FILE_PRINT)
{

  outf0=fopen(cFilename,"at+");

  fprintf(outf0,"\n\nErasure Counter\n");
  for (i=0; i<=2*rs_t-2; i+=2) fprintf(outf0,"%de= %d ",i, erasure_counter[i]);

  fprintf(outf0,"\n\nError Counter 1 (number of times x amount of errors occoured)\n");
```

```
    for(i=0; i<16; i++) fprintf(outf0,"%derr = %d ", 16+i, error_stat1[i]);


    for(i=0; i<16; i++)
    {
      if(i==7) fprintf(outf0,"\n");
      fprintf(outf0,"%derr = %d ", 16+i, error_stat1[i]);
    }


    fprintf(outf0,"\n\nError Counter 2 (# of errors for non-decoded word)\n");

    for(i=0; i<16; i++)
    {
      if(i==7) fprintf(outf0,"\n");
      fprintf(outf0,"%derr = %d ", 16+i, error_stat2[i]);
    }


    fprintf(outf0,"\n\nFrame decoded with x correct\n");
    for(i=0; i<=I; i++) fprintf(outf0,"%dcct = %d ",i, frame_failure[i]);

    if(GEN_STAT)
    {
      fprintf(outf0,"\n\nNumber of GE flagged GE = %d  Prob = %1.3f\n", counter[4],
(float)counter[4]/counter[6]);
      fprintf(outf0,"Number of GE flagged BE = %d  Prob = %1.3f\n",
counter[5],(float)counter[5]/counter[6]);
      fprintf(outf0,"Total number of GE declaired = %d\n",counter[6]);

      fprintf(outf0,"\nNumber of BE flagged BE = %d  Prob = %1.3f\n", counter[7],
(float)counter[7]/counter[9]);
      fprintf(outf0,"Number of BE flagged GE = %d  Prob = %1.3f\n", counter[8],
(float)counter[8]/counter[9]);
      fprintf(outf0,"Total number of GE declaired = %d\n",counter[9]);
    }

    fprintf(outf0,"\n\nBefore Decoding, no erasures used:\n");

    fprintf(outf0,"\nTotal number of bytes = %d", counter[11]);
    fprintf(outf0,"\nNumber of byte errors = %d", counter[12]);
    fprintf(outf0,"\nProb of byte error = %2.5lf",(double)counter[12]/counter[11]);

    fprintf(outf0,"\n\nTotal number of frames = %d", counter[13]);
    fprintf(outf0,"\nNumber of frames in error = %d", counter[14]);
    fprintf(outf0,"\nProb of frame error = %2.5lf", (double)counter[14]/counter[13]);

    fprintf(outf0,"\n\nNumber of RSW = %d",counter[0]);
    fprintf(outf0,"\nNumber of RSW correct = %d",error_stat1[0]);
    fprintf(outf0,"\nNumber of RSW incorrect = %d", counter[0]-error_stat1[0]);
    fprintf(outf0,"\nProb of RSW incorrect = %2.5lf",(double)(counter[0]-error_stat1[0])/counter[0]);

    fprintf(outf0,"\n\nAfter decoding:\n");

    frame_error = 0;
```

```
      for(i=0; i<I; i++) frame_error += frame_failure[i];

      fprintf(outf0,"\nTotal number of frames = %d", counter[13]);
      fprintf(outf0,"\nNumber of frames in error = %d", frame_error);
      fprintf(outf0,"\nProb of frame error = %2.5lf",(double)frame_error/counter[13]);


      fprintf(outf0,"\n\nNumber of RSW = %d",counter[0]);
      fprintf(outf0,"\nNumber of RSW correct = %d",counter[1]);
      fprintf(outf0,"\nNumber of RSW incorrect = %d", counter[2]);
      fprintf(outf0,"\nProb of RSW incorrect = %2.5lf",(double)counter[2]/counter[0]);

      fprintf(outf0,"\n\nTotal number of decoding trials = %d",counter[15]);
      fprintf(outf0,"\nAverage number of decoding trials = %3.3lf", (double)counter[15]/counter[0]);

      fprintf(outf0,"\nNumber of decoding failures = %d\n",counter[10]);


      fprintf(outf0,"\n\nEb/No = %2.2f  dB",EbNo);
      fprintf(outf0,"\n(%d, %d, %d)  Reed Solomon Code",rs_n, rs_k, rs_t);
      fprintf(outf0,"\ninterleaving depth I = %d",I);
      fprintf(outf0,"\n(%d, %d, %d) convolutional code", conv_n, conv_k, conv_m);
      fprintf(outf0,"\niteration number = %d\n# errors = %d\n N total = %ld", ii+1, errors, N_total);
      Pe = (float)errors/N_total;
      fprintf(outf0,"\nPe = %2.10f\n\n",Pe);

      fclose(outf0);
    }

  }

  free(v);
  free(u);
  free(v2);
  free_2d_int_matrix(2, soft_metric);
  free_2d_int_matrix(2, num_bits_per_level);
  free(decoded_word);
  free(frame_failure);
  free(error_stat1);
  free(error_stat2);
}
```

References

[1] Berrou, Claude., Adde, Patrick., Ettiboua., and Faudeil, Stephane., "A Low Complexity Soft-Output Viterbi Decoder Architecture,". Procedings ICC '93, pp. 737 -740. May 1993.

[2] Blahut, R. E., *Theory and Practice of Error Control Codes*, Addison Wesley, Reading Mass., 1984.

[3] Forney, G. D., "On Decoding BCH Codes," *IEEE Transactions on Information Theory*, Volume IT-11, pp. 393-403, October 1965.

[4] Hagenauer, J. and Hoeher, P., "A Viterbi Algorithm with Soft Outputs and Its Applications," *Procedings of the IEEE Globecomm Conference, Dallas, Tex.*, pp. 47.1.1-47.1.7, November 1989.

[5] Jeruchim, Michael C., Balaban, Philip., and Shanmugan, K. Sam., *Simulation of Communication Systems.*, Plentum Press, New York, NY., 1992.

[6] Lin, S., and Costello, D. J., Jr., *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, Inc., Englewood Cliffs, NJ., 1983.

[7] Paaske, Eric, "Improved Decoding for a Concatenated Coding System Recomended by CCSDS," *IEEE Transactions on Communications*, Volume COM-38, pp. 1138-1144, August 1990.

[8] Press, W. H., Teukolsky, S. A., Vettering, W. T., Flannery, B. P., *Numerical Recipies in C.,* Cambridge University Press., 1992.

[9] Sklar, Bernard, *Digital Communications: Fundamentals and Applications,* Prentice Hall, Inc., Englewood Cliffs, NJ., 1988.

[10] Viterbi, Andrew J., Omura, Jim K., *Principles of Digital Communication and Coding*, McGraw-Hill, New York, NY., 1979.

[11] Wicker, Stephen B., *Error Control Systems for Digital Communication and Storage,* Prentice-Hall, Inc., Englewood Cliffs, NJ., 1995.

[12] Pratt, Timothy., Bostian, Charles., *Satellite Communications.* John Willey and Sons., New York, NY., 1976.

[13] Palmer, Larry C., "Computer Modeling and Simulation of Communications Satellite Channels,". *IEEE Journal on Selected Areas of Communications,* Volume SAC-2, No. 1, pp. 89-102, January 1984.

[14] Hagenaurer, J., Offer, E., and Papke, L., "Matching Viterbi Decoders and Reed Solomon Decoders in a Concatenated System,". *Reed Solomon Codes and their Applications,* Ch. 11, edited by Wicker, S., and Bhargava, V., IEEE Press, New York, NY., 1994.

[15] Wicker, S., and Bhargava, V., "An Introduction to Reed Solomon Codes,". *Reed Solomon Codes and their Applications,* Ch. 1, edited by Wicker, S., and Bhargava, V., IEEE Press, New York, NY., 1994.

[16] McEliece, Robert., Swanson, Liaf., "Reed Solomon Codes and the Exploration of the Solar System,". *Reed Solomon Codes and their Applications,* Ch. 3, edited by Wicker, S., and Bhargava, V., IEEE Press, New York, NY., 1994.

[17] Massey, J. L., "Shift Register Synthesis and BCH Decoding," IEEE Transactions on Information Theory, Vol. IT-15, No. 1, pp. 122-127, Jan. 1969.

[18] Feher, Kamilo., *Digital Communications Satellite/Earth Station Engineering*., Prentice Hall, Englewood Cliffs, N. J. 1985.

[19] McEliece, Robert, and Swanson, Liaf., "On the decoder error probability of Reed-Solomon codes," IEEE Transactions on Information Theory, Vol. IT-32, pp 701-703, Sept. 1986.