DSP KIT IMPLEMENTATION OF ENCODER DECODER CIRCUIT OF REED SOLOMON CODES

Enrollment No.: 101065

Name of Student: Aditya Vikram Singh

Supervisor's Name: Mr. Dheeraj Kumar Sharma

May-2014

Submitted in partial fulfillment of the Degree of Bachelor of Technology

DEPARTMENT OF ELECTRONIS AND COMMUNICATION ENGINEERING
JAYPEE UNIVERSITY OF INFORMATION & TECHNOLOGY WAKNAGHAT, SOLAN (H.P)

Table of Contents

CERTIFICATE

This is to certify that the work titled "**DSP KIT IMPLEMENTATION OF REED SOLOMON CODES"** submitted by "**ADITYA VIKRAM SINGH"** in partial fulfillment for the award of degree of B. Tech Electronics and communication Engineering of Jaypee University of Information Technology, Waknaghat has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

_____

(Signature of Supervisor)

Name of Supervisor: Mr. Dheeraj Kumar Sharma

Designation: Assistant Professor

Date:

ACKNOWLEDGEMENT

I take this opportunity to express my profound gratitude and deep regards to Mr. Dheeraj Kumar, my Project Guide, for guiding and correcting me at every step of my work with attention and care. She has taken pain to go through the project and make necessary correction as and when needed. Thanks and appreciation to the helpful people at college for their support. I would also thank my university and my faculty members without whom this project would have been a distant reality. I also extend my heartfelt thanks to my family and friends for their undaunted support and faith in me.

Signature of the Student…….

Name of the Student – Aditya Vikram Singh

Date -

SUMMARY

Objective

DSP kit implementation of encoder and decoder circuit of Reed Solomon code.

Description

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

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

The Reed-Solomon encoder takes a block of digital data and adds extra "redundant" bits. Errors occur during transmission or storage for a number of reasons (for example noise or interference, scratches on a CD, etc). The Reed-Solomon decoder processes each block and attempts to correct errors and recover the original data. The number and type of errors that can be corrected depends on the characteristics of the Reed-Solomon code.

Why I chose this topic?

Communications is my favourite topic and I always had curiosity as to how communications systems work in our world. While studying communications I came across this topic and decided to take it up as a project

---------- ----------

Signature of Student Signature of Supervisor

Name: Name:

Date: Date:

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW OF THE PROJECT

DSP kit implementation of encoder and decoder circuit of Reed Solomon code. DSP kit used (TMS32C677XX) .Encoding and Decoding Algorithm used Berlekamp Massey Algorithm.

1.2 PROBLEM STATEMENT

RS codes are seen as a special case of the larger class of BCH codes but it was not until almost a decade later, by regarding them as cyclic BCH codes, that an efficient decoding algorithm gave them the potential to their widespread application.

1.3 PROJECT DESCRIPTION

1.3.1 Purpose:

RS codes, which are BCH codes, are used in applications such as spacecraft communications, compact disc players, disk drives, and two-dimensional bar codes. According to Bossert (1999) the relationship between BCH and RS codes is such that RS codes comprise a subset of BCH codes and occasionally BCH codes comprise a subset of RS codes. Van Lint (1999) defines an RS code as a primitive BCH code of length $n=q-1$ over $GF(q)$ .So we will use this concept to encode n decode RS codes using Berlekamp Massey Algorithm

1.3.2 Scope:

This project gives a simple way of encoding and decoding RS codes using Berlekamp Massey Algorithm on DSP KIT.

CHAPTER 2

LITERATURE REVIEW

2.1 CODING THEORY

A sender transmits a message through a channel to a receiver. The channel could be air when using a wireless network or the channel could be a data cable. Noise may appear on these types of channels, so in order to receive the message with as few errors as possible, ideally the sender should use high power signal amplification and the channel should be as short as possible. However, in normal situations these are not viable solutions. GSM telephones have, in fact, very small batteries and are rather energy efficient and an Ethernet cable in a building can be up to 100 meters before an active repeater or switch has to amplify the signal. In order to use as little energy as possible and transmit over a long distance, codewords have to be encoded. The message is then transmitted over a channel where errors may be introduced. The received codeword needs to be decoded into the received message.

2.1.1 LINEAR ALGEBRA

Mathematicians developed their coding theory using linear algebra, which works with sets of numbers or fields. These numbers can be added, subtracted, multiplied or divided. Fields like integers, the set of natural numbers $\mathbb{N}=\{0,1,2,\ldots\}$, are infinite fields; we could always imagine its largest element and add 1 to it. Information and code can be seen as elements in a finite field which comes with some advantages when using the binary number system.

2.1.2 GALOIS FIELDS

A finite field Fq is a field F which has a finite number of elements and q is the order of the field. This finite field is often called a Galois field, after the French mathematician Evariste Galois (1811 – 1832) and is denoted $GF(q)$. For the purpose of this thesis we consider only binary field $GF(2)$ and its extension fields $GF(2m)$ where $m\in\{2,3,4,\ldots\}$.
The following is always valid for all numbers in a binary Galois field (Blahut,1983):
• fields contain 0 or 1.
• adding two numbers gives one number in the set.
• subtracting two numbers gives one number in the set.
• multiplying one number gives one number in the set.
• dividing one number by 1, as division by 0 is not allowed, gives one number in the set.
• The distributive law, $(a+b)c=ac+bc$, holds for all elements in the field

2.2 EXTENSION FIELDS

Finite fields exist for all prime numbers q and for all pm where p is prime and m is a positive integer. $GF(q)$ is a sub-field of $GF(pm)$ and as such the elements of $GF(q)$ are a sub-set of the elements of $GF(pm)$, therefore $GF(pm)$ is an extension field of $GF(q)$.

Consider $GF(4)=\{0,1,2,3\}$, which is not a Galois field because it is of order 4, which is not a prime. The element 2 has no multiplicative inverse and therefore we cannot divide by 2. Instead, we could define $GF(4)=\{0,1,a,b\}$ with addition and multiplication . Now all elements do have additive and multiplicative inverses

These extension fields are used to handle non-binary codes where code symbols are expressed as m -bit binary code symbols, For example, $GF(4)$ consists of four different two-bit symbols and $GF(16)$ of 16 hexadecimal symbols. To obtain multiplication for binary, numbers are expressed as polynomials, they are multiplied and divided by the prime polynomial while the remainder is taken as result.

2.3 POLYNOMIALS

Let us we write $GF(4)$ as $GF(2_2)$ and take prime polynomial $p(x)=x_2+x+1$ which is an irreducible polynomial of degree 2, which can be checked by multiplying $p(x)$ with polynomials of a lesser degree, like 1 , x and $x+1$

In order to describe an extension field $GF(p_m)$ it is useful to know its primitive polynomial $p(x)$, where the degree of $p(x)$ is equal to m . For example, $GF(16)= GF(2_4)=\{0000, 0001,0010,\ldots,1111\}$ is a finite field that contains 16 4-bit code symbols. Addition is analogue to the example above. Multiplication can be obtained firstly by writing the symbols as polynomials to express which positions in these 4-bit codes are non-zero and, secondly, by using modulo 2 addition of coefficients in addition and multiplication. Let α be defined as a root of polynomial $p(x)$, such that we can write:$p(\alpha)=0$
Thus for $GF(16)$ with its irreducible polynomial $p(x)=x_4+x+1$ we can write:
$\alpha_4+\alpha+1=0$
$\alpha_4=0-\alpha-1$
We have already noted that subtraction is the same as addition in a binary finite field, so:
$\alpha_4=\alpha+1$
Therefore the polynomial of exponential α_4 is $\alpha+1$. From there we can calculate the polynomial for α_5 by:
$\alpha_5=\alpha\cdot\ \alpha_4$
$=\alpha\cdot\ (\alpha+1)$
$=\alpha_2+\alpha$

2.4 VECTOR SPACE

Linear codes can be represented as sets of vectors. Let us define a vector space $GF(qm)$. This is a vector space of a finite dimension m . The codewords are q -ary sets of m -elements or m -tuples which form the coordinates of the endpoints of the vectors. Figure 2.1 presents two of such m -dimensional vector spaces. In such a vector space, every codeword can be presented as a the sum of two vectors give another vector in the same vector space (Bose, 2008). For example, $GF(22)$ is a two-dimensional vector space. It has four binary vectors.
Take vectors $v1=[0,1]$, $v2=[1,0]$ and $v3=[1,1]$, then $v1+v2=[0,1]+[1,0]=[1,1]$, which is a vector in the same space. Vectors $v1, v2,\dots , v\,k$ are linear independent if there is not a single set of scalars $ai \neq 0$, such that $a_1 v_1+a_2 v_2+\dots+a_k v_k=0$. For example, vectors [0,1] and [1,0] are linearly independent, but [0,1] and [1,1] are linear dependent vectors

CHAPTER 3 : LINEAR BLOCK CODES

3.1 HAMMING WEIGHT , MINIMUM DISTANCE and CODE RATE

The Hamming weight $w_H(x)$ of a codeword or vector x is defined as the amount of non-zero elements or vector coordinates, which ranges from zero to length n of said codeword. $w_H(x)=\sum_{j=0\ n-1} w_H(x_j)$, *where* $w_H(x_j)=\begin{cases} 0, & x_j=0 \\ 1, & x_j\neq 0 \end{cases}$ The Hamming distance $d_H(x , y)$ between two codewords or vectors x and y is defined as amount of elements or coordinates where x and y differ.

$d_H(x , y)=\sum_{j=0\ n-1} w_H(x_j+y_j)$, *where* $w_H(x_j+y_j)=\begin{cases} 0, & x_j=y_j \\ 1, & x_j\neq y_j \end{cases}$
$d_H(x , y)=w_H(x , y)$
The minimum distance d_{min} of code C is the minimum distance between two different codewords. The minimum distance for linear codes is equal to the minimum weight (Bossert, 1999). However, a codeword containing only zeros and, therefore, having a distance of zero is disregarded as the minimum distance cannot be zero. Let x , y be codewords in code C . A received vector, which is the sent vector x in C , plus error vector e can only be corrected if the distance between any other codeword y in C fulfil $d_{min}(x , x+e)<d_{min}(y , x+e)$ or $w_{min}(e)<w_{min}(x+y+e)$. Therefore $w_{min}(e)\leq d-1$, where d is the distance.This is written as $t\leq d-1\ 2$ or $d\geq 2t+1$, where t is the amount of errors that can be corrected. In general, a code C of length n , with M codewords, and a minimum distance $d=d(C)$, is called an (n , M , d) code. Then $M\leq qn-d+1$ and the code rate of a q -ary (n , M , d) code is at most $1-d-1\ n$. A linear q -ary code of length n , with k codewords or message symbols, and distance d , is called a (n , k , d) code or (n, k) code. The code rate is defined as

$R=\log q\ k/n$
If, according to Shannon's channel coding theorem, rate R is less than capacity C , then the code exists but if rate R is larger than capacity C , the error probability is 1 and the length of the codeword becomes infinite.

3.1 SINGLETON BOUND

It is preferable to have a large minimum distance d so that many errors can be corrected. Also, a large amount of codewords M would allow for efficient use of bandwidth when transmitting over a noisy channel. Unfortunately, increasing d tends to increase n or decrease M . The Singleton bound is an upper bound for M in terms of n and d . A code that satisfies the Singleton bound is called a MDS code (maximum distance separable). The Singleton bound can be written as $q_d \leq q_{n+1}/M$
for the MDS code to obtain the largest possible value of d for a given n and m

3.2 MAXIMUM-LIKELIHOOD DECODING

There are two principles of decoding. In hard-decision decoding the received bits are believed to be either 1 or 0 in binary transmission. The decoding is done bit by bit In soft-decisiondecoding, the received codewords may contain samples of bits with many values, not just 1 or 0. Calculating the closest error-free codeword is more complicated but soft-decision decoding has better performance than the hard-decision decoding. Assuming hard-decision decoding is used, the received codeword is decoded into its closest codeword measured by its smallest Hamming distance. This minimum probability of error principle is called Maximum-Likelihood or Minimum Distance Decoding (Geisel, 1990).

3.3 CYCLIC CODES

Cyclic codes are widely used in data communication because their structure makes encoder and decoder circuitry simple. Hill (1986) defines code C as cyclic (n, k) code if C is a linear code of length n over a finite field and if any cyclic shift of a codeword is also a codeword. Thus,(c_0 , c_1, c_2 ,... , c_{n-1}) $\in C$ and (c_{n-1} , c_0 , c_1 ,... , c_{n-2}) $\in C$. Let $g(x)$ be the polynomial with the smallest degree. By dividing its highest coefficient, we may assume that the highest non-zero coefficient of $g(x)$ is 1. The polynomial $g(x)$ is called the generator polynomial for C , which must be a divisor of x_n-1 (in a binary field this is equal to x_n+1) with a degree of $n-k$. Subsequently, every cyclic code is a polynomial The encoder for cyclic codes is then $c(x)=i(x)\cdot g(x)$ where $c(x)$ is the polynomial with degree $n-1$ of codeword (c_0 , c_1, c_2 ,... , c_{n-1}) which is calculated as $c(x) = \sum_{i=0\ n-1} c_i \cdot x_i = c_0+c_1 x+c_2 x_2+...+c_{n-1} x_{n-1}$ and $i(x)$ is the information polynomial of degree $k-1$. Generator polynomial $g(x)$ must be of degree $n-k$.

CHAPTER 4 : BCH CODES

A BCH code is a cyclic polynomial code over a finite field with a particularly chosen generator polynomial. Hamming codes are the subset of BCH codes with $k=2_m-1-m$ and have an error correction of 1. Generally, a family of t - error correcting codes defined over finite fields $GF(q)$, where $2t+1<q$, are BCH codes or RS codes (Hill, 1986). The main advantage of BCH codes is the ease with which they can be decoded using syndrome and many good decoding algorithms exist. A well-known decoding algorithm is the Berlekamp-Massey algorithm. This allows very simple electronic hardware to perform the task, making the need for a computer unnecessary. This implies that a decoding device may be small and consume little power.

4.1 GENERATING BCH CODE

It is easy to generalise the construction of a t -error-correcting code of length $n=2_m-1$ over $GF(q)=\{0, 1, ..., q-1\}$ provided $2t+1\leq n\leq q-1$. According to Hill (1986) it is not difficult to construct a binary BCH code over an extension field $GF(q_m)$. In order to obtain a cyclic code only the generator polynomial $g(x)$ is needed. For any integer $m\geq 3$ and $t<2_{m-1}$, there exists a primitive BCH code with parameters:

$n = 2_m-1$

$n-k \leq m\cdot t$

$d_{min} \leq 2t+1$

Let α be a primitive n -th root of unity of $GF(2_m)$. For $1\leq i\leq t$, let $m_{2i-1}(x)$ be the minimum polynomial of α_{2i-1} . The degree of $m_{2i-1}(x)$ is m or a factor of m . The generator polynomial $g(x)$ of a t -error-correcting primitive BCH codes of length 2_m-1 is given by $g(x)=LeastCommon\ Multiple\ \{m_1(x),m_2(x),m_3(x),...,m_{2t-1}(x),m_{2t}(x)\}$ and because every even power of a primitive element has the same minimal polynomial as some odd power of the element, then $g(x)$ can be reduced to $g(x)=LCM\{m_1(x),m_3(x),...,m_{2t-1}(x)\}$ The degree of $g(x)$ is $m\cdot t$ or less and so is the number of parity check bits, therefore $n-k\leq m\cdot t$

4.2 DECODING a BCH CODE

BCH codes can be decoded in many way and it is most common that

- Syndromes values for are calculated for the received codeword;
- Error polynomials are calculated;
- Roots of these polynomials are calculated to obtain the location of errors;
- Error values are calculated at these locations.

Chapter 5:Reed-Solomon codes

RS codes, which are BCH codes, are used in applications such as spacecraft communications, compact disc players, disk drives, and two-dimensional bar codes. According to Bossert (1999) the relationship between BCH and RS codes is such that RS codes comprise a subset of BCH codes and occasionally BCH codes comprise a subset of RS codes. Van Lint (1999) defines an RS code as a primitive BCH code of length $n=q-1$ over $GF(q)$.

5.1 GENERATING A REED-SOLOMON CODE

Let $GF(q)$ be a finite field with q elements and it generate a rather specific BCH code C over $GF(q)$ of length n , called a Reed-Solomon code. Let α be a primitive n -th root of unity of $GF(q)$ and let code C have a length of $n=q-1$. Now take d so that $1\leq d\leq n$ and the generator polynomial $g(x)$ is given by

$g(x)=\prod_{i=1}^{d-1}(x-\alpha_i) = (x-\alpha)(x-\alpha_2)\ldots(x-a_{d-1})$

.

Trappe and Washington (2006) state that the minimum distance for C is at least d . Since $g(x)$ is a polynomial of degree $d-1$, it has at most d nonzero coefficients. Therefore, the codeword corresponding to the coefficients of $g(x)$ has a weight of at most d . It follows that C has a weight of exactly d and the dimension of C is n minus the degree of $g(x)$ $n-deg(g) = n-(d-1) = n+1-d$.

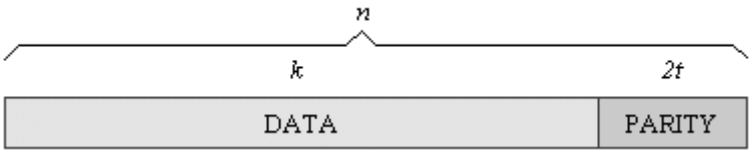
Therefore, a Reed-Solomon code is a cyclic $(n, n+1-d, d)$ code with codewords corresponding to polynomials, where each $f(x)$ is a polynomial with coefficients in $GF(q)$ that cannot be factored into lower degree polynomials while assuming that the highest non-zero coefficient is 1: $g(x)f(x)$ with $deg(f)\leq n-d$. It follows that there are q choices for each $n-d+1$ coefficients of $f(x)$, and thus there are q_{n-d+1} codewords in code C . Therefore, an RS code is a MDS code since it makes the Singleton bound an equality

5.2 PROPERTIES of REED-SOLOMON CODES

Reed Solomon codes are a subset of BCH codes and are linear block codes. A Reed-Solomon code is specified as RS (n,k) with s-bit symbols.

This means that the encoder takes k data symbols of s bits each and adds parity symbols to make an n symbol codeword. There are $n-k$ parity symbols of s bits each. A Reed-Solomon decoder can correct up to t symbols that contain errors in a codeword, where $2t = n-k$.

The following diagram shows a typical Reed-Solomon codeword (this is known as a Systematic code because the data is left unchanged and the parity symbols are appended).



Example:

A popular Reed-Solomon code is RS (255,223) with 8-bit symbols. Each codeword contains 255 code word bytes, of which 223 bytes are data and 32 bytes are parity. For this code:

n = 255, k = 223, s = 8

2t = 32, t = 16

The decoder can correct any 16 symbol errors in the code word: i.e. errors in up to 16 bytes anywhere in the codeword can be automatically corrected. Given a symbol size s, the maximum codeword length (n) for a Reed-Solomon code is $n = 2^s - 1$. For example, the maximum length of a code with 8-bit symbols (s=8) is 255 bytes. Reed-Solomon codes may be shortened by (conceptually) making a number of data symbols zero at the encoder, not transmitting them, and then re-inserting them at the decoder.

Example: The (255,223) code described above can be shortened to (200,168). The encoder takes a block of 168 data bytes, (conceptually) adds 55 zero bytes, creates a (255,223) codeword and transmits only the 168 data bytes and 32 parity bytes.

The amount of processing "power" required to encode and decode Reed-Solomon codes is related to the number of parity symbols per codeword. A large value of t means that a large number of errors can be corrected but requires more computational power than a small value of t.

Symbol Errors

One symbol error occurs when 1 bit in a symbol is wrong or when all the bits in a symbol are wrong.

Example: RS (255,223) can correct 16 symbol errors. In the worst case, 16 bit errors may occur, each in a separate symbol (byte) so that the decoder corrects 16 bit errors. In the best case, 16 complete byte errors occur so that the decoder corrects 16 x 8 bit errors. Reed-Solomon codes are particularly well suited to correcting burst errors (where a series of bits in the codeword are received in error).

5.3 DECODING

Reed-Solomon algebraic decoding procedures can correct errors and erasures. An erasure occurs when the position of an erred symbol is known. A decoder can correct up to *t* errors or up to *2t* erasures. Erasure information can often be supplied by the demodulator in a digital communication system, i.e. the demodulator "flags" received symbols that are likely to contain errors.

When a codeword is decoded, there are three possible outcomes:

1. If 2s + r < 2t (s errors, r erasures) then the original transmitted code word will always be recovered,

OTHERWISE

2. The decoder will detect that it cannot recover the original code word and indicate this fact.

OR

3. The decoder will mis-decode and recover an incorrect code word without any indication.

The probability of each of the three possibilities depends on the particular Reed-Solomon code and on the number and distribution of errors.

Coding Gain

The advantage of using Reed-Solomon codes is that the probability of an error remaining in the decoded data is (usually) much lower than the probability of an error if Reed-Solomon is not used. This is often described as coding gain.

Example: A digital communication system is designed to operate at a Bit Error Ratio (BER) of 10^{-9}, i.e. no more than 1 in 10^{9} bits are received in error. This can be achieved by boosting the power of the transmitter or by adding Reed-Solomon (or another type of Forward Error Correction). Reed-Solomon allows the system to achieve this target BER with a lower transmitter output power. The power "saving" given by Reed-Solomon (in decibels) is the coding gain.

5.3.1 REED SOLOMON DECODER:

The received codeword is entered to RS decoder to be decoded, the decoder first tries to check if this codeword is a valid codeword or not. If it does not, errors occurred during transmission. This part of the decoder processing is called error detection. If errors are detected, the decoder tries to correct this error using error correction part.

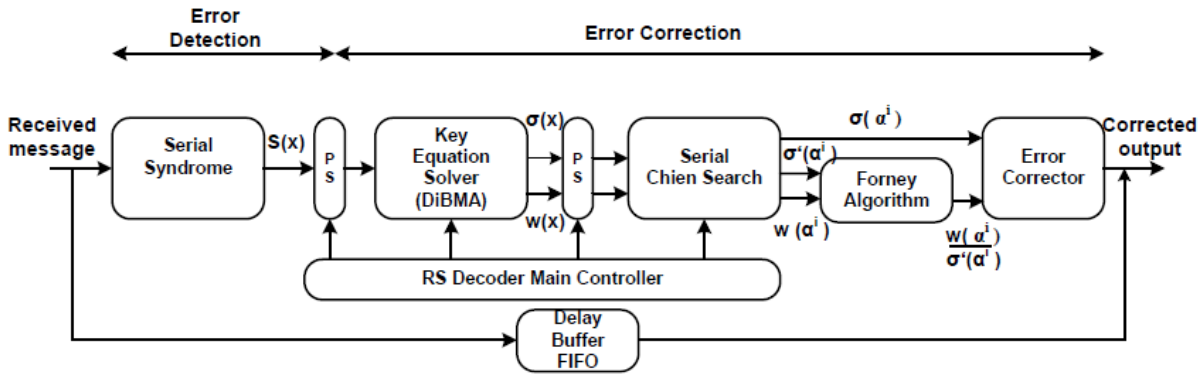


Figure shows the main block diagram of Reed Solomon decoder which consists of two main parts:

1. Error detection part, in this part we use "Syndrome computation" block.
2. Error correction part, this part consists of three blocks:

   - Decoding algorithm which used to find the coefficients of error-location polynomial σ(x) and error-evaluator polynomial $W(x)$ it sometimes called "Key equation solver".
   - Chien search block which used to find the roots of $\sigma(x)$ which present the inverse of the error locations.
   - Forney algorithm block which used to find the values of the errors.

After getting the values and locations of the error, we can correct the received codeword by xoring the received vector with the error vector.

Error Detection "Syndrome Calculation":

The first step in RS decoder is to check if there is any error in the received codeword or not. This done using Syndrome computation block.

- Let the transmitted codeword polynomial c(X) formed as follow:

  $$c(x) = c_0 + c_1 X + \ldots\ldots + c_{n-1} X^{n-1}, \quad where \quad c_i \in GF(2^m)$$

- Let the received codeword polynomial r(X) formed as follow:

  $$r(x) = r_0 + r_1 X + \ldots\ldots + r_{n-1} X^{n-1}, \quad where \quad r_i \in GF(2^m)$$

- Let the error polynomial e(X) which added by the channel formed as:

$$e(x) = e_0 + e_1 X + ... + e_{n-1} X^{n-1}, \quad where \quad e_i \in GF(2^m)$$

Which is related to the received polynomial r(X) and the transmitted polynomial c(X) as follows:

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

The transmitted polynomial c(x) must be multiple of the generator polynomial g(X), and the received polynomial r(X) is evaluated form the addition between c(X) and e(X). So the roots of g(X) should give zero in the received polynomial if the error polynomial is zero. i.e., no errors occurred.

- Let the syndrome polynomial S(x) formed as:

$$S(x) = \sum_{i=1}^{2t} S_i x^{i-1}$$

Where, $i = 1, 2, ..., 2t$.

Each coefficient can be described as follows:

$$S_i = r(\alpha^i), i = 1, 2, ..., 2t$$

If there is no error, all syndrome coefficients must give zero. If there is any non-zero coefficient, it means that there is an occurrence for error.

5.3.2 THE DECODING ALGORITHM:

After calculation of the syndrome coefficients we can detect if there exist errors in the received codeword or not by checking these values, if all these coefficients are zeros there will be no errors if not there will be error in an unknown location in the codeword with an unknown value.

The main function of the decoding algorithm is to get the error location polynomial σ(x), and the error evaluator polynomial $W(x)$, which represents the locations and the values of the errors respectively.
The first error correction procedure for Reed Solomon codes was found by Gornstien and Zierler, and improved by Chien and Forney. This procedure is known as the key equation solver, as it will be discussed later.

Decoding algorithms can be categorized into two types:

- Serial algorithms in which the error locator polynomial $\sigma(x)$ is calculated first then we substituted in the key equation to calculate the error evaluator polynomial $W(x)$, e.g. (Berlekamp–Massey algorithm).
- Parallel algorithms in which the error locator polynomial $\sigma(x)$ and the error evaluator polynomial $W(x)$ are calculated are in parallel, e.g. (Euclidean algorithm).

5.3.3 DECODING of RS CODES USING BERLEKAMP-MASSEY ALGORITHM:

The Berlekamp–Massey (B-M) algorithm is a method used as decoding algorithm used for RS and BCH codes. This method used in RS codes to calculate the coefficients of the error locator polynomial for the error locations, and the coefficients of the error evaluator polynomial for the error values. In BCH the values of the errors are binary so we only calculate the coefficients of the error locator polynomial.

Let the error polynomial e(X) contains τ errors placed at positions $X^{j1}, X^{j2}, ..., X^{j\tau}$ with error values $e_{j1}, e_{j2}, ..., e_{j\tau}$ then:

$$e(X) = e_{j1}X^{j1} + e_{j2}X^{j2} + ... + e_{j\tau}X^{j\tau}$$

Now we have to calculate the values of e_{ji} and the powers of X^{ji}.

We have 2t syndrome coefficients. Each syndrome coefficient S_i can be expressed as:

$$s_i = r(\alpha^i) = c(\alpha^i) + e(\alpha^i) = e(\alpha^i)$$

From both above equations, we can obtain set of equation that relates the error locations and values to the syndrome coefficient in the form of:

$$s_1 = r(\alpha) = e(\alpha) = e_{j1}\alpha^{j1} + e_{j2}\alpha^{j2} + ... + e_{j\tau}\alpha^{j\tau}$$
$$s_2 = r(\alpha^2) = e(\alpha^2) = e_{j1}\alpha^{2j1} + e_{j2}\alpha^{2j2} + ... + e_{j\tau}\alpha^{2j\tau}$$
.
.
.
$$s_{2t} = r(\alpha^{2t}) = e(\alpha^{2t}) = e_{j1}\alpha^{2tj1} + e_{j2}\alpha^{2tj2} + ... + e_{j\tau}\alpha^{2tj\tau}$$

This set of equation can be simplified in the form:

$$s_1 = r(\alpha) = e(\alpha) = e_{j1}\beta_1 + e_{j2}\beta_2 + ... + e_{j\tau}\beta_\tau$$
$$s_2 = r(\alpha^2) = e(\alpha^2) = e_{j1}\beta_1^2 + e_{j2}\beta_2^2 + ... + e_{j\tau}\beta_\tau^2$$
.
.
.
$$s_{2t} = r(\alpha^{2t}) = e(\alpha^{2t}) = e_{j1}\beta_1^{2t} + e_{j2}\beta_2^{2t} + ... + e_{j\tau}\beta_\tau^{2t}$$

where, $\beta_i = \alpha^{ji}$ *and* $i = 1, 2, 3, ..., \tau$

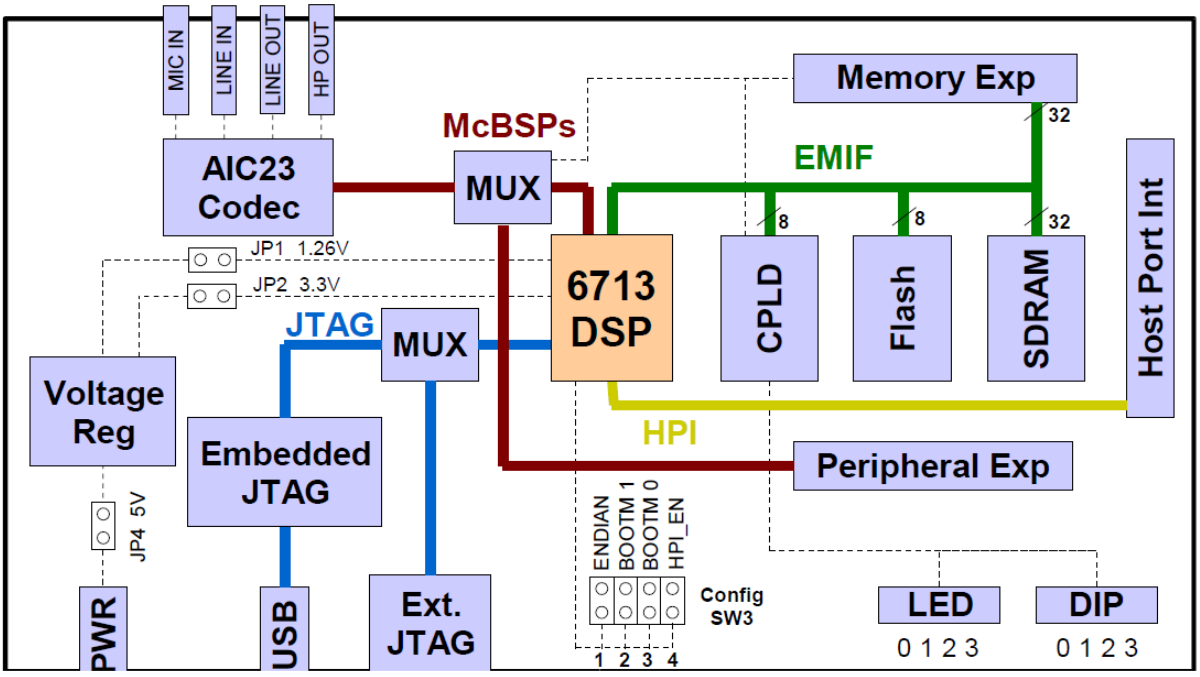
From above equation, we have 2*t* equations in 2*t* unknowns as worst case, but these equations are not linear equations so we define the two polynomials:

- The error locator polynomial $\sigma(x)$ which present the locations of the error.
- The error evaluator polynomial $W(x)$ which presents the values of the errors.

As mentioned before that Berlekamp-Massey algorithm is a serial algorithm so the error location polynomial $\sigma(x)$ is calculated first then the error evaluator polynomial $W(x)$.

CHAPTER 6: DSP KITARCHITECTURE:

The 6713 DSK is a low-cost standalone development platform that enables customers to evaluate and develop applications for the TI C67XX DSP family. The DSK also serves as a hardware reference design for the TMS320C6713 DSP. Schematics, logic equations and application notes are available to ease hardware development and reduce time to market. The DSK uses the 32-bit EMIF for the SDRAM (CE0) and daughter card expansion interface (CE2 and CE3). The Flash is attached to CE1 of the EMIF in 8-bit mode. An on-board AIC23 codec allows the DSP to transmit and receive analog signals. McBSP0 is used for the codec control interface and McBSP1 is used for data. Analog audio I/O is done through four 3.5mm audio jacks that correspond to microphone input, line input, line output and headphone output. The codec can select the microphone or the line input as the active input. The analog output is driven to both the line out (fixed gain) and headphone (adjustable gain) connectors. McBSP1 can be re-routed to the expansion connectors in software. A programmable logic device called a CPLD is used to implement glue logic that ties the board components together. The CPLD has a register based user interface that lets the user configure the board by reading and writing to the CPLD registers. The registers reside at the midpoint of CE1.



The DSK includes 4 LEDs and 4 DIP switches as a simple way to provide the user withinteractive feedback. Both are accessed by reading and writing to the CPLD registers. An included 5V external power supply is used to power the board. On-board voltage regulators provide the 1.26V DSP core voltage, 3.3V digital and 3.3V analog voltages. A voltage

supervisor monitors the internally generated voltage, and will hold the board in reset until the

supplies is within operating specifications and the reset button is released. If desired, JP1 and JP2 can be used as power test points for the core and I/O power supplies. Code Composer communicates with the DSK through an embedded JTAG emulator with a USB host interface. The DSK can also be used with an external emulator through the external JTAG connector.

6.1 DSP KIT KEY FEATURES:

The C6713 DSK is a low-cost standalone development platform that enables users to evaluate and develop applications for the TI C67xx DSP family. The DSK also serves as a hardware reference design for the TMS320C6713 DSP. Schematics, logic equations and application notes are available to ease hardware development and reduce time to market.

The DSK comes with a full complement of on-board devices that suit a wide variety of application environments. Key features include:

- A Texas Instruments TMS320C6713 DSP operating at 225 MHz.
- An AIC23 stereo codec
- 16 Mbytes of synchronous DRAM
- 512 Kbytes of non-volatile Flash memory (256 Kbytes usable in default configuration)
- 4 user accessible LEDs and DIP switches
- Software board configuration through registers implemented in CPLD
- Configurable boot options
- Standard expansion connectors for daughter card use
- JTAG emulation through on-board JTAG emulator with USB host interface or external emulator
- Single voltage power supply (+5V)

CHAPTER 7: IMPLEMENTATION PROCEDURES

1. To create a New Project
   Project – New *(rscode.pjt)*
2. To Create a Source file
   File- New
   Type the code (Save as & give a name to file, Eg: rscode.asm).
3. To Add Source files to Project
   Project- Add files to Project- rscode.asm
4. To Add rts6700.lib file & hello.cmd:
   Project -Add files to Project - rts6700.lib
   Path: c:\CCStudio\c6000\cgtools\lib\rts6700.lib
   Note: Select Object & Library in (*.o,*.l) in Type of file
   Project - Add files to Project - hello.cmd
   Path: c:\CCstudio\tutorial\dsk6713\hello1\hello.cmd
   Note: Select Linker Command file (*.cmd) in Type of files
5. To Compile:
   Project - Compile File
6. To build or Link:
   Project – Rebuild all,
   Which will create the final executable (.out) file (Eg. rscode.out).
7. Procedure to Load and Run program:
   Load program to DSK:
   File - Load program -rscode.out
8. To execute project:
   Debug - Run.

7.1 CODE IMPLEMENTATION:

```
#include <math.h>
#include <stdio.h>
#define mm 4          /* RS code over GF(2**4) - change to suit */
#define nn 15         /* nn=2**mm -1   length of codeword */
#define tt 3          /* number of errors that can be corrected */
#define kk 9          /* kk = nn-2*tt*/
int pp [mm+1] = { 1, 1, 0, 0, 1} ; /* specify irreducible polynomial coeffts */
int alpha_to [nn+1], index_of [nn+1], gg [nn-kk+1];
int recd [nn], data [kk], bb [nn-kk] ;
void generate_gf()
```

```
/* generate GF (2**mm) from the irreducible polynomial p(X) in pp[0]..pp[mm]; lookup tables:
index->polynomial form    alpha_to[] contains j=alpha**i; polynomial form -> index form
index_of[j=alpha**i] = i alpha=2 is the primitive element of GF(2**mm)*/
 {
register int i, mask ;
mask = 1 ;
alpha_to[mm] = 0 ;
for (i=0; i<mm; i++)
{
alpha_to[i] = mask ;
index_of[alpha_to[i]] = i ;
if (pp[i]!=0)
alpha_to[mm] ^= mask ;
mask<<= 1 ;
}
index_of[alpha_to[mm]] = mm ;
mask>>= 1 ;
for (i=mm+1; i<nn; i++)
{
if (alpha_to[i-1] >= mask)
alpha_to[i] = alpha_to[mm] ^ ((alpha_to[i-1]^mask)<<1) ;
else alpha_to[i] = alpha_to[i-1]<<1 ;
index_of[alpha_to[i]] = i ;
   }
index_of[0] = -1 ;
 }
void gen_poly()
/* Obtain the generator polynomial of the tt-error correcting, lengthnn=(2**mm -1) Reed
Solomon code  from the product of (X+alpha**i), i=1..2*tt*/
 {
register int i, j ;
gg[0] = 2 ;    /* primitive element alpha = 2  for GF(2**mm)  */
gg[1] = 1 ;    /* g(x) = (X+alpha) initially */
for (i=2; i<=nn-kk; i++)
{
```

```
gg[i] = 1 ;
for (j=i-1; j>0; j--)
      if (gg[j] != 0)  gg[j] = gg[j-1]^ alpha_to[(index_of[gg[j]]+i)%nn] ;
else gg[j] = gg[j-1] ;
gg[0] = alpha_to[(index_of[gg[0]]+i)%nn] ;     /* gg[0] can never be zero */
   }
  /* convert gg[] to index form for quicker encoding */
for (i=0; i<=nn-kk; i++)  gg[i] = index_of[gg[i]] ;
 }
void encode_rs()
/* take the string of symbols in data[i], i=0..(k-1) and encode systematically to produce 2*tt parity symbols in bb[0]..bb[2*tt-1], data[] is input and bb[] is output in polynomial form. Encoding is done by using a feedback shift register with appropriate connections specified by the elements of gg[], which was generated above. Codeword is   c(X) = data(X)*X**(nn-kk)+ b(X) */
 {
register int i, j ;
int feedback ;
for (i=0; i<nn-kk; i++)   bb[i] = 0 ;
for (i=kk-1; i>=0; i--)
{
feedback = index_of[data[i]^bb[nn-kk-1]] ;
if (feedback != -1)
{
for (j=nn-kk-1; j>0; j--)
if (gg[j] != -1)
bb[j] = bb[j-1]^alpha_to[(gg[j]+feedback)%nn] ;
else
bb[j] = bb[j-1] ;
bb[0] = alpha_to[(gg[0]+feedback)%nn] ;
      }
else
{
for (j=nn-kk-1; j>0; j--)
bb[j] = bb[j-1] ;
bb[0] = 0 ;
```

```
      } ;
   } ;
 } ;
void decode_rs()
{
register int i, j, u, q ;
int elp[nn-kk+2][nn-kk], d[nn-kk+2], l[nn-kk+2], u_lu[nn-kk+2], s[nn-kk+1] ;
int count=0, syn_error=0, root[tt], loc[tt], z[tt+1], err[nn], reg[tt+1] ;
/* first form the syndromes */
for (i=1; i<=nn-kk; i++)
{
s[i] = 0 ;
for (j=0; j<nn; j++)
if (recd[j]!=-1)
s[i] ^= alpha_to[(recd[j]+i*j)%nn] ;      /* recd[j] in index form */
/* convert syndrome from polynomial form to index form */
if (s[i]!=0)  syn_error=1 ;       /* set flag if non-zero syndrome => error */
s[i] = index_of[s[i]] ;
   } ;
if (syn_error)       /* if errors, try and correct */
   {
/* initialize table entries */
d[0] = 0 ;           /* index form */
d[1] = s[1] ;        /* index form */
elp[0][0] = 0 ;      /* index form */
elp[1][0] = 1 ;      /* polynomial form */
for (i=1; i<nn-kk; i++)
{
elp[0][i] = -1 ;   /* index form */
elp[1][i] = 0 ;   /* polynomial form */
       }
l[0] = 0 ;
l[1] = 0 ;
u_lu[0] = -1 ;
```

```
u_lu[1] = 0 ;
    u = 0 ;
do
    {
     u++;
if (d[u]==-1)
{
l[u+1] = l[u] ;
for (i=0; i<=l[u]; i++)
{
elp[u+1][i] = elp[u][i] ;
elp[u][i] = index_of[elp[u][i]] ;
          }
        }
else
/* search for words with greatest u_lu[q] for which d[q]!=0 */
{
q = u-1 ;
while ((d[q]==-1) && (q>0)) q-- ;
/* have found first non-zero d[q] */
if (q>0)
{
j=q;
do
{
j--;
if ((d[j]!=-1) && (u_lu[q]<u_lu[j]))
                q = j;
}
while (j>0) ;
          } ;
/* have now found q such that d[u]!=0 and u_lu[q] is maximum */
/* store degree of new elp polynomial */
if (l[u]>l[q]+u-q)  l[u+1] = l[u] ;
```

```
else  l[u+1] = l[q]+u-q ;
/* form new elp(x) */
for (i=0; i<nn-kk; i++)    elp[u+1][i] = 0 ;
for (i=0; i<=l[q]; i++)
if (elp[q][i]!=-1)
elp[u+1][i+u-q] = alpha_to[(d[u]+nn-d[q]+elp[q][i])%nn] ;
for (i=0; i<=l[u]; i++)
{
elp[u+1][i] ^= elp[u][i] ;
elp[u][i] = index_of[elp[u][i]] ;  /*convert old elp value to index*/
            }
         }
u_lu[u+1] = u-l[u+1] ;

/* form (u+1)th discrepancy */
if (u<nn-kk)    /* no discrepancy computed on last iteration */
         {
if (s[u+1]!=-1)
d[u+1] = alpha_to[s[u+1]] ;
else
d[u+1] = 0 ;
for (i=1; i<=l[u+1]; i++)
if ((s[u+1-i]!=-1) && (elp[u+1][i]!=0))
              d[u+1] ^= alpha_to[(s[u+1-i]+index_of[elp[u+1][i]])%nn] ;
d[u+1] = index_of[d[u+1]] ;    /* put d[u+1] into index form */
         }
      } while ((u<nn-kk) && (l[u+1]<=tt)) ;
      u++ ;
if (l[u]<=tt)         /* can correct error */
       {
/* put elp into index form */
for (i=0; i<=l[u]; i++)
elp[u][i] = index_of[elp[u][i]] ;
/* find roots of the error location polynomial */
```

```
for (i=1; i<=l[u]; i++)
reg[i] = elp[u][i] ;
count = 0 ;
for (i=1; i<=nn; i++)
{
q = 1 ;
for (j=1; j<=l[u]; j++)
if (reg[j]!=-1)
{
reg[j] = (reg[j]+j)%nn ;
                q ^= alpha_to[reg[j]] ;
              } ;
if (!q)       /* store root and error location number indices */
{
root[count] = i;
loc[count] = nn-i ;
               count++ ;
             };
         } ;
if (count==l[u])    /* no. roots = degree of elp hence <= tt errors */
         {
/* form polynomial z(x) */
for (i=1; i<=l[u]; i++)       /* Z[0] = 1 always - do not need */
{
if ((s[i]!=-1) && (elp[u][i]!=-1))
z[i] = alpha_to[s[i]] ^ alpha_to[elp[u][i]] ;
else if ((s[i]!=-1) && (elp[u][i]==-1))
z[i] = alpha_to[s[i]] ;
else if ((s[i]==-1) && (elp[u][i]!=-1))
z[i] = alpha_to[elp[u][i]] ;
else
z[i] = 0 ;
for (j=1; j<i; j++)
if ((s[j]!=-1) && (elp[u][i-j]!=-1))
```

```
z[i] ^= alpha_to[(elp[u][i-j] + s[j])%nn] ;
z[i] = index_of[z[i]] ;         /* put into index form */
          } ;
  /* evaluate errors at locations given by error location numbers loc[i] */
for (i=0; i<nn; i++)
{
err[i] = 0 ;
if (recd[i]!=-1)        /* convert recd[] to polynomial form */
recd[i] = alpha_to[recd[i]] ;
else recd[i] = 0 ;
           }
for (i=0; i<l[u]; i++)    /* compute numerator of error term first */
{
err[loc[i]] = 1;       /* accounts for z[0] */
for (j=1; j<=l[u]; j++)
if (z[j]!=-1)
err[loc[i]] ^= alpha_to[(z[j]+j*root[i])%nn] ;
if (err[loc[i]]!=0)
{
err[loc[i]] = index_of[err[loc[i]]] ;
              q = 0 ;     /* form denominator of error term */
for (j=0; j<l[u]; j++)
if (j!=i)
                  q += index_of[1^alpha_to[(loc[j]+root[i])%nn]] ;
              q = q % nn ;
err[loc[i]] = alpha_to[(err[loc[i]]-q+nn)%nn] ;
recd[loc[i]] ^= err[loc[i]] ;  /*recd[i] must be in polynomial form */
             }
           }
         }
else    /* no. roots != degree of elp =>>tt errors and cannot solve */
for (i=0; i<nn; i++)        /* could return error flag if desired */
if (recd[i]!=-1)        /* convert recd[] to polynomial form */
recd[i] = alpha_to[recd[i]] ;
```

```
else  recd[i] = 0 ;     /* just output received codeword as is */
      }
else        /* elp has degree has degree >tt hence cannot solve */
for (i=0; i<nn; i++)       /* could return error flag if desired */
if (recd[i]!=-1)        /* convert recd[] to polynomial form */
recd[i] = alpha_to[recd[i]] ;
else  recd[i] = 0 ;     /* just output received codeword as is */
   }
else       /* no non-zero syndromes => no errors: output received codeword */
for (i=0; i<nn; i++)
if (recd[i]!=-1)        /* convert recd[] to polynomial form */
recd[i] = alpha_to[recd[i]] ;
else  recd[i] = 0 ;
 }
main()
{
register int i;
/* generate the Galois Field GF(2**mm) */
generate_gf() ;
printf("Look-up tables for GF(2**%2d)\n",mm) ;
printf("  i   alpha_to[i]  index_of[i]\n") ;
for (i=0; i<=nn; i++)
printf("%3d      %3d          %3d\n",i,alpha_to[i],index_of[i]) ;
printf("\n\n") ;
/* compute the generator polynomial for this RS code */
gen_poly() ;
/* for known data, stick a few numbers into a zero codeword. Data is in
polynomial form.
*/
for  (i=0; i<kk; i++)   data[i] = 0 ;
/* for example, say we transmit the following message (nothing special!) */
data[0] = 8 ;
data[1] = 6 ;
data[2] = 8 ;
```

```
data[3] = 1 ;
data[4] = 2 ;
data[5] = 4 ;
data[6] = 15 ;
data[7] = 9 ;
data[8] = 9 ;
/* encode data[] to produce parity in bb[].  Data input and parity output
is in polynomial form
*/
encode_rs() ;
/* put the transmitted codeword, made up of data plus parity, in recd[] */
for (i=0; i<nn-kk; i++)  recd[i] = bb[i] ;
for (i=0; i<kk; i++) recd[i+nn-kk] = data[i] ;
/* if you want to test the program, corrupt some of the elements of recd[] here. This can also be done easily in a debugger. */
/* again, lets say that a middle element is changed */
data[nn-nn/2] = 3 ;
for (i=0; i<nn; i++)
recd[i] = index_of[recd[i]] ;          /* put recd[i] into index form */
/* decode recv[] */
decode_rs() ;         /* recd[] is returned in polynomial form */
/* print out the relevant stuff - initial and decoded {parity and message} */
printf("Results for Reed-Solomon code (n=%3d, k=%3d, t= %3d)\n\n",nn,kk,tt) ;
printf("  i  data[i]   recd[i](decoded)   (data, recd in polynomial form)\n");
for (i=0; i<nn-kk; i++)
printf("%3d    %3d      %3d\n",i, bb[i], recd[i]) ;
for (i=nn-kk; i<nn; i++)
printf("%3d    %3d      %3d\n",i, data[i-nn+kk], recd[i]) ;
}
```

CHAPTER 8:RESULT

The encoder and decoder circuit of Reed Solomon code (15, 9) is successfully implemented on DSP kit. The result has been obtained after running the program on DSP kit is given below:

Look-up tables for GF (2** 4)

i	alpha_to[i]	index_of[i]
0	1	-1
1	2	0
2	4	1
3	8	4
4	3	2
5	6	8
6	12	5
7	11	10
8	5	3
9	10	14
10	7	9
11	14	7
12	15	6
13	13	13
14	9	11
15	10616849	12

Results for Reed-Solomon code (n= 15, k= 9, t= 3)

i	data[i]	recd[i](decoded)
0	15	15
1	13	13
2	10	10
3	0	0
4	6	6
5	13	13
6	8	8
7	6	6
8	8	8
9	1	1
10	2	2
11	4	4
12	15	15
13	9	9
14	3	9

CHAPTER 9:CONCLUSION

In this project we have implemented the encoding and decoding circuit of Reed Solomon code successfully on DSP kit TMS32C677XX. The RS (15, 9) is developed in this. Systematic encoder is used for encoding and the Berlekamp-Massey Algorithm for decoding. The advantage of this algorithm is that there is no need to calculate the error values, as it is enough to determine the position of the errors to perform the error correction. This algorithm is works in iterative manner, so it is quite complex. We can work on Euclidean Decoder in future to remove the iterative complexity and can compare their performance.

REFERENCES

Blahut, R.E. (1983) *Theory and Practice of Error Control Codes. Reading: Addison-*Wesley Pub. Co.

Blahut, R.E. (2003) *Algebraic codes for data transmission. Cambridge: University* Press.

Bose, R. (2008) *Information theory, coding and cryptography. 2nd ed. New Delhi: Tata* McGraw-Hill Publishing Company Ltd.

Bossert, M. (1999) *Channel Coding For Telecommunications. New York: John Wiley &* Sons.

Geisel, W.A. (1990) *Tutorial on Reed-Solomon error correction coding. Houston:*