CHAPTER- 6
CONCEPTS OF ERROR CONTROL CODING -- BLOCK CODES

The earlier chapters have given you enough background of Information theory and Source encoding. In this chapter you will be introduced to another important signal-processing operation, namely, “Channel Encoding”, which is used to provide ‘reliable’ transmission of information over the channel. In particular, we present, in this and subsequent chapters, a survey of ‘Error control coding’ techniques that rely on the systematic addition of ‘Redundant’ symbols to the transmitted information so as to facilitate two basic objectives at the receiver: ‘Error-detection’ and ‘Error correction’. We begin with some preliminary discussions highlighting the role of error control coding.

6.1 Rationale for Coding:

The main task required in digital communication is to construct ‘cost effective systems’ for transmitting information from a sender (one end of the system) at a rate and a level of reliability that are acceptable to a user (the other end of the system). The two key parameters available are transmitted signal power and channel bandwidth. These two parameters along with power spectral density of noise determine the signal energy per bit to noise power density ratio, \( E_b/N_0 \) and this ratio, as seen in chapter 4, uniquely determines the bit error for a particular scheme and we would like to transmit information at a rate \( R_{Max} = 1.443 \frac{S}{N} \). Practical considerations restrict the limit on \( E_b/N_0 \) that we can assign. Accordingly, we often arrive at modulation schemes that cannot provide acceptable data quality (i.e. low enough error performance). For a fixed \( E_b/N_0 \), the only practical alternative available for changing data quality from problematic to acceptable is to use “coding”.

Another practical motivation for the use of coding is to reduce the required \( E_b/N_0 \) for a fixed error rate. This reduction, in turn, may be exploited to reduce the required signal power or reduce the hardware costs (example: by requiring a smaller antenna size).

The coding methods discussed in chapter 5 deals with minimizing the average word length of the codes with an objective of achieving the lower bound viz. \( H(S) / \log r \), accordingly, coding is termed “entropy coding”. However, such source codes cannot be adopted for direct transmission over the channel. We shall consider the coding for a source having four symbols with probabilities \( p(s_1) = 1/2, p(s_2) = 1/4, p(s_3) = p(s_4) = 1/8 \). The resultant binary code using Huffman’s procedure is:

\[
\begin{align*}
    s_1 \ldots \ldots & 0 & s_3 \ldots & 1 & 1 & 0 \\
    s_2 \ldots \ldots & 10 & s_4 \ldots & 1 & 1 & 1
\end{align*}
\]

Clearly, the code efficiency is 100% and \( L = 1.75 \text{ bints/sym} = H(S) \). The sequence \( s_3s_2s_1 \) will then correspond to 11101110. Suppose a one-bit error occurs so that the received sequence is 0101110. This will be decoded as “1s_3s_2s_1”, which is altogether different than the transmitted sequence. Thus although the coding provides 100% efficiency in the light of Shannon’s theorem, it suffers a major disadvantage. Another disadvantage of a ‘variable length’ code lies in the fact that output data rates measured over short time periods will fluctuate widely. To avoid this problem, buffers of large length will be needed both at the encoder and at the decoder to store the variable rate bit stream if a fixed output rate is to be maintained.

Some of the above difficulties can be resolved by using codes with “fixed length”. For example, if the codes for the example cited are modified as 000, 100, 110, and 111. Observe that even if there is a one-bit error, it affects only one “block” and that the output data rate will not fluctuate.
The encoder/decoder structure using ‘fixed length’ code words will be very simple compared to the complexity of those for the variable length codes.

Hereafter, we shall mean by “Block codes”, the fixed length codes only. Since as discussed above, single bit errors lead to ‘single block errors’, we can devise means to detect and correct these errors at the receiver. Notice that the price to be paid for the efficient handling and easy manipulations of the codes is reduced efficiency and hence increased redundancy.

In general, whatever be the scheme adopted for transmission of digital/analog information, the probability of error is a function of signal-to-noise power ratio at the input of a receiver and the data rate. However, the constraints like maximum signal power and bandwidth of the channel (mainly the Governmental regulations on public channels) etc, make it impossible to arrive at a signaling scheme which will yield an acceptable probability of error for a given application. The answer to this problem is then the use of ‘error control coding’, also known as ‘channel coding’. In brief, “error control coding is the calculated addition of redundancy”. The block diagram of a typical data transmission system is shown in Fig. 6.1

The information source can be either a person or a machine (a digital computer). The source output, which is to be communicated to the destination, can be either a continuous waveform or a sequence of discrete symbols. The ‘source encoder’ transforms the source output into a sequence of binary digits, the information sequence $u$. If the source output happens to be continuous, this involves A-D conversion as well. The source encoder is ideally designed such that (i) the number of bits per unit time (bit rate, $r_b$) required to represent the source output is minimized (ii) the source output can be uniquely reconstructed from the information sequence $u$.

The ‘channel encoder’ transforms $u$ to the encoded sequence $v$, in general, a binary sequence, although non-binary codes can also be used for some applications. As discrete symbols are not suited for transmission over a physical channel, the code sequences are transformed to waveforms of specified durations. These waveforms, as they enter the channel get corrupted by noise. Typical channels include telephone lines, High frequency radio links, Telemetry links, Microwave links, and Satellite links and so on. Core and semiconductor memories, Tapes, Drums, disks, optical memory and so on are typical storage mediums. The switching impulse noise, thermal noise, cross talk and lightning are some examples of noise disturbance over a physical channel. A surface defect on a magnetic tape is a source of disturbance. The demodulator processes each received waveform and produces an output, which may be either continuous or discrete – the sequence $r$. The channel decoder transforms $r$ into a binary sequence, $\hat{u}$ which gives the estimate of $u$, and ideally should be the replica of $u$. The source decoder then transforms $\hat{u}$ into an estimate of source output and delivers this to the destination.

![Fig 6.1 Block diagram of a Typical data transmission system.](Image)
Error control for data integrity may be exercised by means of ‘forward error correction’ (FEC) where in the decoder performs error correction operation on the received information according to the schemes devised for the purpose. There is however another major approach known as ‘Automatic Repeat Request’ (ARQ), in which a re-transmission of the ambiguous information is effected, is also used for solving error control problems. In ARQ, error correction is not done at all. The redundancy introduced is used only for ‘error detection’ and upon detection, the receiver requests a repeat transmission which necessitates the use of a return path (feed back channel).

In summary, channel coding refers to a class of signal transformations designed to improve performance of communication systems by enabling the transmitted signals to better withstand the effect of various channel impairments such as noise, fading and jamming. Main objective of error control coding is to reduce the probability of error or reduce the $E_b/N_0$ at the cost of expending more bandwidth than would otherwise be necessary. Channel coding is a very popular way of providing performance improvement. Use of VLSI technology has made it possible to provide as much as $8-10\text{ dB}$ performance improvement through coding, at much lesser cost than through other methods such as high power transmitters or larger Antennas.

We will briefly discuss in this chapter the channel encoder and decoder strategies, our major interest being in the design and implementation of the channel ‘encoder/decoder’ pair to achieve fast transmission of information over a noisy channel, reliable communication of information and reduction of the implementation cost of the equipment.

6.2 Types of errors:

The errors that arise in a communication system can be viewed as ‘independent errors’ and ‘burst errors’. The first type of error is usually encountered by the ‘Gaussian noise’, which is the chief concern in the design and evaluation of modulators and demodulators for data transmission. The possible sources are the thermal noise and shot noise of the transmitting and receiving equipment, thermal noise in the channel and the radiations picked up by the receiving antenna. Further, in majority situations, the power spectral density of the Gaussian noise at the receiver input is white. The transmission errors introduced by this noise are such that the error during a particular signaling interval does not affect the performance of the system during the subsequent intervals. The discrete channel, in this case, can be modeled by a Binary symmetric channel. These transmission errors due to Gaussian noise are referred to as ‘independent errors’ (or random errors).

The second type of error is encountered due to the ‘impulse noise’, which is characterized by long quiet intervals followed by high amplitude noise bursts (As in switching and lightning). A noise burst usually affects more than one symbol and there will be dependence of errors in successive transmitted symbols. Thus errors occur in bursts

6.3 Types of codes:

There are mainly two types of error control coding schemes – Block codes and convolutional codes, which can take care of either type of errors mentioned above.

In a block code, the information sequence is divided into message blocks of $k$ bits each, represented by a binary $k$-tuple, $u = (u_1, u_2 \ldots u_k)$ and each block is called a message. The symbol $u$, here, is used to denote a $k$ – bit message rather than the entire information sequence. The encoder then transforms $u$ into an $n$-tuple $v = (v_1, v_2 \ldots v_n)$. Here $v$ represents an encoded block rather than the entire encoded sequence. The blocks are independent of each other.
The encoder of a convolutional code also accepts \( k \)-bit blocks of the information sequence \( u \) and produces an \( n \)-symbol block \( v \). Here \( u \) and \( v \) are used to denote sequences of blocks rather than a single block. Further each encoded block depends not only on the present \( k \)-bit message block but also on \( m \)-previous blocks. Hence the encoder has a memory of order ‘\( m \)’. Since the encoder has memory, implementation requires sequential logic circuits.

If the code word with \( n \)-bits is to be transmitted in no more time than is required for the transmission of the \( k \)-information bits and if \( \tau_b \) and \( \tau_c \) are the bit durations in the encoded and coded words, i.e. the input and output code words, then it is necessary that

\[
n.\tau_c = k.\tau_b
\]

We define the “rate of the code” by (also called rate efficiency)

\[
R_c = \frac{k}{n}
\]

Accordingly, with \( f_b = \frac{1}{\tau_b} \) and \( f_c = \frac{1}{\tau_c} \), we have \( \frac{f_b}{f_c} = \frac{\tau_c}{\tau_b} = \frac{k}{n} = R_c \)

### 6.4 Example of Error Control Coding:

Better way to understand the important aspects of error control coding is by way of an example. Suppose that we wish transmit data over a telephone link that has a useable bandwidth of 4 KHZ and a maximum SNR at the out put of 12 dB, at a rate of 1200 bits/sec with a probability of error less than \( 10^{-3} \). Further, we have DPSK modem that can operate at speeds of 1200, 1400 and 3600 bits/sec with error probabilities \( 2 \times (10^{-3}) \), \( 4 \times (10^{-3}) \) and \( 8 \times (10^{-3}) \) respectively. We are asked to design an error control coding scheme that would yield an overall probability of error < \( 10^{-3} \). We have:

\[
C > 16300 \text{ bits/sec, } R_c = 1200, 2400 \text{ or } 3600 \text{ bits/sec.}
\]

\([C=Blog_2 (1+ \frac{S}{N}), \frac{S}{N} = 12dB \text{ or } 15.85 , B=4\text{KHZ}], p = 2(10^{-3}), 4(10^{-3}) \text{ and } 8(10^{-3}) \) respectively. Since \( R_c < C \), according to Shannon’s theorem, we should be able to transmit data with arbitrarily small probability of error. We shall consider two coding schemes for this problem.

(i) **Error detection**: Single parity check-coding. Consider the \((4, 3)\) even parity check code.

<table>
<thead>
<tr>
<th>Message</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Codeword</td>
<td>0000</td>
<td>0011</td>
<td>0101</td>
<td>0110</td>
<td>1001</td>
<td>1010</td>
<td>1100</td>
<td>1111</td>
</tr>
</tbody>
</table>

Parity bit appears at the right most symbol of the codeword.

This code is capable of ‘detecting’ all single and triple error patterns. Data comes out of the channel encoder at a rate of 3600 bits/sec and at this rate the modem has an error probability of \( 8 \times (10^{-3}) \). The decoder indicates an error only when parity check fails. This happens for single and triple errors only.
\( p_d = \) Probability of error detection.

\[ = p(X = 1) + p(X = 3), \text{ where } X = \text{ Random variable of errors.} \]

Using binomial probability law, we have with \( p = 8(10^{-3}) \):

\[
P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}
\]

\[
p_d = \left(\frac{4}{1}\right)p(1 - p)^3 + \left(\frac{4}{3}\right)p^3(1 - p), \quad \left(\frac{4}{1}\right) = 4C_1 = 4, \left(\frac{4}{3}\right) = 4C_3 = 4
\]

Expanding we get \( p_d = 4p - 12p^2 + 16p^3 - 8p^4 \)

Substituting the value of \( p \) we get:

\[
p_d = 32 \times (10^{-3}) - 768 \times (10^{-6}) + 8192 \times (10^{-9}) - 32768 \times (10^{-12}) = 0.031240326 > > (10^{-3})
\]

However, an error results if the decoder does not indicate any error when an error indeed has occurred. This happens when two or 4 errors occur. Hence probability of a detection error = \( p_{nd} \) (probability of no detection) is given by:

\[
p_{nd} = P(X = 2) + P(X = 4) = \left(\frac{4}{2}\right)p^2(1 - p)^2 + \left(\frac{4}{4}\right)p^4(1 - p)^0 = 6p^2 - 12p^3 + 7p^4
\]

Substituting the value of \( p \) we get \( p_{nd} = 0.4 \times 10^{-3} < 10^{-3} \)

Thus probability of error is less than \( 10^{-3} \) as required.

(ii) **Error Correction:** The triplets 000 and 111 are transmitted whenever 0 and 1 are inputted. A majority logic decoding, as shown below, is employed assuming only single errors.

<table>
<thead>
<tr>
<th>Received Triplet</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>100</th>
<th>011</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output message</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Probability of decoding error, \( p_{dc} = P \) (two or more bits in error)

\[
= \left(\frac{3}{2}\right)p^2(1-p) + \left(\frac{3}{3}\right)p^3(1-p)^0 = 3p^2 - 2p^3
\]

\[= 190.464 \times 10^{-6} = 0.19 \times 10^{-3} < p = 10^{-3} \]
Probability of no detection, \( p_{nd} = P(\text{All 3 bits in error}) = p^3 = 512 \times 10^{-9} < p_{de} \)

In general observe that probability of no detection, \( p_{nd} < < \) probability of decoding error, \( p_{de} \).

The preceding examples illustrate the following aspects of error control coding. Note that in both examples with out error control coding the probability of error = \( 8 \times (10^{-3}) \) of the modem.

1. It is possible to detect and correct errors by adding extra bits—the check bits, to the message sequence. Because of this, not all sequences will constitute bonafide messages.

2. It is not possible to detect and correct all errors.

3. Addition of check bits reduces the effective data rate through the channel.

4. Since probability of no detection is always very much smaller than the decoding error probability, it appears that the error detection schemes, which do not reduce the rate efficiency as the error correcting schemes do, are well suited for our application. Since error detection schemes always go with ARQ techniques, and when the speed of communication becomes a major concern, Forward error correction (FEC) using error correction schemes would be desirable.

6.5 Block codes:

We shall assume that the output of an information source is a sequence of Binary digits. In ‘Block coding’ this information sequence is segmented into ‘message’ blocks of fixed length, say \( k \). Each message block, denoted by \( u \) then consists of \( k \) information digits. The encoder transforms these \( k \)-tuples into blocks of code words \( v \), each an \( n \)-tuple ‘according to certain rules’. Clearly, corresponding to \( 2^k \) information blocks possible, we would then have \( 2^k \) code words of length \( n > k \). This set of \( 2^k \) code words is called a “Block code”. For a block code to be useful these \( 2^k \) code words must be distinct, i.e. there should be a one-to-one correspondence between \( u \) and \( v \). \( u \) and \( v \) are also referred to as the ‘input vector’ and ‘code vector’ respectively. Notice that encoding equipment must be capable of storing the \( 2^k \) code words of length \( n > k \). Accordingly, the complexity of the equipment would become prohibitory if \( n \) and \( k \) become large unless the code words have a special structural property conducive for storage and mechanization. This structural is the ‘linearity’.

6.5.1 Linear Block Codes:

A block code is said to be linear \((n,k)\) code if and only if the \( 2^k \) code words from a \( k \)-dimensional sub space over a vector space of all \( n \)-Tuples over the field \( GF(2) \).

Fields with \( 2^m \) symbols are called ‘Galois Fields’ (pronounced as Galva fields), \( GF(2^m) \). Their arithmetic involves binary additions and subtractions. For two valued variables, \((0, 1)\).The modulo – 2 addition and multiplication is defined in Fig 6.3.

![Fig 6.3](image)

![Fig 6.4](image)
The binary alphabet \((0, 1)\) is called a field of two elements (a binary field and is denoted by \(GF(2)\). Notice that \(\oplus\) represents the EX-OR operation and \(\otimes\) represents the AND operation). Further in binary arithmetic, \(-X = X\) and \(X - Y = X \oplus Y\). Similarly for 3-valued variables, modulo \(-3\) arithmetic can be specified as shown in Fig 6.4. However, for brevity while representing polynomials involving binary addition we use \(+\) instead of \(\oplus\) and there shall be no confusion about such usage.

Polynomials \(f(X)\) with 1 or 0 as the coefficients can be manipulated using the above relations. The arithmetic of \(GF(2^m)\) can be derived using a polynomial of degree ‘\(m\)’, with binary coefficients and using a new variable \(\alpha\) called the primitive element, such that \(p(\alpha) = 0\). When \(p(X)\) is irreducible (i.e. it does not have a factor of degree \(< m\) and \(> 0\), for example \(X^3 + X + 1, X^3 + X^2 + 1, X^4 + X^3 + 1, X^5 + X + 1\) etc. are irreducible polynomials, whereas \(f(X) = X^4 + X^3 + X^2 + 1\) is not as \(f(1) = 0\) and hence has a factor \(X+1\)) then \(p(X)\) is said to be a ‘primitive polynomial’.

If \(v_n\) represents a vector space of all \(n\)-tuples, then a subset \(S\) of \(v_n\) is called a subspace if (i) the all Zero vector is in \(S\) (ii) the sum of any two vectors in \(S\) is also a vector in \(S\). To be more specific, a block code is said to be linear if the following is satisfied. “If \(v_1\) and \(v_2\) are any two code words of length \(n\) of the block code then \(v_1 \oplus v_2\) is also a code word length \(n\) of the block code”.

**Example 6.1:** Linear Block code with \(k = 3\), and \(n = 6\)

<table>
<thead>
<tr>
<th>Messages</th>
<th>Code words</th>
<th>Weight (No. of 1’s in the code word)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m_1) 000</td>
<td>(v_1) 0 0 0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>(m_2) 001</td>
<td>(v_2) 0 0 1 1 1 0</td>
<td>3</td>
</tr>
<tr>
<td>(m_3) 010</td>
<td>(v_3) 0 1 0 1 0 1</td>
<td>3</td>
</tr>
<tr>
<td>(m_4) 100</td>
<td>(v_4) 1 0 0 0 1 1</td>
<td>3</td>
</tr>
<tr>
<td>(m_5) 011</td>
<td>(v_5) 0 1 1 0 1 1</td>
<td>4</td>
</tr>
<tr>
<td>(m_6) 101</td>
<td>(v_6) 1 0 1 1 0 1</td>
<td>4</td>
</tr>
<tr>
<td>(m_7) 110</td>
<td>(v_7) 1 1 0 1 1 0</td>
<td>4</td>
</tr>
<tr>
<td>(m_8) 111</td>
<td>(v_8) 1 1 1 0 0 0</td>
<td>3</td>
</tr>
</tbody>
</table>

Observe the linearity property: With \(v_3 = (010 101)\) and \(v_4 = (100 011)\), \(v_3 \oplus v_4 = (110 110) = v_7\).

Remember that \(n\) represents the word length of the code words and \(k\) represents the number of information digits and hence the block code is represented as \((n, k)\) block code.

Thus by definition of a linear block code it follows that if \(g_1, g_2, \ldots g_k\) are the \(k\) linearly independent code words then every code vector, \(v\), of our code is a combination of these code words, i.e.

\[
v = u_1 g_1 \oplus u_2 g_2 \oplus \ldots \oplus u_k g_k
\]

\[\text{..................}
\]

(6.1)

Where \(u_j = 0\) or 1, \(\forall\ 1 \leq j \leq k\)

Eq (6.1) can be arranged in matrix form by nothing that each \(g_j\) is an \(n\)-tuple, i.e.

\[
g_j = (g_{j1}, g_{j2}, \ldots g_{jn})
\]

\[\text{..................}
\]

(6.2)
Thus we have \[ v = u G \] \hspace{1cm} (6.3)

Where:
\[ u = (u_1, u_2, \ldots, u_k) \] \hspace{1cm} (6.4)

represents the data vector and
\[
G = \begin{bmatrix}
    g_{11} & g_{12} & \cdots & g_{1n} \\
    g_{21} & g_{22} & \cdots & g_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    g_{k1} & g_{k2} & \cdots & g_{kn}
\end{bmatrix}
\]

\hspace{1cm} (6.5)
is called the “generator matrix”.

Notice that any \( k \) linearly independent code words of an \((n, k)\) linear code can be used to form a Generator matrix for the code. Thus it follows that an \((n, k)\) linear code is completely specified by the \( k \)-rows of the generator matrix. Hence the encoder need only to store \( k \) rows of \( G \) and form linear combination of these rows based on the input message \( u \).

**Example 6.2:** The \((6, 3)\) linear code of Example 6.1 has the following generator matrix:

\[
G = \begin{bmatrix}
    g_1 \\
    g_2 \\
    g_3
\end{bmatrix}
= \begin{bmatrix}
    1 & 0 & 0 & 0 & 1 & 1 \\
    0 & 1 & 0 & 1 & 0 & 1 \\
    0 & 0 & 1 & 1 & 1 & 0
\end{bmatrix}
\]

If \( u = m_5 \) (say) is the message to be coded, i.e. \( u = (011) \)

We have \( v = u . G = 0.g_1 + 1.g_2 + 1.g_3 \)

\[ = (0, 0, 0, 0, 0, 0) + (0, 1, 0, 1, 0, 1) + (0, 0, 1, 1, 0, 1) = (0, 1, 1, 0, 1, 1) \]

Thus \( v = (0 1 1 0 1 1) \)

“\( v \) can be computed simply by adding those rows of \( G \) which correspond to the locations of 1’s of \( u \).”

**6.5.2 Systematic Block Codes (Group Property):**

A desirable property of linear block codes is the “Systematic Structure”. Here a code word is divided into two parts – Message part and the redundant part. If either the first \( k \) digits or the last \( k \) digits of the code word correspond to the message part then we say that the code is a “Systematic Block Code”. We shall consider systematic codes as depicted in Fig.6.5.

\[ \begin{array}{c|c}
\text{Message Part} & \text{Redundant Part} \\
\hline
\text{k digits} & \text{(u - k) digits} \\
\end{array} \]

*Fig 6.5 Systematic Format of code word.*

In the format of Fig.6.5 notice that:
\[ v_1 = u_1, v_2 = u_2, v_3 = u_3 \ldots v_k = u_k \]
\[ \vdots \]
\[ v_{k+1} = u_1p_{11} + u_2p_{21} + u_3p_{31} + \ldots + u_kp_{k1} \]
\[ v_{k+2} = u_1p_{12} + u_2p_{22} + u_3p_{32} + \ldots + u_kp_{k2} \]
\[ \vdots \]
\[ v_n = u_1p_{1,n-k} + u_2p_{2,n-k} + u_3p_{3,n-k} + \ldots + u_kp_{k,n-k} \]
\[ \text{.................} \quad (6.6) \]

Or in matrix form we have
\[
\begin{bmatrix}
v_1 & v_2 & \ldots & v_k & v_{k+1} & \ldots & v_n \\
\end{bmatrix} = 
\begin{bmatrix}
1 & 0 & 0 & \ldots & 0 & p_{11} & p_{12} & \ldots & p_{1,n-k} \\
0 & 1 & 0 & \ldots & 0 & p_{21} & p_{22} & \ldots & p_{2,n-k} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \ldots & 1 & p_{k1} & p_{k2} & \ldots & p_{k,n-k} \\
\end{bmatrix} 
\text{.................} \quad (6.7)
\]

i.e., \( v = u \cdot G \)

Where \( G = [I_k, P] \)
\[ \text{.................} \quad (6.8) \]

Where \( P = 
\begin{bmatrix}
p_{11} & p_{12} & \ldots & p_{1,n-k} \\
p_{21} & p_{22} & \ldots & p_{2,n-k} \\
\vdots & \vdots & \ddots & \vdots \\
p_{k1} & p_{k2} & \ldots & p_{k,n-k} \\
\end{bmatrix} 
\text{.................} \quad (6.9) \]

\( I_k \) is the \( k \times k \) identity matrix (unit matrix), \( P \) is the \( k \times (n-k) \) ‘parity generator matrix’, in which \( p_{i,j} \) are either 0 or 1 and \( G \) is a \( k \times n \) matrix. The \( (n-k) \) equations given in Eq (6.6b) are referred to as parity check equations. Observe that the \( G \) matrix of Example 6.2 is in the systematic format. The \( n \)-vectors \( a = (a_1, a_2, \ldots a_n) \) and \( b = (b_1, b_2, \ldots b_n) \) are said to be orthogonal if their inner product defined by:

\[ a \cdot b = (a_1, a_2, \ldots a_n) (b_1, b_2, \ldots b_n)^T = 0. \]

where, ‘\(^T\)’ represents transposition. Accordingly for any \( k \times n \) matrix, \( G \), with \( k \) linearly independent rows there exists a \((n-k) \times n\) matrix \( H \) with \((n-k)\) linearly independent rows such that any vector in the row space of \( G \) is orthogonal to the rows of \( H \) and that any vector that is orthogonal to the rows of \( H \) is in the row space of \( G \). Therefore, we can describe an \((n, k)\) linear code generated by \( G \) alternatively as follows:

\[
\text{“An n – tuple, } v \text{ is a code word generated by } G, \text{ if and only if } v \cdot H^T = 0”. \quad (6.9a)
\]

(O represents an all zero row vector.)

This matrix \( H \) is called a ‘parity check matrix’ of the code. Its dimension is \((n-k) \times n\).

If the generator matrix has a systematic format, the parity check matrix takes the following form.

\[ H = [P^T]_{n-k} = 
\begin{bmatrix}
p_{11} & p_{21} & \ldots & p_{k1} & 1 & 0 & 0 & \ldots & 0 \\
p_{12} & p_{22} & \ldots & p_{k2} & 0 & 1 & 0 & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
p_{1,n-k} & p_{2,n-k} & \ldots & p_{k,n-k} & 0 & 0 & 0 & \ldots & 1 \\
\end{bmatrix} 
\text{.................} \quad (6.10) \]
The $i^{th}$ row of $G$ is:

$$g_i = (0 \ 0 \ \ldots \ 0 \ p_{i,1} \ p_{i,2} \ \ldots \ p_{i,n-k})$$

The $j^{th}$ row of $H$ is:

$$h_j = (p_{1,j} \ p_{2,j} \ \ldots \ p_{i,j} \ \ldots \ p_{k,j} \ 0 \ 0 \ 0 \ 0 \ 0)$$

Accordingly the inner product of the above $n$ – vectors is:

$$g_i \times h_j^T = (0 \ 0 \ \ldots \ 0 \ p_{i,1} \ p_{i,2} \ \ldots \ p_{i,n-k}) \ (p_{1,j} \ p_{2,j} \ \ldots \ p_{i,j} \ \ldots \ p_{k,j} \ 0 \ 0 \ 0 \ 0 \ 0)^T$$

$$= p_{ij} + p_{ij} = 0 \ \text{(as the } p_{ij} \text{ are either 0 or 1 and in modulo – 2 arithmetic } X + X = 0)$$

This implies simply that:

$$G \cdot H^T = O_{k \times (n-k)}$$

(6.11)

Where $O_{k \times (n-k)}$ is an all zero matrix of dimension $k \times (n - k)$. 

Further, since the $(n - k)$ rows of the matrix $H$ are linearly independent, the $H$ matrix of Eq. (6.10) is a parity check matrix of the $(n, k)$ linear systematic code generated by $G$. Notice that the parity check equations of Eq. (6.6b) can also be obtained from the parity check matrix using the fact $v \cdot H^T = O$.

**Alternative Method of proving $v \cdot H^T = O$:**

We have $v = u \cdot G = u \cdot [I_k : P] = [u_1, u_2, \ldots, u_k, p_1, p_2, \ldots, p_{n-k}]$
Where \( p_i = (u_1 p_{1,i} + u_2 p_{2,i} + u_3 p_{3,i} \ldots + u_k p_{k,i}) \) are the parity bits found from Eq (6.6b).

Now \( H^T = \begin{bmatrix} P \\ I_{n-k} \end{bmatrix} \)

\[
\therefore v.H^T = [u_1 p_{11} + u_2 p_{21} + \ldots + u_k p_{k1} + p_1, u_1 p_{12} + u_2 p_{22} + \ldots + u_k p_{k2} + p_2, \ldots \]
\[
= [p_1 + p_1, p_2 + p_2 \ldots p_{n-k} + p_{n-k}] 
\]

Thus \( v.H^T = O \). This statement implies that an \( n \)-Tuple \( v \) is a code word generated by \( G \) if and only if \( v.H^T = O \)

Since \( v = u.G \), This means that: \( u.G.H^T = O \)

If this is to be true for any arbitrary message vector \( v \) then this implies: \( G.H^T = O_{k \times (n-k)} \)

**Example 6.3:**

Consider the generator matrix of Example 6.2, the corresponding parity check matrix is

\[
H = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix}
\]

**6.5.3 Syndrome and Error Detection:**

Suppose \( v = (v_1, v_2, \ldots, v_n) \) be a code word transmitted over a noisy channel and let: \( r = (r_1, r_2, \ldots, r_n) \) be the received vector. Clearly, \( r \) may be different from \( v \) owing to the channel noise. The vector sum

\[
e = r - v = (e_1, e_2, \ldots, e_n)
\]

is an \( n \)-tuple, where \( e_j = 1 \) if \( r_j \neq v_j \) and \( e_j = 0 \) if \( r_j = v_j \). This \( n \)-tuple is called the “error vector” or “error pattern”. The \( 1 \)'s in \( e \) are the transmission errors caused by the channel noise. Hence from Eq (6.12) it follows:

\[
r = v \oplus e
\]

Observe that the receiver noise does not know either \( v \) or \( e \). Accordingly, on reception of \( r \) the decoder must first identify if there are any transmission errors and, then take action to locate these errors and correct them (**FEC** – Forward Error Correction) or make a request for re-transmission (**ARQ**). When \( r \) is received, the decoder computes the following \((n-k)\) tuple:

\[
s = r.H^T = (s_1, s_2, \ldots, s_{n-k})
\]
It then follows from Eq (6.9a), that \( s = 0 \) if and only if \( r \) is a code word and \( s \neq 0 \) iffy \( r \) is not a code word. This vector \( s \) is called “The Syndrome” (a term used in medical science referring to collection of all symptoms characterizing a disease). Thus if \( s = 0 \), the receiver accepts \( r \) as a valid code word. Notice that there are possibilities of errors undetected, which happens when \( e \) is identical to a nonzero code word. In this case \( r \) is the sum of two code words which according to our linearity property is again a code word. This type of error pattern is referred to an “undetectable error pattern”. Since there are \( 2^k -1 \) nonzero code words, it follows that there are \( 2^k -1 \) error patterns as well. Hence when an undetectable error pattern occurs the decoder makes a “decoding error”. Eq. (6.13) can be expanded as below:

\[
\begin{bmatrix}
p_{11} & p_{12} & \cdots & p_{1,2^k-1} \\
p_{21} & p_{22} & \cdots & p_{2,2^k-1} \\
\vdots & \vdots & \ddots & \vdots \\
p_{k1} & p_{k,2} & \cdots & p_{k,2^k-1} \\
1 & 0 & \cdots & 0 \\
0 & 1 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 1 \\
\end{bmatrix}
\begin{bmatrix}
r_1 \\
r_2 \\
\vdots \\
r_k \\
r_{k+1} \\
r_{k+2} \\
\vdots \\
r_n \\
\end{bmatrix}
\begin{bmatrix}
r_k+p_{1,k}+r_{k+1}+\cdots+r_{n-k} \\
r_{k+1}+p_{2,k+2}+\cdots+r_{n-k} \\
\vdots \\
r_{n-k}+p_{n-k,n-k}+r_{n} \\
\end{bmatrix}
\]

From which we have

\[
\begin{align*}
s_1 &= r_1 + r_2 p_{21} + \cdots + r_k p_{k1} + r_{k+1} \\
s_2 &= r_1 + r_2 p_{22} + \cdots + r_k p_{k2} + r_{k+2} \\
\vdots & \quad \vdots \\
s_{n-k} &= r_1 p_{1,n-k} + r_2 p_{2,n-k} + \cdots + r_k p_{k,n-k} + r_n
\end{align*}
\]

A careful examination of Eq. (6.14) reveals the following point. The syndrome is simply the vector sum of the received parity digits \((r_{k+1}, r_{k+2}, \ldots, r_n)\) and the parity check digits recomputed from the received information digits \((r_1, r_2, \ldots, r_n)\).

**Example 6.4:**

We shall compute the syndrome for the \((6, 3)\) systematic code of Example 6.2. We have

\[
s = (s_1, s_2, s_3) = (r_1, r_2, r_3, r_4, r_5, r_6)
\]

or \( s_1 = r_2 + r_3 + r_4 \)
\( s_2 = r_1 + r_3 + r_5 \)
\( s_3 = r_1 + r_2 + r_6 \)

In view of Eq. (6.12a), and Eq. (6.9a) we have

\[
s = r . H^T = (v \oplus e) H^T
\]

\[
= v . H^T \oplus e . H^T
\]
or \( s = e.H^T \).......................... (6.15)

as \( v.H^T = 0 \). Eq. (6.15) indicates that the syndrome depends only on the error pattern and not on the transmitted code word \( v \). For a linear systematic code, then, we have the following relationship between the syndrome digits and the error digits.

\[
\begin{align*}
\begin{bmatrix}
s_1 \\
s_2 \\
\vdots \\
s_{n-k}
\end{bmatrix} &= \begin{bmatrix}
e_1p_{11} + e_2p_{21} + \ldots + e_kp_{k,1} + e_{k+1} \\
e_1p_{12} + e_2p_{22} + \ldots + e_kp_{k,2} + e_{k+2} \\
\vdots \\
é_1p_{1,n-k} + e_2p_{2,n-k} + \ldots + e_kp_{k,n-k} + e_n
\end{bmatrix} \\
&= s \\
&= r.H^T = (1, 1, 0)
\end{align*}
\]

Thus, the syndrome digits are linear combinations of error digits. Therefore they must provide us information about the error digits and help us in error correction.

Notice that Eq. (6.16) represents \((n-k)\) linear equations for \(n\) error digits – an under-determined set of equations. Accordingly it is not possible to have a unique solution for the set. As the rank of the \(H\) matrix is \(k\), it follows that there are \(2^k\) non-trivial solutions. In other words there exist \(2^k\) error patterns that result in the same syndrome. Therefore to determine the true error pattern is not any easy task!

**Example 6.5:**

For the \((6, 3)\) code considered in Example 6 2, the error patterns satisfy the following equations:

\[
s_1 = e_2 + e_3 + e_4, \quad s_2 = e_1 + e_3 + e_5, \quad s_3 = e_1 + e_2 + e_6
\]

Suppose, the transmitted and received code words are \(v = (0 1 0 1 0 1), r = (0 1 1 1 0 1)\)

Then \(s = r.H^T = (1, 1, 0)\)

Then it follows that:

\[
\begin{align*}
e_2 + e_3 + e_4 &= 1 \\
e_1 + e_3 + e_5 &= 1 \\
e_1 + e_2 + e_6 &= 0
\end{align*}
\]

There are \(2^3 = 8\) error patterns that satisfy the above equations. They are:

\[
\{0 0 0 1 0 0 0, 1 0 0 0 0, 0 0 0 1 1 0, 0 1 0 0 1 1, 1 0 0 1 0 1, 0 1 1 1 0 1, 1 0 1 0 1 1, 1 1 1 1 1 0\}
\]

To minimize the decoding error, the “Most probable error pattern” that satisfies Eq (6.16) is chosen as the true error vector. For a BSC, the most probable error pattern is the one that has the smallest number of nonzero digits. For the Example 6.5, notice that the error vector \((0 0 1 0 0 0)\) has the smallest number of nonzero components and hence can be regarded as the most probable error vector. Then using Eq. (6.12) we have

\[
\hat{v} = r \oplus e
\]

\[
=(0 1 1 1 0 1) + (0 0 1 0 0 0) = (0 1 0 1 0 1)
\]

Notice now that \(\hat{v}\) indeed is the actual transmitted code word.

**6.6 Minimum Distance Considerations:**
The concept of distance between code words and single error correcting codes was first developed by R. W. Hamming. Let the n-tuples, 
\[ \alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n), \quad \beta = (\beta_1, \beta_2, \ldots, \beta_n) \]
be two code words. The “Hamming distance” \( d(\alpha, \beta) \) between such pair of code vectors is defined as the number of positions in which they differ. Alternatively, using Modulo-2 arithmetic, we have
\[
d(\alpha, \beta) \triangleq \sum_{j=1}^{n} (\alpha_j \oplus \beta_j)
\]
(Notice that \( \sum \) represents the usual decimal summation and \( \oplus \) is the modulo-2 sum, the EX-OR function).

The “Hamming Weight” \( \omega(\alpha) \) of a code vector \( \alpha \) is defined as the number of nonzero elements in the code vector. Equivalently, the Hamming weight of a code vector is the distance between the code vector and the ‘all zero code vector’.

**Example 6.6:** Let \( \alpha = (0 1 1 1 0 1), \beta = (1 0 1 0 1 1) \)

Notice that the two vectors differ in 4 positions and hence \( d(\alpha, \beta) = 4 \). Using Eq (6.17) we find
\[
d(\alpha, \beta) = (0 \oplus 1) + (1 \oplus 0) + (1 \oplus 1) + (1 \oplus 0) + (0 \oplus 1) + (1 \oplus 1)
\]
\[
= 1 + 1 + 0 + 1 + 1 + 0
\]
\[
= 4 \ldots \quad \text{(Here + is the algebraic plus not modulo – 2 sum)}
\]

**Further,** \( \omega(\alpha) = 4 \) and \( \omega(\beta) = 4 \).

The “Minimum distance” of a linear block code is defined as the smallest Hamming distance between any pair of code words in the code or the minimum distance is the same as the smallest Hamming weight of the difference between any pair of code words. Since in linear block codes, the sum or difference of two code vectors is also a code vector, it follows then that “the minimum distance of a linear block code is the smallest Hamming weight of the nonzero code vectors in the code”.

The Hamming distance is a metric function that satisfies the triangle inequality. Let \( \alpha, \beta \) and \( \delta \) be three code vectors of a linear block code. Then
\[
d(\alpha, \beta) + d(\beta, \delta) \geq d(\alpha, \delta)
\]
(6.18)

From the discussions made above, we may write
\[
d(\alpha, \beta) = \omega(\alpha \oplus \beta)
\]
(6.19)

**Example 6.7:** For the vectors \( \alpha \) and \( \beta \) of Example 6.6, we have:
\[
\alpha \oplus \beta = (0 \oplus 1), (1 \oplus 0), (1 \oplus 1) (1 \oplus 0), (0 \oplus 1) (1 \oplus 1) = (1 1 0 1 1 0)
\]
\[ \omega(\alpha \oplus \beta) = 4 = d(\alpha, \beta) \]

If \( \delta = (10101010) \), we have \( d(\alpha, \beta) = 4; d(\beta, \delta) = 1; d(\alpha, \delta) = 5 \)

Notice that the above three distances satisfy the triangle inequality:

\[ d(\alpha, \beta) + d(\beta, \delta) = 5 = d(\alpha, \delta) \]
\[ d(\beta, \delta) + d(\alpha, \delta) = 6 > d(\alpha, \beta) \]
\[ d(\alpha, \delta) + d(\alpha, \beta) = 9 > d(\beta, \delta) \]

Similarly, the minimum distance of a linear block code, \( 'C' \) may be mathematically represented as below:

\[ d_{\text{min}} = \text{Min} \{d(\alpha, \beta) : \alpha, \beta \in C, \alpha \neq \beta \} \]
\[ = \text{Min} \{\omega(\alpha \oplus \beta) : \alpha, \beta \in C, \alpha \neq \beta \} \]
\[ = \text{Min} \{\omega(\alpha) : \alpha \in C, \alpha \neq 0 \} \]

That is \( d_{\text{min}} \triangleq \omega_{\text{min}} \). The parameter \( \omega_{\text{min}} \) is called the “minimum weight” of the linear code \( C \). The minimum distance of a code, \( d_{\text{min}} \), is related to the parity check matrix, \( H \), of the code in a fundamental way. Suppose \( v \) is a code word. Then from Eq. (6.9a) we have:

\[ 0 = v.H^T \]
\[ = v_1h_1 \oplus v_2h_2 \oplus \ldots \oplus v_nh_n \]

Here \( h_1, h_2 \ldots h_n \) represent the columns of the \( H \) matrix. Let \( v_{j1}, v_{j2} \ldots v_{jl} \) be the \( 'l' \) nonzero components of \( v \) i.e. \( v_{j1} = v_{j2} = \ldots = v_{jl} = 1 \). Then it follows that:

\[ h_{j1} \oplus h_{j2} \oplus \ldots \oplus h_{jl} = O^T \]

That is “if \( v \) is a code vector of Hamming weight \( 'l' \), then there exist \( 'l' \) columns of \( H \) such that the vector sum of these columns is equal to the zero vector”. Suppose we form a binary \( n \)-tuple of weight \( 'l' \), viz. \( x = (x_1, x_2 \ldots x_n) \) whose nonzero components are \( x_{j1}, x_{j2} \ldots x_{jl} \). Consider the product:

\[ x.H^T = x_1h_1 \oplus x_2h_2 \oplus \ldots \oplus x_nh_n = x_{j1}h_{j1} \oplus x_{j2}h_{j2} \oplus \ldots \oplus x_{jl}h_{jl} = h_{j1} \oplus h_{j2} \oplus \ldots \oplus h_{jl} \]

If Eq. (6.22) holds, it follows \( x.H^T = O \) and hence \( x \) is a code vector. Therefore, we conclude that “if there are \( 'l' \) columns of \( H \) matrix whose vector sum is the zero vector then there exists a code vector of Hamming weight \( 'l' \) ”.

From the above discussions, it follows that:

i) If no \((d-1)\) or fewer columns of \( H \) add to \( O^T \), the all zero column vector, the code has a minimum weight of at least \( 'd' \).

ii) The minimum weight (or the minimum distance) of a linear block code \( C \), is the smallest number of columns of \( H \) that sum to the all zero column vector.
For the H matrix of Example 6.3, i.e. $H = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$, notice that all columns of $H$ are non zero and distinct. Hence no two or fewer columns sum to zero vector. Hence the minimum weight of the code is at least 3. Further notice that the 1st, 2nd and 3rd columns sum to $\mathbf{0}^T$. Thus the minimum weight of the code is 3. We see that the minimum weight of the code is indeed 3 from the table of Example 6.1.

**6.6.1 Error Detecting and Error Correcting Capabilities:**

The minimum distance, $d_{\text{min}}$, of a linear block code is an important parameter of the code. To be more specific, it is the one that determines the error correcting capability of the code. To understand this we shall consider a simple example. Suppose we consider 3-bit code words plotted at the vertices of the cube as shown in Fig.6.10.

![Fig 6.10 The Distance concept](image_url)

Clearly, if the code words used are \{000, 101, 110, 011\}, the Hamming distance between the words is 2. Notice that any error in the received words locates them on the vertices of the cube which are not code words and may be recognized as single errors. The code word pairs with Hamming distance = 3 are: \(000, 111\), \(100, 011\), \(101, 010\) and \(001, 110\). If a code word \(000\) is received as \(100, 010, 001\), observe that these are nearer to \(000\) than to \(111\). Hence the decision is made that the transmitted word is \(000\).

Suppose an \((n, k)\) linear block code is required to detect and correct all error patterns (over a BSC), whose Hamming weight, $\omega \leq t$. That is, if we transmit a code vector $\alpha$ and the received vector is $\beta = \alpha \oplus e$, we want the decoder output to be $\hat{\alpha} = \alpha$ subject to the condition $\omega(e) \leq t$.

Further, assume that $2^k$ code vectors are transmitted with equal probability. The best decision for the decoder then is to pick the code vector nearest to the received vector $\beta$ for which the Hamming distance is the smallest. i.e., $d(\alpha, \beta)$ is minimum. With such a strategy the decoder will be able to detect and correct all error patterns of Hamming weight $\omega(e) \leq t$ provided that the minimum distance of the code is such that:

$$d_{\text{min}} \geq (2t + 1) \quad \text{.........................} \quad (6.23)$$

$d_{\text{min}}$ is either odd or even. Let ‘$t$’ be a positive integer such that
\[ 2t + 1 \leq \text{d}_{\text{min}} \leq 2t + 2 \]  

Suppose \( \delta \) be any other code word of the code. Then, the Hamming distances among \( \alpha, \beta \) and \( \delta \) satisfy the triangular inequality:

\[ d(\alpha, \beta) + d(\beta, \delta) \geq d(\alpha, \delta) \]  

Suppose an error pattern of ‘\( t' \)’ errors occurs during transmission of \( \alpha \). Then the received vector \( \beta \) differs from \( \alpha \) in ‘\( t' \)’ places and hence \( d(\alpha, \beta) = t' \). Since \( \alpha \) and \( \delta \) are code vectors, it follows from Eq. (6.24):

\[ d(\alpha, \delta) \geq \text{d}_{\text{min}} \geq 2t + 1 \]  

Combining Eq. (6.25) and (6.26) and with the fact that \( d(\alpha, \beta) = t' \), it follows that:

\[ d(\delta, \beta) \geq 2t + 1 - t' \]  

Hence if \( t' \leq t \), then: \( d(\delta, \beta) > t \)  

Eq 6.28 says that if an error pattern of ‘\( t' \)’ or fewer errors occurs, the received vector \( \beta \) is closer (in Hamming distance) to the transmitted code vector \( \alpha \) than to any other code vector \( \delta \) of the code. For a BSC, this means \( P(\beta | \alpha) > P(\beta | \delta) \) for \( \alpha \neq \delta \). Thus based on the maximum likelihood decoding scheme, \( \beta \) is decoded as \( \alpha \), which indeed is the actual transmitted code word and this results in the correct decoding and thus the errors are corrected.

On the contrary, the code is not capable of correcting error patterns of weight \( t > t' \). To show this we proceed as below:

Suppose \( d(\alpha, \delta) = d_{\text{min}} \) and let \( e_1 \) and \( e_2 \) be two error patterns such that:

\( i) \quad e_1 \oplus e_2 = \alpha \oplus \delta \)

\( ii) \quad e_1 \) and \( e_2 \) do not have nonzero components in common places. Clearly,

\[ \omega(e_1) + \omega(e_2) = \omega(\alpha \oplus \delta) = d(\alpha, \delta) = d_{\text{min}} \]  

Suppose, \( \alpha \) is the transmitted code vector and is corrupted by the error pattern \( e_1 \). Then the received vector is:

\[ \beta = \alpha \oplus e_1 \]  

and

\[ d(\alpha, \beta) = \omega(\alpha \oplus \beta) = \omega(e_1) \]  

\[ d(\delta, \beta) = \omega(\delta \oplus \beta) \]

\[ = \omega(\delta \oplus \alpha \oplus e_1) = \omega(e_2) \]

If the error pattern \( e_1 \) contains more than ‘\( t' \)’ errors, i.e. \( \omega(e_1) > t \), and since \( 2t + 1 \leq d_{\text{min}} \leq 2t + 2 \), it follows
\[ \omega(e_2) \leq t - 1 \]  \\
\[ \therefore d(\alpha, \beta) \geq d(\delta, \beta) \]  

(6.33)  

(6.34)

This inequality says that there exists an error pattern of \( l > t \) errors which results in a received vector closer to an incorrect code vector i.e. based on the maximum likelihood decoding scheme decoding error will be committed.

To make the point clear, we shall give yet another illustration. The code vectors and the received vectors may be represented as points in an \( n \)-dimensional space. Suppose we construct two spheres, each of equal radii, \( t \) around the points that represent the code vectors \( \alpha \) and \( \delta \). Further let these two spheres be mutually exclusive or disjoint as shown in Fig. 6.11 (a).

For this condition to be satisfied, we then require \( d(\alpha, \delta) \geq 2t + 1 \). In such a case if \( d(\alpha, \beta) \leq t \), it is clear that the decoder will pick \( \alpha \) as the transmitted vector.

\[ d(\alpha, \delta) \geq 2t + 1 \]  

\[ d(\alpha, \beta) \leq 2t \]

(a) \( d(\alpha, \delta) \geq 2t + 1 \)  

(b) \( d(\alpha, \delta) \leq 2t \)

Fig 6.11 Illustrating the distance concept

On the other hand, if \( d(\alpha, \delta) \leq 2t \), the two spheres around \( \alpha \) and \( \delta \) intersect and if ‘\( \beta \)’ is located as in Fig. 6.11 (b), and \( \alpha \) is the transmitted code vector it follows that even if \( d(\alpha, \beta) \leq t \), yet \( \beta \) is as close to \( \delta \) as it is to \( \alpha \). The decoder can now pick \( \delta \) as the transmitted vector which is wrong. Thus it is imminent that “an \((n, k)\) linear block code has the power to correct all error patterns of weight ‘\( t \)’ or less if and only if \( d(\alpha, \delta) \geq 2t + 1 \) for all \( \alpha \) and \( \delta \)”.

However, since the smallest distance between any pair of code words is the minimum distance of the code, \( d_{\text{min}} \), ‘guarantees’ correcting all the error patterns of

\[ t \leq \left\lfloor \frac{1}{2} (d_{\text{min}} - 1) \right\rfloor \]  

(6.35)

where \( \left\lfloor \frac{1}{2} (d_{\text{min}} - 1) \right\rfloor \) denotes the largest integer no greater than the number \( \frac{1}{2} (d_{\text{min}} - 1) \). The parameter ‘\( t \)’ = \( \left\lfloor \frac{1}{2} (d_{\text{min}} - 1) \right\rfloor \) is called the “random-error-correcting capability” of the code and the code is referred to as a “\( t \)-error correcting code”. The \((6, 3)\) code of Example 6.1 has a minimum distance of 3 and from Eq. (6.35) it follows \( t = 1 \), which means it is a ‘Single Error Correcting’ (SEC) code. It is capable of correcting any error pattern of single errors over a block of six digits.

For an \((n, k)\) linear code, observe that, there are \( 2^{n-k} \) syndromes including the all zero syndrome. Each syndrome corresponds to a specific error pattern. If ‘\( j \)’ is the number of error
locations in the $n$-dimensional error pattern $e$, we find in general, there are $\binom{n}{j} = nC_j$ multiple error patterns. It then follows that the total number of all possible error patterns $= \sum_{j=0}^{t} \binom{n}{j}$, where $t$ is the maximum number of error locations in $e$. Thus we arrive at an important conclusion. “If an $(n, k)$ linear block code is to be capable of correcting up to $t$ errors, the total number of syndromes shall not be less than the total number of all possible error patterns”, i.e.

$$2^{n-k} \geq \sum_{j=0}^{t} \binom{n}{j}$$

(6.36)

Eq (6.36) is usually referred to as the “Hamming bound”. A binary code for which the Hamming Bound turns out to be equality is called a “Perfect code”.

6.7 Standard Array and Syndrome Decoding:

The decoding strategy we are going to discuss is based on an important property of the syndrome.

Suppose $v_j, j = 1, 2 \ldots 2^k$, be the $2^k$ distinct code vectors of an $(n, k)$ linear block code. Correspondingly let, for any error pattern $e$, the $2^k$ distinct error vectors, $e_j$ be defined by

$$e_j = e \oplus v_j, j = 1, 2 \ldots 2^k$$

(6.37)

The set of vectors $\{e_j, j = 1, 2 \ldots 2^k\}$ so defined is called the “co-set” of the code. That is, a ‘co-set’ contains exactly $2^k$ elements that differ at most by a code vector. It then follows that there are $2^{n-k}$ co-sets for an $(n, k)$ linear block code. Post multiplying Eq (6.37) by $H^T$, we find

$$e_j H^T = e H^T \oplus v_j H^T$$

$$= e H^T$$

(6.38)

Notice that the RHS of Eq (6.38) is independent of the index $j$, as for any code word the term $v_j H^T = 0$. From Eq (6.38) it is clear that “all error patterns that differ at most by a code word have the same syndrome”. That is, each co-set is characterized by a unique syndrome.

Since the received vector $r$ may be any of the $2^n$ $n$-tuples, no matter what the transmitted code word was, observe that we can use Eq (6.38) to partition the received code words into $2^k$ disjoint sets and try to identify the received vector. This will be done by preparing what is called the “standard array”. The steps involved are as below:

**Step 1**: Place the $2^k$ code vectors of the code in a row, with the all zero vector $v_1 = (0, 0, 0 \ldots 0) = O$ as the first (left most) element.

**Step 2**: From among the remaining $(2^n - 2^k)$ - $n$ – tuples, $e_2$ is chosen and placed below the all-zero vector, $v_1$. The second row can now be formed by placing $(e_2 \oplus v_j)$, $j = 2, 3 \ldots 2^k$ under $v_j$.

**Step 3**: Now take an un-used $n$-tuple $e_3$ and complete the $3^{rd}$ row as in step 2.

**Step 4**: continue the process until all the $n$-tuples are used.
The resultant array is shown in Fig. 6.12.

<table>
<thead>
<tr>
<th>$v_j$</th>
<th>$v_j \oplus e_2$</th>
<th>$v_j \oplus e_3$</th>
<th>……</th>
<th>$v_j \oplus e_{2k}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_2$</td>
<td>$v_2 \oplus e_2$</td>
<td>$v_3 \oplus e_2$</td>
<td>……</td>
<td>$v_{2k} \oplus e_2$</td>
</tr>
<tr>
<td>$e_3$</td>
<td>$v_2 \oplus e_3$</td>
<td>$v_3 \oplus e_3$</td>
<td>……</td>
<td>$v_{2k} \oplus e_3$</td>
</tr>
<tr>
<td>……</td>
<td>……</td>
<td>……</td>
<td>……</td>
<td>……</td>
</tr>
<tr>
<td>$e_{2n-k}$</td>
<td>$v_2 \oplus e_{2n-k}$</td>
<td>$v_3 \oplus e_{2n-k}$</td>
<td>……</td>
<td>$v_{2k} \oplus e_{2n-k}$</td>
</tr>
</tbody>
</table>

Fig 6.12 Standard Array for an $(n, k)$ linear Block code.

Since all the code vectors, $v_j$, are all distinct, the vectors in any row of the array are also distinct. For, if two $n$-tuples in the $l$-th row are identical, say $e_l \oplus v_j = e_l \oplus v_m, j \neq m$; we should have $v_j = v_m$ which is impossible. Thus it follows that “no two $n$-tuples in the same row of a standard array are identical”.

Next, let us consider that an $n$-tuple appears in both $l$-th row and the $m$-th row. Then for some $j_1$ and $j_2$ this implies $e_l \oplus v_{j_1} = e_m \oplus v_{j_2}$, which then implies $e_l = e_m \oplus (v_{j_2} \oplus v_{j_1})$; (remember that $X \oplus X = 0$ in modulo-2 arithmetic) or $e_l = e_m \oplus v_{j_3}$ for some $j_3$. Since by property of linear block codes $v_{j_3}$ is also a code word, this implies, by the construction rules given, that $e_l$ must appear in the $m$-th row, which is a contradiction of our steps, as the first element of the $m$-th row is $e_m$ and is an unused vector in the previous rows. This clearly demonstrates another important property of the array: “Every $n$-tuple appears in one and only one row”.

From the above discussions it is clear that there are $2^{n-k}$ disjoint rows or co-sets in the standard array and each row or co-set consists of $2^k$ distinct entries. The first $n$-tuple of each co-set, (i.e., the entry in the first column) is called the “Co-set leader”. Notice that any element of the co-set can be used as a co-set leader and this does not change the element of the co-set - it results simply in a permutation.

Suppose $D_j^T$ is the $j$-th column of the standard array. Then it follows

$$D_j = \{v_j, e_2 \oplus v_j, e_3 \oplus v_j, \ldots e_{2n-k} \oplus v_j\} \quad \text{--------------------------} (6.39)$$

where $v_j$ is a code vector and $e_2, e_3, \ldots e_{2n-k}$ are the co-set leaders.

The $2^k$ disjoints columns $D_1^T, D_2^T, \ldots D_{2^k}^T$ can now be used for decoding of the code. If $v_j$ is the transmitted code word over a noisy channel, it follows from Eq (6.39) that the received vector $r$ is in $D_j^T$ if the error pattern caused by the channel is a co-set leader. If this is the case $r$ will be decoded correctly as $v_j$. If not an erroneous decoding will result for, any error pattern $\hat{e}$ which is not a co-set leader must be in some co-set and under some nonzero code vector, say, in the $i$-th co-set and under $v \neq 0$. Then it follows

$$\hat{e} = e_i \oplus v_l \quad \text{and the received vector is } r$$

$$= v_j \oplus \hat{e} = v_j \oplus (e_i \oplus v_l) = e_i \oplus v_m$$
Thus the received vector is in $D_m^T$ and it will be decoded as $v_m$ and a decoding error has been committed. Hence it is explicitly clear that “Correct decoding is possible if and only if the error pattern caused by the channel is a co-set leader”. Accordingly, the $2^{n-k}$ co-set leaders (including the all zero vector) are called the “Correctable error patterns”, and it follows “Every $(n, k)$ linear block code is capable of correcting $2^{n-k}$ error patterns”.

So, from the above discussion, it follows that in order to minimize the probability of a decoding error, “The most likely to occur” error patterns should be chosen as co-set leaders. For a BSC an error pattern of smallest weight is more probable than that of a larger weight. Accordingly, when forming a standard array, error patterns of smallest weight should be chosen as co-set leaders. Then the decoding based on the standard array would be the ‘minimum distance decoding’ (the maximum likelihood decoding). This can be demonstrated as below.

Suppose a received vector $r$ is found in the $j^{th}$ column and $l^{th}$ row of the array. Then $r$ will be decoded as $v_j$. We have

$$d(r, v_j) = \omega(r \oplus v_j) = \omega(e_l \oplus v_j \oplus v_j) = \omega(e_l)$$

where we have assumed $v_j$ indeed is the transmitted code word. Let $v_s$ be any other code word, other than $v_j$. Then

$$d(r, v_s) = \omega(r \oplus v_s) = \omega(e_l \oplus v_j \oplus v_s) = \omega(e_l \oplus v_j)$$

as $v_j$ and $v_s$ are code words, $v_i = v_j \oplus v_s$ is also a code word of the code. Since $e_l$ and $(e_l \oplus v_i)$ are in the same co set and, that $e_l$ has been chosen as the co-set leader and has the smallest weight it follows $\omega(e_l) \leq \omega(e_l \oplus v_i)$ and hence $d(r, v_j) \leq d(r, v_s)$. Thus the received vector is decoded into a closet code vector. Hence, if each co-set leader is chosen to have minimum weight in its co-set, the standard array decoding results in the minimum distance decoding or maximum likely hood decoding.

Suppose “$a_0, a_1, a_2, \ldots, a_n$” denote the number of co-set leaders with weights 0, 1, 2… n. This set of numbers is called the “Weight distribution” of the co-set leaders. Since a decoding error will occur if and only if the error pattern is not a co-set leader, the probability of a decoding error for a BSC with error probability (transition probability) $p$ is given by

$$P(E) = 1 - \sum_{j=0}^{n} a_j p^j (1 - p)^{n-j} \quad \text{................................. (6.40)}$$

Example 6.8:

For the (6, 3) linear block code of Example 6.1 the standard array, along with the syndrome table, is as below:
The weight distribution of the co-set leaders in the array shown are \( a_0 = 1, a_1 = 6, a_2 = 1, a_3 = a_4 = a_5 = a_6 = 0 \). From Eq (6.40) it then follows:

\[
P(E) = 1 - [(1-p)^6 +6p(1-p)^5 + p^2 (1-p)^4]
\]

With \( p = 10^{-2} \), we have \( P(E) = 1.3643879 \times 10^{-3} \)

A received vector \((010 001)\) will be decoded as \((010101)\) and a received vector \((100 110)\) will be decoded as \((110 110)\).

We have seen in Eq. (6.38) that each co-set is characterized by a unique syndrome or there is a one-to-one correspondence between a co-set leader (a correctable error pattern) and a syndrome. These relationships, then, can be used in preparing a decoding table that is made up of \(2^{n-k}\) co-set leaders and their corresponding syndromes. This table is either stored or wired in the receiver. The following are the steps in decoding:

**Step 1**: Compute the syndrome \( s = rH^T \)

**Step 2**: Locate the co-set leader \( e_j \) whose syndrome is \( s \). Then \( e_j \) is assumed to be the error pattern caused by the channel.

**Step 3**: Decode the received vector \( r \) into the code vector \( v = r \oplus e_j \)

This decoding scheme is called the “Syndrome decoding” or the “Table look up decoding”.

Observe that this decoding scheme is applicable to any linear \((n, k)\) code, i.e., it need not necessarily be a systematic code.

**Comments**:

1) Notice that for all correctable single error patterns the syndrome will be identical to a column of the \( H \) matrix and indicates that the received vector is in error corresponding to that column position.
For Example, if the received vector is \((010001)\), then the syndrome is \((100)\). This is identical with the \(4^{th}\) column of the \(H\)-matrix and hence the \(4^{th}\) position of the received vector is in error. Hence the corrected vector is \(010101\). Similarly, for a received vector \((100110)\), the syndrome is \(101\) and this is identical with the second column of the \(H\)-matrix. Thus the second position of the received vector is in error and the corrected vector is \((110110)\).

2) A table can be prepared relating the error locations and the syndrome. By suitable combinatorial circuits data recovery can be achieved. For the \((6, 3)\) systematic linear code we have the following table for \(r = (r_1 r_2 r_3 r_4 r_5 r_6)\).

<table>
<thead>
<tr>
<th>Error location</th>
<th>Error in digits</th>
<th>Syndrome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(r_1)</td>
<td>0 1 1</td>
</tr>
<tr>
<td>2</td>
<td>(r_2)</td>
<td>1 0 1</td>
</tr>
<tr>
<td>3</td>
<td>(r_3)</td>
<td>1 1 0</td>
</tr>
<tr>
<td>4</td>
<td>(r_4)</td>
<td>1 0 0</td>
</tr>
<tr>
<td>5</td>
<td>(r_5)</td>
<td>0 1 0</td>
</tr>
<tr>
<td>6</td>
<td>(r_6)</td>
<td>0 0 1</td>
</tr>
<tr>
<td>No error</td>
<td></td>
<td>0 0 0</td>
</tr>
</tbody>
</table>

6.8 Hamming Codes:

Hamming code is the first class of linear block codes devised for error correction. The single error correcting (SEC) Hamming codes are characterized by the following parameters.

Code length: \(n = (2^m - 1)\)

Number of Information symbols: \(k = (2^m - m - 1)\)

Number of parity check symbols: \((n - k) = m\)

Error correcting capability: \(t = 1\), \((d_{min} = 3)\)

The parity check matrix \(H\) of this code consists of all the non-zero \(m\)-tuples as its columns. In systematic form, the columns of \(H\) are arranged as follows

\[ H = [Q : I_m] \]

Where \(I_m\) is an identity (unit) matrix of order \(m \times m\) and \(Q\) matrix consists of \((2^m - m - 1)\) columns which are the \(m\)-tuples of weight 2 or more. As an illustration for \(k=4\) we have from \(k = 2^m - m - 1\).

\[ m=1 \quad k=0, m=2 \quad k=1, m=3 \quad k=4 \]

Thus we require 3 parity check symbols and the length of the code \(2^3 - 1 = 7\). This results in the \((7, 4)\) Hamming code.
The parity check matrix for the \((7, 4)\) linear systematic Hamming code is then

\[
H = \begin{bmatrix}
1110 & 000 \\
1101 & 010 \\
1011 & 001
\end{bmatrix}
\]

The generator matrix of the code can be written in the form

\[
G = \begin{bmatrix}
I_{2^m-m-1} & Q^T
\end{bmatrix}
\]

And for the \((7, 4)\) systematic code it follows:

\[
G = \begin{bmatrix}
1000 & 111 \\
0100 & 110 \\
0010 & 101 \\
0001 & 011
\end{bmatrix}
\]

\(p_1, p_2, m_1, p_3, m_2, m_3, p_4, m_5, m_6, m_7, m_8, m_9, m_{10}, m_{11}, p_5, m_{12}\)

Where \(p_1, p_2, p_3\)… are the parity digits and \(m_1, m_2, m_3\)… are the message digits. For example, let us consider the non systematic \((7, 4)\) Hamming code.

\(p_1 = 1, 3, 5, 7, 9, 11, 13, 15…\)

\(p_2 = 2, 3, 6, 7, 10, 11, 14, 15…\)

\(p_3 = 4, 5, 6, 7, 12, 13, 14, 15…\)

It can be verified that \((7, 4), (15, 11), (31, 26), (63, 57)\) are all single error correcting Hamming codes and are regarded quite useful.

An important property of the Hamming codes is that they satisfy the condition of Eq. (6.36) with equality sign, assuming that \(t=1\). This means that Hamming codes are “single error correcting binary perfect codes”. This can also be verified from Eq. (6.35)

We may delete any \(l\) columns from the parity check matrix \(H\) of the Hamming code resulting in the reduction of the dimension of \(H\) matrix to \(m \times (2^m-l-1)\). Using this new matrix as the parity check matrix we obtain a “shortened” Hamming code with the following parameters.

Code length: \(n = 2^m-l-1\)

Number of Information symbols: \(k=2^m-m-l-1\)

Number of parity check symbols: \(n-k = m\)
Minimum distance: \( d_{\text{min}} \geq 3 \)

Notice that if the deletion of the columns of the H matrix is proper, we may obtain a Hamming code with \( d_{\text{min}} = 4 \). For example if we delete from the sub-matrix \( Q \) all the columns of even weight, we obtain an \( m \times 2^{m-1} \) matrix

\[
\overline{H} = [\overline{Q} : I_m]
\]

Where \( \overline{Q} \) contains \( (2^{m-1} - m) \) columns of odd weight. Clearly no three columns add to zero as all columns have odd weight. However, for a column in \( \overline{Q} \), there exist three columns in \( I_m \) such that four columns add to zero. Thus the shortened Hamming codes with \( \overline{H} \) as the parity check matrix has minimum distance exactly 4.

The distance – 4 shortened Hamming codes can be used for correcting all single error patterns while simultaneously detecting all double error patterns. Notice that when single errors occur the syndromes contain odd number of one’s and for double errors it contains even number of ones. Accordingly the decoding can be accomplished in the following manner.

1. If \( s = 0 \), no error occurred.
2. If \( s \) contains odd number of ones, single error has occurred. The single error pattern pertaining to this syndrome is added to the received code vector for error correction.
3. If \( s \) contains even number of one’s an uncorrectable error pattern has been detected.

Alternatively the \( \text{SEC} \) Hamming codes may be made to detect double errors by adding an extra parity check in its \((n+1)^{\text{th}}\) position. Thus \((8, 4), (6, 11)\) etc. codes have \( d_{\text{min}} = 4 \) and correct single errors with detection of double errors.

**CYCLIC CODES**

“Binary cyclic codes” form a sub class of linear block codes. Majority of important linear block codes that are known to-date are either cyclic codes or closely related to cyclic codes. Cyclic codes are attractive for two reasons: First, encoding and syndrome calculations can be easily implemented using simple shift registers with feed back connections. Second, they posses well defined mathematical structure that permits the design of higher-order error correcting codes.

A binary code is said to be "cyclic" if it satisfies:

1. **Linearity property** – sum of two code words is also a code word.
2. **Cyclic property** – Any lateral shift of a code word is also a code word.

The second property can be easily understood from Fig. 7.1. Instead of writing the code as a row vector, we have represented it along a circle. The direction of traverse may be either clockwise or counter clockwise (right shift or left shift).
For example, if we move in a counter clockwise direction then starting at ‘A’ the code word is 110001100 while if we start at B it would be 011001100. Clearly, the two code words are related in that one is obtained from the other by a cyclic shift.

If the \( n \)-tuple, read from ‘A’ in the CW direction in Fig 7.1,

\[
v = (v_0, v_1, v_2, v_3, v_{n-2}, v_{n-1})
\]

is a code vector, then the code vector, read from B, in the CW direction, obtained by a one bit cyclic right shift:

\[
v^{(1)} = (v_{n-1}, v_0, v_1, v_2, \ldots, v_{n-3}, v_{n-2})
\]

is also a code vector. In this way, the \( n \)-tuples obtained by successive cyclic right shifts:

\[
v^{(2)} = (v_{n-2}, v_{n-1}, v_n, v_0, v_1, \ldots, v_{n-3})
\]

\[
v^{(3)} = (v_{n-3}, v_{n-2}, v_{n-1}, v_n, \ldots, v_0, v_1, v_{n-4})
\]

\[\vdots\]

\[
v^{(i)} = (v_{n-i}, v_{n-i+1}, \ldots, v_{n-1}, v_0, v_1, \ldots, v_{n-i-1})
\]

are all code vectors. This property of cyclic codes enables us to treat the elements of each code vector as the co-efficients of a polynomial of degree \((n-1)\).

This is the property that is extremely useful in the analysis and implementation of these codes. Thus we write the 'code polynomial' \( V(X) \) for the code in Eq (7.1) as a vector polynomial as:

\[
V(X) = v_0 + v_1 X + v_2 X^2 + v_3 X^3 + \ldots + v_{i-1} X^{i-1} + \ldots + v_{n-3} X^{n-3} + v_{n-2} X^{n-2} + v_{n-1} X^{n-1}
\]

Notice that the co-efficients of the polynomial are either '0' or '1' (binary codes), i.e. they belong to \( GF(2) \) as discussed in sec 6.7.1.

. Each power of \( X \) in \( V(X) \) represents a one bit cyclic shift in time.

. Therefore multiplication of \( V(X) \) by \( X \) maybe viewed as a cyclic shift or rotation to the right subject
to the condition \( X^n = 1 \). This condition (i) restores \( XV(X) \) to the degree \((n-1)\) (ii) Implies that right most bit is fed-back at the left.

This special form of multiplication is called "Multiplication modulo \( X^n + 1 \)"

Thus for a single shift, we have

\[
XV(X) = v_0X + v_1 X^2 + v_2 X^3 + \ldots + v_{n-2} X^{n-1} + v_{n-1} X^n
\]

\[
= v_{n-1} + v_0 X + v_1 X^2 + \ldots + v_{n-2} X^{n-1} + v_{n-1}(X^n + 1)
\]

\[
= V^{(1)}(X) = \text{Remainder obtained by dividing } XV(X) \text{ by } X^n + 1
\]

(Remember: \( X \text{ mod } Y \) means remainder obtained after dividing \( X \) by \( Y \))

Thus it turns out that

\[
V^{(1)}(X) = v_{n-1} + v_0 X + v_1 X^2 + \ldots + v_{n-2} X^{n-1}
\]

(7.5)

is the code polynomial for \( v^{(1)} \). We can continue in this way to arrive at a general format:

\[
X^i V(X) = V^{(i)}(X) + q(X)(X^n + 1)
\]

\[
\uparrow \quad \uparrow
\]

Where

\[
V^{(i)}(X) = v_{n-i} + v_{n-i+1}X + v_{n-i+2}X^2 + \ldots + v_{n-1}X^{i} + \ldots + v_{0}X^{i-1} + v_{1}X^{i+1} + \ldots + v_{n-2}X^{n-2} + v_{n-1}X^{n-1}
\]

(7.7)

### 7.1 Generator Polynomial for Cyclic Codes:

An \((n, k)\) cyclic code is specified by the complete set of code polynomials of degree \((\leq n-1)\) and contains a polynomial \( g(X) \), of degree \((n-k)\) as a factor, called the "generator polynomial" of the code. This polynomial is equivalent to the generator matrix \( G \), of block codes. Further, it is the only polynomial of minimum degree and is unique. Thus we have an important theorem

**Theorem 7.1** "If \( g(X) \) is a polynomial of degree \((n-k)\) and is a factor of \((X^n + 1)\) then \( g(X) \) generates an \((n, k)\) cyclic code in which the code polynomial \( V(X) \) for a data vector \( u = (u_0, u_1, \ldots, u_{k-1}) \) is generated by

\[
V(X) = U(X) \times g(X)
\]

Where

\[
U(X) = u_0 + u_1 X + u_2 X^2 + \ldots + u_{k-1} X^{k-1}
\]

is the data polynomial of degree \((k-1)\).

The theorem can be justified by Contradiction: - If there is another polynomial of same degree, then add the two polynomials to get a polynomial of degree \(< (n, k)\) (use linearity property and binary arithmetic). Not possible because minimum degree is \((n-k)\). Hence \( g(X) \) is unique

Clearly, there are \( 2^k \) code polynomials corresponding to \( 2^k \) data vectors. The code vectors
corresponding to these code polynomials form a linear \((n, k)\) code. We have then, from the theorem

\[
g(X) = 1 + \sum_{i=1}^{n-k-1} g_i X^i + X^{n-k} \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (7.10)
\]

As \(g(X) = g_0 + g_1 X + g_2 X^2 + \ldots + g_{n-k-1} X^{n-k-1} + g_{n-k} X^{n-k} \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (7.11)\)

is a polynomial of minimum degree, it follows that \(g_0 = g_{n-k} = 1\) always and the remaining coefficients may be either '0' or '1'. Performing the multiplication said in Eq (7.8) we have:

\[
U(X) g(X) = u_0 g(X) + u_1 X g(X) + \ldots + u_{k-1} X^{k-1} g(X) \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (7.12)
\]

Suppose \(u_0=1\) and \(u_1=u_2= \ldots =u_{k-1}=0\). Then from Eq (7.8) it follows \(g(X)\) is a code word polynomial of degree \((n-k)\). This is treated as a ‘basis code polynomial’ (All rows of the \(G\) matrix of a block code, being linearly independent, are also valid code vectors and form ‘basis vectors’ of the code). Therefore from cyclic property \(X^i g(X)\) is also a code polynomial. Moreover, from the linearity property - a linear combination of code polynomials is also a code polynomial. It follows therefore that any multiple of \(g(X)\) as shown in Eq (7.12) is a code polynomial. Conversely, any binary polynomial of degree \(\leq (n-1)\) is a code polynomial if and only if it is a multiple of \(g(X)\). The code words generated using Eq (7.8) are in non-systematic form. Non-systematic cyclic codes can be generated by simple binary multiplication circuits using shift registers.

In this book we have described cyclic codes with right shift operation. Left shift version can be obtained by simply re-writing the polynomials. Thus, for left shift operations, the various polynomials take the following form

\[
U(X) = u_0 X^{k-1} + u_1 X^{k-2} + \ldots + u_{k-2} X + u_{k-1} \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (7.13a)
\]

\[
V(X) = v_0 X^{n-1} + v_1 X^{n-2} + \ldots + v_{n-2} X + v_{n-1} \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (7.13b)
\]

\[
g(X) = g_0 X^{n-k} + g_1 X^{n-k-1} + \ldots + g_{n-k-1} X + g_{n-k} \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (7.13c)
\]

\[
=X_{n-k} + \sum_{i=1}^{n-k} g_i X^{n-k-i} + g_{n-k} \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (7.13d)
\]

Other manipulation and implementation procedures remain unaltered.

7.2 Multiplication Circuits:

Construction of encoders and decoders for linear block codes are usually constructed with combinational logic circuits with mod-2 adders. Multiplication of two polynomials \(A(X)\) and \(B(X)\) and the division of one by the other are realized by using sequential logic circuits, mod-2 adders and shift registers. In this section we shall consider multiplication circuits.

As a convention, the higher-order coefficients of a polynomial are transmitted first. This is the reason for the format of polynomials used in this book.

For the polynomial: \(A(X) = a_0 + a_1 X + a_2 X^2 + \ldots + a_{n-1} X^{n-1} \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (7.14)\)

where \(a_i\)'s are either a '0' or a '1', the right most bit in the sequence \((a_0, a_1, a_2 \ldots a_{n-1})\) is transmitted.
first in any operation. The product of the two polynomials \( A(X) \) and \( B(X) \) yield:

\[
C(X) = A(X) \times B(X)
\]

\[
= (a_0 + a_1 X + a_2 X^2 + \ldots + a_{n-1} X^{n-1}) (b_0 + b_1 X + b_2 X^2 + \ldots + b_{m-1} X^{m-1})
\]

\[
= a_0 b_0 + (a_1 b_0 + a_0 b_1) X + (a_0 b_2 + a_0 b_2 + a_1 b_1) X^2 + \ldots + (a_{n-2} b_{m-1} + a_{n-1} b_{m-2}) X^{n+m-3} + a_{n-1} b_{m-1} X^{n+m-2}
\]

This product may be realized with the circuits of Fig 7.2 (a) or (b), where \( A(X) \) is the input and the co-efficient of \( B(X) \) are given as weighting factor connections to the mod-2 adders. A '0' indicates no connection while a '1' indicates a connection. Since higher order co-efficients are first sent, the highest order co-efficient \( a_{n-1} \times b_{m-1} \) of the product polynomial is obtained first at the output of Fig 7.2(a). Then the co-efficient of \( X^{n+m-3} \) is obtained as the sum of \( \{a_{n-2} b_{m-1} + a_{n-1} b_{m-2}\} \), the first term directly and the second term through the shift register \( SR1 \). Lower order co-efficients are then generated through the successive \( SR \)'s and mod-2 adders. After \( (n + m - 2) \) shifts, the \( SR \)'s contain \( \{0, 0 \ldots 0, a_0, a_1\} \) and the output is \( (a_0 b_1 + a_1 b_0) \) which is the co-efficient of \( X \). After \( (n + m - 1) \) shifts, the \( SR \)'s contain \( \{0, 0, 0, \ldots, a_0\} \) and the output is \( a_0 \times b_0 \). The product is now complete and the contents of the \( SR \)'s become \( \{0, 0, 0 \ldots, 0, 0\} \). Fig 7.2(b) performs the multiplication in a similar way but the arrangement of the \( SR \)'s and ordering of the co-efficients are different (reverse order!). This modification helps to combine two multiplication operations into one as shown in Fig 7.2(c).

From the above description, it is clear that a non-systematic cyclic code may be generated using \( (n-k) \) shift registers. Following examples illustrate the concepts described so far.

![Circuit Diagrams](image)

**Fig 7.2 Multiplication circuits**

**Example 7.1:** Consider that a polynomial \( A(X) \) is to be multiplied by

\[
B(X) = 1 + X + X^3 + X^4 + X^6
\]

The circuits of Fig 7.3 (a) and (b) give the product \( C(X) = A(X) \cdot B(X) \).
Example 7.2: Consider the generation of a (7, 4) cyclic code. Here \( n-k = 7-4 = 3 \) and we have to find a generator polynomial of degree 3 which is a factor of \( X^n + 1 = X^7 + 1 \).

To find the factors of degree 3, divide \( X^7 + 1 \) by \( X^3 + aX^2 + bX + 1 \), where 'a' and 'b' are binary numbers, to get the remainder as \( abX^2 + (1+a+b)X + (a+b+ab+1) \). Only condition for the remainder to be zero is \( a + b = 1 \) which means either \( a = 1, b = 0 \) or \( a = 0, b = 1 \). Thus we have two possible polynomials of degree 3, namely

\[
g_1(X) = X^3 + X^2 + 1 \quad \text{and} \quad g_2(X) = X^3 + X + 1
\]

In fact, \( X^7 + 1 \) can be factored as:

\[
(X^7+1) = (X+1)(X^3 + X^2 + 1)(X^3 + X + 1)
\]

Thus selection of a 'good' generator polynomial seems to be a major problem in the design of cyclic codes. No clear-cut procedures are available. Usually computer search procedures are followed.

Let us choose \( g(X) = X^3 + X + 1 \) as the generator polynomial. The encoding circuits are shown in Fig 7.4(a) and (b).

To understand the operation, Let us consider \( u = (10 1 1) \) i.e.

\[
U(X) = 1 + X^2 + X^3.
\]
We have 
\[ V(X) = (1 + X^2 + X^3)(1 + X + X^3) = 1 + X^2 + X^3 + X + X^3 + X^4 + X^5 + X^6 \]

because \( X^3 + X^3 = 0 \)

\[ \Rightarrow v = (1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1) \]

The multiplication operation, performed by the circuit of Fig 7.4(a), is listed in the Table below step by step. In shift number 4, ‘00’ is introduced to flush the registers. As seen from the tabulation the product polynomial is:

\[ V(X) = 1 + X + X^2 + X^3 + X^4 + X^5 + X^6, \]

and hence output code vector is \( v = (1 \ 1 \ 1 \ 1 \ 1 \ 1) \), as obtained by direct multiplication. The reader can verify the operation of the circuit in Fig 7.4(b) in the same manner. Thus the multiplication circuits of Fig 7.4 can be used for generation of non-systematic cyclic codes.

<table>
<thead>
<tr>
<th>Shift Number</th>
<th>Input Queue</th>
<th>Bit shifted IN</th>
<th>Contents of shift registers.</th>
<th>Output</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0001011</td>
<td>-</td>
<td>0 0 0</td>
<td>-</td>
<td>Circuit In reset mode</td>
</tr>
<tr>
<td>1</td>
<td>000101</td>
<td>1</td>
<td>1 0 0 1</td>
<td>1</td>
<td>Co-efficient of ( X^6 )</td>
</tr>
<tr>
<td>2</td>
<td>00010</td>
<td>1</td>
<td>1 1 1 0</td>
<td>1</td>
<td>Co-efficient of ( X^5 )</td>
</tr>
<tr>
<td>3</td>
<td>0001</td>
<td>0</td>
<td>0 1 1 1</td>
<td>X^4 co-efficient</td>
<td></td>
</tr>
<tr>
<td>*4</td>
<td>000</td>
<td>1</td>
<td>1 0 1 1</td>
<td>X^3 co-efficient</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>00</td>
<td>0</td>
<td>0 1 0 1</td>
<td>X^2 co-efficient</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>00</td>
<td>0</td>
<td>0 0 1 1</td>
<td>X^1 co-efficient</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>0</td>
<td>0 0 0 1</td>
<td>X^0 co-efficient</td>
<td></td>
</tr>
</tbody>
</table>

7.3 Dividing Circuits:

As in the case of multipliers, the division of \( A(X) \) by \( B(X) \) can be accomplished by using shift registers and Mod-2 adders, as shown in Fig 7.5. In a division circuit, the first co-efficient of the quotient is \( a_{n-1} : (b_{m-1}) = q_1 \), and \( q_1.B(X) \) is subtracted from \( A(X) \). This subtraction is carried out by the feedback connections shown. This process will continue for the second and subsequent terms. However, remember that these coefficients are binary coefficients. After \( (n-1) \) shifts, the entire quotient will appear at the output and the remainder is stored in the shift registers.
It is possible to combine a divider circuit with a multiplier circuit to build a “composite multiplier-divider circuit” which is useful in various encoding circuits. An arrangement to accomplish this is shown in Fig 7.6(a) and an illustration is shown in Fig 7.6(b).

We shall understand the operation of one divider circuit through an example. Operation of other circuits can be understood in a similar manner.

**Example 7.3:**

Let \( A(X) = X^3 + X^5 + X^6 \), \( \rightarrow A = (0001011) \), \( B(X) = 1 + X + X^3 \). We want to find the quotient and remainder after dividing \( A(X) \) by \( B(X) \). The circuit to perform this division is shown in Fig 7.7, drawn using the format of Fig 7.5(a). The operation of the divider circuit is listed in the table:
Table Showing the Sequence of Operations of the Dividing circuit

<table>
<thead>
<tr>
<th>Shift Number</th>
<th>Input Queue</th>
<th>Bit shifted IN</th>
<th>Contents of shift Registers.</th>
<th>Output</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0001011</td>
<td>-</td>
<td>SRI</td>
<td>SR2</td>
<td>SR3</td>
</tr>
<tr>
<td>1</td>
<td>000101</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>00010</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>*000</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The quotient co-efficients will be available only after the fourth shift as the first three shifts result in entering the first 3-bits to the shift registers and in each shift output of the last register, \(SR3\), is zero.

The quotient co-efficient serially presented at the output are seen to be \((1111)\) and hence the quotient polynomial is \(Q(X) = 1 + X + X^2 + X^3\). The remainder co-efficients are \((100)\) and the remainder polynomial is \(R(X) = 1\). The polynomial division steps are listed in the next page.

**Division Table for Example 7.3:**

![Fig 7.7 Circuit for dividing \(A(X)\) by \((1 + X + X^3)\)]
Let us assume a systematic format for the cyclic code as below:

\[ v = (p_0, p_1, p_2 \ldots p_{n-k-1}, u_0, u_1, u_2 \ldots u_{k-1}) \] \hspace{1cm} (7.15)

The code polynomial in the assumed systematic format becomes:

\[ V(X) = p_0 + p_1 X + p_2 X^2 + \ldots + p_{n-k-1} X^{n-k-1} + u_0 X^{n-k} + u_1 X^{n-k+1} + \ldots + u_{k-1} X^{n-1} \] \hspace{1cm} (7.16)

\[ = P(X) + X^{n-k} U(X) \] \hspace{1cm} (7.17)

Since the code polynomial is a multiple of the generator polynomial we can write:

\[ V(X) = P(X) + X^{n-k} U(X) = Q(X) g(X) \] \hspace{1cm} (7.18)

\[ \Rightarrow \frac{X^{n-k} U(X)}{g(X)} = Q(X) + \frac{P(X)}{g(X)} \] \hspace{1cm} (7.19)

Thus division of \( X^{n-k} U(X) \) by \( g(X) \) gives us the quotient polynomial \( Q(X) \) and the remainder polynomial \( P(X) \). Therefore to obtain the cyclic codes in the systematic form, we determine the remainder polynomial \( P(X) \) after dividing \( X^{n-k} U(X) \) by \( g(X) \). This division process can be easily achieved by noting that "multiplication by \( X^{n-k} \) amounts to shifting the sequence by \( (n-k) \) bits". Specifically in the circuit of Fig 7.5 (a), if the input \( A(X) \) is applied to the Mod-2 adder after the \( (n-k) \)th shift register the result is the division of \( X^{n-k} A(X) \) by \( B(X) \).

Accordingly, we have the following scheme to generate systematic cyclic codes. The generator polynomial is written as:
The circuit of Fig 7.8 does the job of dividing $X^{n-k} U (X)$ by $g(X)$. The following steps describe the encoding operation.

1. The switch $S$ is in position 1 to allow transmission of the message bits directly to an output shift register during the first $k$-shifts.
2. At the same time the 'GATE' is 'ON' to allow transmission of the message bits into the $(n-k)$ stage encoding shift register.
3. After transmission of the $k^{th}$ message bit the GATE is turned OFF and the switch $S$ is moved to position 2.
4. $(n-k)$ zeroes introduced at "A" after step 3, clear the encoding register by moving the parity bits to the output register.
5. The total number of shifts is equal to $n$ and the contents of the output register is the code word polynomial $V (X) = P (X) + X^{n-k} U (X)$.
6. After step-4, the encoder is ready to take up encoding of the next message input.

Clearly, the encoder is very much simpler than the encoder of an $(n, k)$ linear block code and the memory requirements are reduced. The following example illustrates the procedure.

Example 7.4:

Let $u = (1 \ 0 \ 1 \ 1)$ and we want a $(7, 4)$ cyclic code in the systematic form. The generator polynomial chosen is $g (X) = 1 + X + X^3$

For the given message, $U (X) = 1 + X^2 + X^3$

$X^{n-k} U (X) = X^3 U (X) = X^3 + X^5 + X^6$

We perform direct division $X^{n-k} U (X)$ by $g (X)$ as shown below. From direct division observe that $p_0=1, p_1=p_2=0$. Hence the code word in systematic format is:

$v = (p_0, p_1, p_2; u_0, u_1, u_2, u_3) = (1, 0, 1, 0, 1, 1)$
The encoder circuit for the problem on hand is shown in Fig 7.9. The operational steps are as follows:

<table>
<thead>
<tr>
<th>Shift Number</th>
<th>Input Queue</th>
<th>Bit shifted IN</th>
<th>Register contents</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1011</td>
<td>-</td>
<td>000</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>0</td>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>1</td>
<td>101</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>1</td>
<td>100</td>
<td>1</td>
</tr>
</tbody>
</table>

After the Fourth shift \( GATE \) Turned \( OFF \), switch \( S \) moved to position 2, and the parity bits contained in the register are shifted to the output. The output code vector is \( v = (100 \, 1011) \) which agrees with the direct hand calculation.

### 7.6 Syndrome Calculation - Error Detection and Error Correction:

Suppose the code vector \( v=(v_0, v_1, v_2 \ldots v_{n-1}) \) is transmitted over a noisy channel. Hence the received vector may be a corrupted version of the transmitted code vector. Let the received code vector be \( r=(r_0, r_1, r_2 \ldots r_{n-1}) \). The received vector may not be anyone of the \( 2^k \) valid code vectors. The function of the decoder is to determine the transmitted code vector based on the received vector.

The decoder, as in the case of linear block codes, first computes the syndrome to check whether or not the received code vector is a valid code vector. In the case of cyclic codes, if the syndrome is zero, then the received code word polynomial must be divisible by the generator polynomial. If the syndrome is non-zero, the received word contains transmission errors and needs error correction. Let the received code vector be represented by the polynomial.
\[ R(X) = r_0 + r_1X + r_2X^2 + \ldots + r_{n-1}X^{n-1} \]

Let \( A(X) \) be the quotient and \( S(X) \) be the remainder polynomials resulting from the division of \( R(X) \) by \( g(X) \) i.e.

\[ \frac{R(X)}{g(X)} = A(X) + \frac{S(X)}{g(X)} \quad \text{.......................... (7.21)} \]

The remainder \( S(X) \) is a polynomial of degree \( (n-k-1) \) or less. It is called the "Syndrome polynomial". If \( E(X) \) is the polynomial representing the error pattern caused by the channel, then we have:

\[ R(X) = V(X) + E(X) \quad \text{.......................... (7.22)} \]

And it follows as \( V(X) = U(X) g(X) \), that:

\[ E(X) = [A(X) + U(X)] g(X) + S(X) \quad \text{.................. (7.23)} \]

That is, the syndrome of \( R(X) \) is equal to the remainder resulting from dividing the error pattern by the generator polynomial; and the syndrome contains information about the error pattern, which can be used for error correction. Hence syndrome calculation can be accomplished using divider circuits discussed in Sec 7.4, Fig7.5. A "Syndrome calculator" is shown in Fig 7.10.

![Syndrome calculator using (n-k) Shift registers](image)

The syndrome calculations are carried out as below:

1. The register is first initialized. With \( GATE 2 \) -ON and \( GATE1 \) -OFF, the received vector is entered into the register.

2. After the entire received vector is shifted into the register, the contents of the register will be the syndrome, which can be shifted out of the register by turning \( GATE1 \) ON and \( GATE2 \) OFF. The circuit is ready for processing next received vector.

Cyclic codes are extremely well suited for 'error detection'. They can be designed to detect many combinations of likely errors and implementation of error-detecting and error correcting circuits is practical and simple. Error detection can be achieved by employing (or adding) an additional R-S flip-flop to the syndrome calculator. If the syndrome is nonzero, the flip-flop sets and provides an indication of error. Because of the ease of implementation, virtually all error detecting codes are invariably 'cyclic codes'. If we are interested in error correction, then the decoder must be capable of determining the error pattern \( E(X) \) from the syndrome \( S(X) \) and add it to \( R(X) \) to determine the transmitted \( V(X) \). The following scheme shown in Fig 7.11 may be employed for the purpose. The error correction procedure consists of the following steps:

**Step1.** Received data is shifted into the buffer register and syndrome registers with switches \( S_{IN} \) closed and \( S_{OUT} \) open and error correction is performed with \( S_{IN} \) open and \( S_{OUT} \) closed.

**Step2.** After the syndrome for the received code word is calculated and placed in the
syndrome register, the contents are read into the error detector. The detector is a combinatorial circuit designed to output a '1' if and only if the syndrome corresponds to a correctable error pattern with an error at the highest order position $X^{n-1}$. That is, if the detector output is a '1' then the received digit at the right most stage of the buffer register is assumed to be in error and will be corrected. If the detector output is '0' then the received digit at the right most stage of the buffer is assumed to be correct. Thus the detector output is the estimate error value for the digit coming out of the buffer register.

![Fig 7.11 General Decoder for Cyclic codes](image)

**Step3.** In the third step, the first received digit in the syndrome register is shifted right once. If the first received digit is in error, the detector output will be '1' which is used for error correction. The output of the detector is also fed to the syndrome register to modify the syndrome. This results in a new syndrome corresponding to the ‘altered’ ‘received code word shifted to the right by one place.

**Step4.** The new syndrome is now used to check and correct the second received digit, which is now at the right most position, is an erroneous digit. If so, it is corrected, a new syndrome is calculated as in step-3 and the procedure is repeated.

**Step5.** The decoder operates on the received data digit by digit until the entire received code word is shifted out of the buffer.

At the end of the decoding operation, that is, after the received code word is shifted out of the buffer, all those errors corresponding to correctable error patterns will have been corrected, and the syndrome register will contain all zeros. If the syndrome register does not contain all zeros, this means that an un-correctable error pattern has been detected. The decoding schemes described in Fig 7.10 and Fig 7.11 can be used for any cyclic code. However, the practicality depends on the complexity of the combinational logic circuits of the error detector. In fact, there are special classes of cyclic codes for which the decoder can be realized by simpler circuits. However, the price paid for such simplicity is in the reduction of code efficiency for a given block size.

**7.7 Bose- Chaudhury - Hocquenghem (BCH) Codes:**
One of the major considerations in the design of optimum codes is to make the block size $n$ smallest for a given size $k$ of the message block so as to obtain a desirable value of $d_{\text{min}}$. Or for given code length $n$ and efficiency $k/n$, one may wish to design codes with largest $d_{\text{min}}$. That means we are on the look out for the codes that have 'best error correcting capabilities". The BCH codes, as a class, are one of the most important and powerful error-correcting cyclic codes known. The most common BCH codes are characterized as follows. Specifically, for any positive integer $m \geq 3$, and $t < 2^m - 1)/2$, there exists a binary BCH code (called 'primitive' BCH code) with the following parameters:

**Block length** : $n = 2^m - 1$
**Number of message bits** : $k \leq n - mt$
**Minimum distance** : $d_{\text{min}} \geq 2t + 1$

Clearly, BCH codes are "$t$ - error correcting codes". They can detect and correct up to ‘$t$’ random errors per code word. The Hamming SEC codes can also be described as BCH codes. The BCH codes are best known codes among those which have block lengths of a few hundred or less. The major advantage of these codes lies in the flexibility in the choice of code parameters viz: block length and code rate. The parameters of some useful BCH codes are given below. Also indicated in the table are the generator polynomials for block lengths up to 31.

**NOTE**: Higher order co-efficients of the generator polynomial are at the left. For example, if we are interested in constructing a (15, 7) BCH code from the table we have (111 010 001) for the co-efficients of the generator polynomial. Hence

$$g(X) = 1 + X^4 + X^6 + X^7 + X^8$$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$k$</th>
<th>$t$</th>
<th>Generator Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>4</td>
<td>1</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>1</td>
<td>1 0 0 1 1 1</td>
</tr>
<tr>
<td>15</td>
<td>7</td>
<td>2</td>
<td>1 1 1 0 1 0 0 0 1</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>3</td>
<td>1 0 1 0 1 1 0 1 1</td>
</tr>
<tr>
<td>31</td>
<td>26</td>
<td>1</td>
<td>1 0 0 1 1 0 1</td>
</tr>
<tr>
<td>31</td>
<td>21</td>
<td>2</td>
<td>1 1 1 1 0 1 0 1 0 0 1</td>
</tr>
<tr>
<td>31</td>
<td>16</td>
<td>3</td>
<td>1 0 0 0 1 1 1 1 0 1 1 0 1 1</td>
</tr>
<tr>
<td>31</td>
<td>11</td>
<td>5</td>
<td>1 0 1 0 0 1 0 0 1 1 1 0 1 0 1</td>
</tr>
<tr>
<td>31</td>
<td>6</td>
<td>7</td>
<td>1 1 0 0 1 0 1 1 1 1 1 0 1 0 0 0 1 0 0 1 1 1</td>
</tr>
</tbody>
</table>

For further higher order codes, the reader can refer to Shu Lin and Costello Jr. The alphabet of a BCH code for $n = (2^m - 1)$ may be represented as the set of elements of an appropriate Galois field, $GF(2^m)$ whose primitive element is $\alpha$. The generator polynomial of the $t$-error correcting BCH code is the least common multiple (LCM) of $M_i(X)$, $M_2(X), \ldots M_{2t}(X)$, where $M_i(X)$ is the minimum polynomial of $\alpha^i$, $i = 1, 2 \ldots 2t$. For further details of the procedure and discussions the reader can refer to J.Das etal.

There are several iterative procedures available for decoding of BCH codes. Majority of them can be programmed on a general purpose digital computer, which in many practical applications form an integral part of data communication networks. Clearly, in such systems software implementation of the algorithms has several advantages over hardware implementation

### 7.8 Cyclic Redundancy Check (CRC) codes:
Cyclic redundancy check codes are extremely well suited for "error detection". The two important reasons for this statement are, (1) they can be designed to detect many combinations of likely errors. (2) The implementation of both encoding and error detecting circuits is practical. Accordingly, all error detecting codes used in practice, virtually, are of the CRC -type. In an n-bit received word if a contiguous sequence of 'b-bits' in which the first and the last bits and any number of intermediate bits are received in error, then we say a CRC "error burst' of length 'b' has occurred. Such an error burst may also include an end-shifted version of the contiguous sequence.

In any event, Binary (n, k) CRC codes are capable of detecting the following error patterns:

1. All CRC error bursts of length (n-k) or less.
2. A fraction of (1 - 2^{(n-k-1)}) of CRC error bursts of length (n - k + 1).
3. A fraction (1-2^{(n-k)}) of CRC error bursts of length greater than (n - k + 1).
4. All combinations of (d_{min}-1) or fewer errors.
5. All error patterns with an odd number of errors if the generator polynomial g (X) has an even number of non zero coefficients.

Generator polynomials of three CRC codes, internationally accepted as standards are listed below. All three contain (1 +X) as a prime factor. The CRC-12 code is used when the character lengths is 6-bits. The others are used for 8-bit characters.

* CRC-12 code: g (X) = 1 + X + X^2+X^3 + X^{11} + X^{12} *

* CRC-16 code: g (X) = 1 +X^2 + X^{15} + X^{16}

* CRC-CCITT code: g (X) = 1 + X^5 + x^{12} + X^{16}

(Expansion of CCITT: "Commitè Consultatif International Téléphonique et Télégraphique" a Geneva-based organization made up of telephone companies from all over the world)

7.9 Maximum Length codes:

For any integer m ≥ 3, the maximum length codes exist with parameters:

| Block length | n = 2^m - 1 |
| Message bits | k = m |
| Minimum distance | d_{min} = 2^{m-1} |

Maximum length codes are generated by polynomials of the form g(X) = \frac{1+X^n}{p(X)}

Maximum length codes are generated by polynomials of degree 'm'. Notice that any cyclic code generated by a primitive polynomial is a Hamming code of d_{min} = 3. It follows then that the maximum length codes are the 'duals' of Hamming codes. These codes are also referred to as 'pseudo Noise (PN) codes' or "simplex codes".

7.10 Majority Logic Decodable Codes:

These codes form a smaller sub-class of cyclic codes than do the BCH codes. Their error
correction capabilities, for most interesting values code length and efficiency, are much inferior to \( \text{BCH} \) codes. The main advantage is that the decoding can be performed using simple circuits. The concepts are illustrated here with two examples.

Consider a \((7, 3)\) simplex code, which is dual to the \((7, 4)\) Hamming code. Here \(d_{\text{min}}=4\) and \(t = 1\). This code is generated by \(G\) and corresponding parity check matrix \(H\) given below:

\[
G = \begin{bmatrix}
1 & 0 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 1
\end{bmatrix} \quad H = \begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 1
\end{bmatrix}
\]

The error vector \(e=(e_0, e_1, e_2, e_3, e_4, e_5, e_6)\) is checked by forming the syndromes:

\[
s_0 = e_0 + e_4 + e_5; \quad s_2 = e_2 + e_4 + e_5 + e_6; \\
\]

\[
A_1 = s_1 = e_1 + e_5 + e_6; \quad A_2 = s_3 = e_3 + e_4 + e_6; \quad A_3 = s_0 + s_2 = e_0 + e_2 + e_6
\]

It is observed that all the check sums check for the error bit \(e_6\) and no other bit is checked by more than one check sum. Then a majority decision can be taken that \(e_6 = 1\) if two or more \(A_i\)'s are non-zero. If \(e_6 = 0\) and any other bit is in error then only one of the \(A_i\)'s will be non-zero. It is said that the check sums \(A_i\)'s are orthogonal on the error bit \(e_6\). A circulating \(SR\) memory circuit along with a few logic circuits shown in Fig 7.16 forms the hardware of the decoder.

Initially, the received code vector \(R(X)\) is loaded into the \(SR\)'s and check sums \(A_1, A_2, A_3\) are formed in the circuit. If \(e_6\) is in error then the majority logic output is '1' and is corrected as it is shifted out of the buffer. If \(e_6\) is correct, then \(e_5\) is checked after one shift of the SR content.
Thus all the bits are checked by successive shifts and the corrected $V(X)$ is reloaded in the buffer. It is possible to correct single errors only by using two of the check sums. However, by using three check sums, the decoder also corrects some double error patterns. The decoder will correct all single errors and detect all double error patterns if the decision is made on the basis of

(i). $A_1 = 1, A_2 = 1, A_3 = 1$ for single errors
(ii). One or more checks fail for double errors.

We have devised the majority logic decoder assuming it is a Block code. However we should not forget that it is also a cyclic code with a generator polynomial

$$g(X) = 1 + X^2 + X^3 + X^4.$$ 

Then one could generate the syndromes at the decoder by using a divider circuit as already discussed. An alternative format for the decoder is shown in Fig 7.17. Successive bits are checked for single error in the block. The feed back shown is optional - The feed back will be needed if it is desired to correct some double error patterns.
Let us consider another example – This time the (7, 4) Hamming code generated by the polynomial $g(X) = 1 + X + X^3$.

Its parity check matrix is: $H = \begin{bmatrix}
1 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 1
\end{bmatrix}$

The Syndromes are seen to be:

$s_0 = e_0 + e_3 + (e_5 + e_6)$
$s_1 = e_1 + e_3 + e_4 + e_5$
$s_2 = e_2 + e_4 + (e_5 + e_6) = e_2 + e_5 + (e_4 + e_6)$

Check sum $A_1 = (s_0 + s_1) = e_0 + e_1 + (e_4 + e_6)$

It is seen that $s_0$ and $s_2$ are orthogonal on $B_1 = (e_5 + e_6)$, as both of them provide check for this sum. Similarly, $A_1$ and $s_2$ are orthogonal on $B_2 = (e_4 + e_6)$. Further $B_1$ and $B_2$ are orthogonal on $e_6$. Therefore it is clear that a two-step majority vote will locate the error on $e_6$. The corresponding decoder is shown in Fig 7.18, where the second level majority logic circuit gives the correction signal and the stored $R(X)$ is corrected as the bits are read out from the buffer. Correct decoding is achieved if $t < d / 2 = 1$ error ($d$ = no. of steps of majority vote). The circuit provides majority vote '1' when the syndrome state is $\{1 0 1\}$. The basic principles of both types of decoders, however, are the same. Detailed discussions on the general principles of Majority logic decoding may be found in Shu-Lin and Costello Jr., J.Das et al and other standard books on error control coding. The idea of this section was "only to introduce the reader to the concept of majority logic decoding.

The Hamming codes ($2^m - 1$, $2^m - m - 1$), $m$ any integer, are majority logic decodable, $(l5, 7)$ BCH code with $t \leq 2$ is 1-step majority logic decodable. Reed-Muller codes, maximum length (simplex) codes Difference set codes and a sub-class of convolutional codes are examples majority logic decodable codes.
7.11 Shortened cyclic codes:

The generator polynomials for the cyclic codes, in general, are determined from among the divisors of $X^n + 1$. Since for a given $n$ and $k$, there are relatively few divisors, there are usually very few cyclic codes of a given length. To overcome this difficulty and to increase the number of pairs $(n, k)$ for which useful codes can be constructed, cyclic codes are often used in shortened form. In this form the last \( j \) information digits are always taken to be zeros and these are not transmitted. The decoder for the original cyclic code can decode the shortened cyclic codes simply by padding the received \((n-j)\) tuples with \( j \) zeros. Hence, we can always construct an \((n-j, k-j)\) shortened cyclic code starting from a \((n, k)\) cyclic code. Therefore the code thus devised is a sub-set of the cyclic code from which it was derived - which means its minimum distance and error correction capability is at least as great as that of the original code. The encoding operation, syndrome calculation and error correction procedures for shortened codes are identical to those described for cyclic codes. This implies shortened cyclic codes inherit nearly all of the implementation advantages and much of the mathematical structure of cyclic codes.

7.12 Golay codes:

Golay code is a \((23, 12)\) perfect binary code that is capable of correcting any combination of three or fewer random errors in a block of 23 bits. It is a perfect code because it satisfies the Hamming bound with the equality sign for \( t = 3 \) as:

\[
\left[ \begin{array}{c} 23 \\ 0 \\ 23 \\ 1 \\ 23 \\ 2 \\ 23 \\ 3 \end{array} \right] + 2^{12} = 2^{23}
\]

The code has been used in many practical systems. The generator polynomial for the code is obtained from the relation $(X^{23}+1) = (X+1) g_1(X) g_2(X)$, where:
The encoder can be implemented using shift registers using either $g_1 (X)$ or $g_2 (X)$ as the divider polynomial. The code has a minimum distance, $d_{\text{min}} = 7$. The extended Golay code, a $(924, 12)$ code has $d_{\text{min}} = 8$. Besides the binary Golay code, there is also a perfect ternary $(11, 6)$ Golay code with $d_{\text{min}} = 5$.

### 7.13 Reed-Solomon Codes:

The Reed-Solomon (RS) codes are an important sub class of BCH codes, where the symbols are from $GF (q)$, $q \neq 2^m$ in general, but usually taken as $2^m$. The encoder for an RS code differs from a binary encoder in that it operates on multiple bits rather than individual bits. A 't'-error correcting RS code has the following parameters.

- **Block length**: $n = (q - 1)$ symbols
- **Number of parity check symbols**: $r = (n - k) = 2t$
- **Minimum distance**: $d_{\text{min}} = (2t + 1)$

The encoder for an RS $(n, k)$ code on $m$-bit symbols groups the incoming binary data stream into blocks, each $km$ bits long. Each block is treated as $k$ symbols, with each symbol having $m$-bits. The encoding algorithm expands a block of $k$ symbols by adding $(n - k)$ redundant symbols. When $m$ is an integer power of 2, the $m$-bit symbols are called 'Bytes'. A popular value of $m$ is 8 and 8-bit RS codes are extremely powerful. Notice that no $(n, k)$ linear block code can have $d_{\text{min}} > (n - k + 1)$. For the RS code the block length is one less than the size of a code symbol and minimum distance is one greater than the number of parity symbols - "The $d_{\text{min}}$ is always equal to the design distance of the code". An $(n, k)$ linear block code for which $d_{\text{min}} = (n - k - l)$ is called 'Maximum - distance separable' code. Accordingly, every RS code is 'maximum - distance separable' code-They make highly efficient use of redundancy and can be adjusted to accommodate wide range of message sizes. They provide wide range of code rates $(k / n)$ that can be chosen to optimize performance. Further, efficient decoding techniques are available for use with RS codes (usually similar to those of BCH codes).

Reed-Muller Codes (RM codes) are a class of binary group codes, which are majority logic decodable and have a wide range of rates and minimum distances. They are generated from the Hadamard matrices. (Refer J. Das etal).
In block codes, a block of $n$-digits generated by the encoder depends only on the block of $k$-data digits in a particular time unit. These codes can be generated by combinatorial logic circuits. In a convolutional code the block of $n$-digits generated by the encoder in a time unit depends not only on the block of $k$-data digits with in that time unit, but also on the preceding ‘m’ input blocks. An $(n, k, m)$ convolutional code can be implemented with $k$-input, $n$-output sequential circuit with input memory $m$. Generally, $k$ and $n$ are small integers with $k < n$ but the memory order $m$ must be made large to achieve low error probabilities. In the important special case when $k = 1$, the information sequence is not divided into blocks but can be processed continuously.

Similar to block codes, convolutional codes can be designed to either detect or correct errors. However, since the data are usually re-transmitted in blocks, block codes are better suited for error detection and convolutional codes are mainly used for error correction.

Convolutional codes were first introduced by Elias in 1955 as an alternative to block codes. This was followed later by Wozen Craft, Massey, Fano, Viterbi, Omura and others. A detailed discussion and survey of the application of convolutional codes to practical communication channels can be found in Shu-Lin & Costello Jr., J. Das et al and other standard books on error control coding.

To facilitate easy understanding we follow the popular methods of representing convolutional encoders starting with a connection pictorial - needed for all descriptions followed by connection vectors.

### 8.1 Connection Pictorial Representation:

The encoder for a (rate $1/2$, $K = 3$) or $(2, 1, 2)$ convolutional code is shown in Fig.8.1. Both sketches shown are one and the same. While in Fig.8.1 (a) we have shown a 3-bit register, by noting that the content of the third stage is simply the output of the second stage, the circuit is modified using only two shift register stages. This modification, then, clearly tells us that” the memory requirement $m = 2$. For every bit inputted the encoder produces two bits at its output. Thus the encoder is labeled $(n, k, m)$→ $(2, 1, 2)$ encoder.

At each input bit time one bit is shifted into the left most stage and the bits that were present in the registers shifted to the right by one position. Output switch (commutator /MUX) samples the output of each X-OR gate and forms the code symbol pairs for the bits introduced. The final code is obtained after flushing the encoder with "m" zero's where ‘m’- is the memory order (In Fig.8.1, $m = 2$). The sequence of operations performed by the encoder of Fig.8.1 for an input sequence $u = (101)$ are
illustrated diagrammatically in Fig 8.2.

From Fig 8.2, the encoding procedure can be understood clearly. Initially the registers are in Re-set mode i.e. \((0, 0)\). At the first time unit the input bit is 1. This bit enters the first register and pushes out its previous content namely ‘0’ as shown, which will now enter the second register and pushes out its previous content. All these bits as indicated are passed on to the X-OR gates and the output pair \((1, 1)\) is obtained. The same steps are repeated until time unit 4, where zeros are introduced to clear the register contents producing two more output pairs. At time unit 6, if an additional ‘0’ is introduced the encoder is re-set and the output pair \((0, 0)\) obtained. However, this step is not absolutely necessary as the next bit, whatever it is, will flush out the content of the second register. The ‘0’ and the ‘1’ indicated at the output of second register at time unit 5 now vanishes.

Hence after \((L+m) = 3 + 2 = 5\) time units, the output sequence will read \(v = (11, 10, 00, 10, 11)\). (Note: \(L = \text{length of the input sequence}\)). This then is the code word produced by the encoder. It is very important to remember that “**Left most symbols represent earliest transmission**”.

As already mentioned the convolutional codes are intended for the purpose of error correction. However, it suffers from the ‘problem of choosing connections’ to yield good distance properties. The selection of connections indeed is very complicated and has not been solved yet. Still, good codes have been developed by computer search techniques for all constraint lengths less than 20. Another point to be noted is that the convolutional codes do not have any particular block size. They can be periodically truncated. Only thing is that they require \(m\)-zeros to be appended to the end of the input sequence for the purpose of ‘clearing’ or ‘flushing’ or ‘re-setting’ of the encoding shift registers off the data bits. These added zeros carry no information but have the effect of reducing the code rate below \((k/n)\). To keep the code rate close to \((k/n)\), the truncation period is generally made as long as practical.

The encoding procedure as depicted pictorially in Fig 8.2 is rather tedious. We can approach the encoder in terms of “Impulse response” or “generator sequence” which merely represents the
response of the encoder to a single ‘1’ bit that moves through it.

8.2 Convolutional Encoding – Time domain approach:

The encoder for a \((2, 1, 3)\) code is shown in Fig. 8.3. Here the encoder consists of \(m=3\) stage shift register, \(n=2\) modulo-2 adders (X-OR gates) and a multiplexer for serializing the encoder outputs. Notice that module-2 addition is a linear operation and it follows that all convolution encoders can be implemented using a “linear feed forward shift register circuit”.

The “information sequence” \(u = (u_1, u_2, u_3 \ldots \ldots)\) enters the encoder one bit at a time starting from \(u_1\). As the name implies, a convolutional encoder operates by performing convolutions on the information sequence. Specifically, the encoder output sequences, in this case \(v^{(1)} = \{v_1^{(1)}, v_2^{(1)}, v_3^{(1)} \ldots \} \) and \(v^{(2)} = \{v_1^{(2)}, v_2^{(2)}, v_3^{(2)} \ldots \} \) are obtained by the discrete convolution of the information sequence with the encoder 'impulse responses'. The impulse responses are obtained by determining the output sequences of the encoder produced by the input sequence \(u = (1, 0, 0, 0 \ldots)\). The impulse responses so defined are called 'generator sequences' of the code. Since the encoder has a \(m\)-time unit memory the impulse responses can last at most \((m+1)\) time units (That is a total of \((m+1)\) shifts are necessary for a message bit to enter the shift register and finally come out) and are written as:

\[
g^{(i)} = \{g_1^{(i)}, g_2^{(i)}, g_3^{(i)} \ldots g_{m+1}^{(i)}\}.
\]

For the encoder of Fig.8.3, we require the two impulse responses,\n
\[
g^{(1)} = \{g_1^{(1)}, g_2^{(1)}, g_3^{(1)}, g_4^{(1)}\} \quad \text{and} \quad g^{(2)} = \{g_1^{(2)}, g_2^{(2)}, g_3^{(2)}, g_4^{(2)}\}
\]

By inspection, these can be written as: \(g^{(1)} = \{1, 0, 1, 1\}\) and \(g^{(2)} = \{1, 1, 1, 1\}\)

Observe that the generator sequences represented here is simply the ‘connection vectors’ of the encoder. In the sequences a '1' indicates a connection and a '0' indicates no connection to the corresponding X-OR gate. If we group the elements of the generator sequences so found in to pairs, we get the overall impulse response of the encoder, Thus for the encoder of Fig 8.3, the ‘overall impulse response’ will be:

\[
v = (11, 01, 11, 11)
\]

The encoder outputs are defined by the convolution sums:
\[ v^{(1)} = u * g^{(1)} \quad \text{………………………….} \quad (8.1 \ a) \]
\[ v^{(2)} = u * g^{(2)} \quad \text{………………………….} \quad (8.1 \ b) \]

Where * denotes the ‘discrete convolution’, which implies:

\[ v^{(j)}_l = \sum_{i=0}^{m} u_{l-i} \cdot g_{l+1}^{(j)} \]
\[ = u_1 g_1^{(j)} + u_{l-1} g_2^{(j)} + u_{l-2} g_3^{(j)} + \ldots + u_{l-m} g_{m+1}^{(j)} \]
\[ \quad \text{………………………….} \quad (8.2) \]

for \( j = 1, 2 \) and where \( u_{l-i} = 0 \) for all \( l < i \) and all operations are modulo - 2. Hence for the encoder of Fig 8.3, we have:

\[ v^{(1)}_l = u_l + u_{l-2} + u_{l-3} \]
\[ v^{(2)}_l = u_l + u_{l-1} + u_{l-2} + u_{l-3} \]

This can be easily verified by direct inspection of the encoding circuit. After encoding, the two output sequences are multiplexed into a single sequence, called the "code word" for transmission over the channel. The code word is given by:

\[ v = \{ v_1^{(1)}, v_1^{(2)}, v_2^{(1)}, v_2^{(2)}, v_3^{(1)}, v_3^{(2)} \ldots \} \]

Example 8.1:

Suppose the information sequence be \( u = (10111) \). Then the output sequences are:

\[ v^{(1)} = (10111) * (1011) = (100000001), \]
\[ v^{(2)} = (10111) * (111) = (110111101), \]

and the code word is

\[ v = (11, 01, 00, 01, 01, 00, 11) \]

The discrete convolution operation described in Eq (8.2) is merely the addition of shifted impulses. Thus to obtain the encoder output we need only to shift the overall impulse response by 'one branch word', multiply by the corresponding input sequence and then add them. This is illustrated in the table below:
The Modulo-2 sum represents the same sequence as obtained before. There is no confusion at all with respect to indices and suffices! Very easy approach - super position or linear addition of shifted impulse response - demonstrates that the convolutional codes are linear codes just as the block codes and cyclic codes. This approach then permits us to define a 'Generator matrix' for the convolutional encoder. Remember, that interlacing of the generator sequences gives the overall impulse response and hence they are used as the rows of the matrix. The number of rows equals the number of information digits. Therefore the matrix that results would be “Semi-Infinite”. The second and subsequent rows of the matrix are merely the shifted versions of the first row -They are each shifted with respect to each other by "One branch word". If the information sequence \( u \) has a finite length, say \( L \), then \( G \) has \( L \) rows and \( n \times (m +L) \) columns (or \( (m +L) \) branch word columns) and \( v \) has a length of \( n \times (m +L) \) or a length of \( (m +L) \) branch words. Each branch word is of length '\( n \)'. Thus the Generator matrix \( G \), for the encoders of type shown in Fig 8.3 is written as:

\[
G = \begin{bmatrix}
ge_1^{(1)} & g_1^{(2)} & g_2^{(1)} & g_2^{(2)} & g_3^{(1)} & g_3^{(2)} & g_4^{(1)} & g_4^{(2)} & \cdots & \cdots & \cdots \\
ge_1^{(1)} & g_1^{(2)} & g_2^{(1)} & g_2^{(2)} & g_3^{(1)} & g_3^{(2)} & g_4^{(1)} & g_4^{(2)} & \cdots & \cdots & \cdots \\
ge_1^{(1)} & g_1^{(2)} & g_2^{(1)} & g_2^{(2)} & g_3^{(1)} & g_3^{(2)} & g_4^{(1)} & g_4^{(2)} & \cdots & \cdots & \cdots \\
ge_1^{(1)} & g_1^{(2)} & g_2^{(1)} & g_2^{(2)} & g_3^{(1)} & g_3^{(2)} & g_4^{(1)} & g_4^{(2)} & \cdots & \cdots & \cdots \\
ge_1^{(1)} & g_1^{(2)} & g_2^{(1)} & g_2^{(2)} & g_3^{(1)} & g_3^{(2)} & g_4^{(1)} & g_4^{(2)} & \cdots & \cdots & \cdots 
\end{bmatrix} \hspace{1cm} \text{......... (8.3)}
\]

(Blank places are zeros.)

The encoding equations in Matrix form is:

\[
v = u . G \hspace{1cm} \text{......................... (8.4)}
\]

**Example 8.2:**

For the information sequence of Example 8.1, the G matrix has 5 rows and 2(3 +5) =16 columns and we have

\[
G = \begin{bmatrix}
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 
\end{bmatrix}
\]

Performing multiplication, \( v = u . G \) as per Eq (8.4), we get: \( v = (11, 01, 00, 01, 01, 00, 11) \) same as before.

As a second example of a convolutional encoder, consider the \((3, 2, 1)\) encoder shown in Fig.8.4. Here, as \( k =2 \), the encoder consists of two \( m = 1 \) stage shift registers together with \( n = 3 \) modulo -2 adders and two multiplexers. The information sequence enters the encoder \( k = 2 \) bits at a time and can be written as \( u = \{u_1^{(1)}, u_1^{(2)}, u_2^{(1)}, u_2^{(2)}, u_3^{(1)}, u_3^{(2)} \ldots \} \) or as two separate input sequences:

\[
u^{(1)} = \{u_1^{(1)}, u_2^{(1)}, u_3^{(1)} \ldots \} \text{ and } u^{(2)} = \{u_1^{(2)}, u_2^{(2)}, u_3^{(2)} \ldots \}.
\]
There are three generator sequences corresponding to each input sequence. Letting 
\[ g_i^{(j)} = \{g_{i,1}^{(j)}, g_{i,2}^{(j)}, g_{i,3}^{(j)}, \ldots, g_{i,m+1}^{(j)} \} \]
represent the generator sequence corresponding to input \( i \) and output \( j \). The generator sequences for the encoder are:

\[
\begin{align*}
g_1^{(1)} &= (1, 1), g_1^{(2)} = (1, 0), g_1^{(3)} = (1, 0) \\
g_2^{(1)} &= (0, 1), g_2^{(2)} = (1, 1), g_2^{(3)} = (0, 0)
\end{align*}
\]

The encoding equations can be written as:

\[
\begin{align*}
v^{(1)} &= u^{(1)} \ast g_1^{(1)} + u^{(2)} \ast g_2^{(1)} \quad (8.5 \, a) \\
v^{(2)} &= u^{(1)} \ast g_1^{(2)} + u^{(2)} \ast g_2^{(2)} \quad (8.5 \, b) \\
v^{(3)} &= u^{(1)} \ast g_1^{(3)} + u^{(2)} \ast g_2^{(3)} \quad (8.5 \, c)
\end{align*}
\]

The convolution operation implies that:

\[
\begin{align*}
v_1^{(1)} &= u_1^{(1)} + u_{t-1}^{(1)} + u_{t-1}^{(2)} \\
v_1^{(2)} &= u_1^{(1)} + u_{t-1}^{(2)} \\
v_1^{(3)} &= u_1^{(1)}
\end{align*}
\]

as can be seen from the encoding circuit.

After multiplexing, the code word is given by:

\[
v = \{ v_1^{(1)} v_1^{(2)} v_1^{(3)} , v_2^{(1)} v_2^{(2)} v_2^{(3)} , v_3^{(1)} v_3^{(2)} v_3^{(3)} \ldots \}
\]

Example 8.3:

Suppose \( u = (1 \, 1 \, 0 \, 1 \, 1 \, 0) \). Hence \( u^{(1)} = (1 \, 0 \, 1) \) and \( u^{(2)} = (1 \, 1 \, 0) \). Then

\[
\begin{align*}
v^{(1)} &= (1 \, 0 \, 1) \ast (1, 1) + (1 \, 1 \, 0) \ast (0, 1) = (1 \, 0 \, 0 \, 1) \\
v^{(2)} &= (1 \, 0 \, 1) \ast (1, 0) + (1 \, 1 \, 0) \ast (1, 1) = (0 \, 0 \, 0 \, 0) \\
v^{(3)} &= (1 \, 0 \, 1) \ast (1, 0) + (1 \, 1 \, 0) \ast (0, 0) = (1 \, 0 \, 1 \, 0)
\end{align*}
\]
\[ v = (101, 000, 001, 100). \]

The generator matrix for a \((3, 2, m)\) code can be written as:

\[
G = \begin{bmatrix}
  g_{11}^{(1)} & g_{11}^{(2)} & g_{11}^{(3)} & g_{12}^{(1)} & g_{12}^{(2)} & g_{12}^{(3)} & \cdots & g_{1, m+1}^{(1)} & g_{1, m+1}^{(2)} & g_{1, m+1}^{(3)} \\
  g_{21}^{(1)} & g_{21}^{(2)} & g_{21}^{(3)} & g_{22}^{(1)} & g_{22}^{(2)} & g_{22}^{(3)} & \cdots & g_{2, m+1}^{(1)} & g_{2, m+1}^{(2)} & g_{2, m+1}^{(3)} \\
  \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\
  \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\
  \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\
  \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\
\end{bmatrix}
\]

The encoding equations in matrix form are again given by \( v = u \, G \). Observe that each set of \( k = 2 \) rows of \( G \) is identical to the preceding set of rows but shifted by \( n = 3 \) places or one branch word to the right.

**Example 8.4:**

For the Example 8.3, we have

\[ u = \{u_1^{(1)} u_1^{(2)}, u_2^{(1)} u_2^{(2)}, u_3^{(1)} u_3^{(2)}\} = (11, 01, 10) \]

The generator matrix is:

\[
G = \begin{bmatrix}
  1 & 1 & 1 & 1 & 0 & 0 \\
  0 & 1 & 0 & 1 & 1 & 0 \\
  1 & 1 & 1 & 1 & 0 & 0 \\
  0 & 1 & 0 & 1 & 1 & 0 \\
  1 & 1 & 1 & 1 & 0 & 0 \\
  0 & 1 & 0 & 1 & 1 & 0 \\
\end{bmatrix}
\]

*Remember that the blank places in the matrix are all zeros.*

Performing the matrix multiplication, \( v = u \, G \), we get: \( v = (101, 000, 001, 100) \), again agreeing with our previous computation using discrete convolution.

This second example clearly demonstrates the complexities involved, when the number of input sequences are increased beyond \( k > 1 \), in describing the code. In this case, although the encoder contains \( k \) shift registers all of them need not have the same length. If \( k_i \) is the length of the \( i \)-th shift register, then we define the encoder "memory order, \( m \)" by

\[
m = \text{Max}_{1 \leq i \leq k} k_i
\]

(i.e. the maximum length of all \( k \)-shift registers)

An example of a \((4, 3, 2)\) convolutional encoder in which the shift register lengths are \( 0, 1 \) and \( 2 \) is shown in Fig 8.5.
Since each information bit remains in the encoder up to \((m + 1)\) time units and during each time unit it can affect any of the \(n\)-encoder outputs (which depends on the shift register connections) it follows that "the maximum number of encoder outputs that can be affected by a single information bit" is

\[
\begin{align*}
\mathbf{n}_A & \triangleq n(m + 1) \\
& \equiv 1 + \Delta m n A
\end{align*}
\] (8.8)

\(\mathbf{n}_A\) is called the 'constraint length" of the code. For example, the constraint lengths of the encoders of Figures 8.3, 8.4 and 8.5 are 8, 6 and 12 respectively. Some authors have defined the constraint length (For example: Simon Haykin) as the number of shifts over which a single message bit can influence the encoder output. In an encoder with an \(m\)-stage shift register, the "memory" of the encoder equals \(m\)-message bits, and the constraint length \(n_A = (m + 1)\). However, we shall adopt the definition given in Eq (8.8).

The number of shifts over which a single message bit can influence the encoder output is usually denoted as \(K\). For the encoders of Fig 8.3, 8.4 and 8.5 have values of \(K = 4\), 2 and 3 respectively. The encoder in Fig 8.3 will be accordingly labeled as a 'rate 1/2, \(K = 4\) convolutional encoder. The term \(K\) also signifies the number of branch words in the encoder's impulse response.

Turning back, in the general case of an \((n, k, m)\) code, the generator matrix can be put in the form:

\[
G = \begin{bmatrix}
G_1 & G_2 & G_3 & \cdots & G_m & G_{m+1} \\
G_1 & G_2 & \cdots & G_{m-1} & G_m & G_{m+1} \\
G_1 & \cdots & G_{m-2} & G_{m-1} & G_m & G_{m+1} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
G_1 & \cdots & G_{m-2} & G_{m-1} & G_m & G_{m+1}
\end{bmatrix}
\] (8.9)

Where each \(G_i\) is a \((k \times n)\) sub matrix with entries as below:

\[
G_i = \begin{bmatrix}
g_{1,i}^{(1)} & g_{1,i}^{(2)} & \cdots & g_{1,i}^{(n)} \\
g_{2,i}^{(1)} & g_{2,i}^{(2)} & \cdots & g_{2,i}^{(n)} \\
\vdots & \vdots & \vdots & \vdots \\
g_{k,i}^{(1)} & g_{k,i}^{(2)} & \cdots & g_{k,i}^{(n)}
\end{bmatrix}
\] (8.10)

Notice that each set of \(k\)-rows of \(G\) are identical to the previous set of rows but shifted \(n\)-places to the right. For an information sequence \(u = (u_1, u_2, \ldots)\) where \(u_i = \{u_i^{(1)}, u_i^{(2)}, \ldots u_i^{(k)}\}\), the code word is \(v = (v_1, v_2, \ldots)\) where \(v_j = (v_j^{(1)}, v_j^{(2)}, \ldots v_j^{(n)})\) and \(v = u \cdot G\). Since the code word is a linear combination
of rows of the $G$ matrix it follows that an $(n, k, m)$ convolutional code is a linear code.

Since the convolutional encoder generates $n$-encoded bits for each $k$-message bits, we define $R = k/n$ as the "code rate". However, an information sequence of finite length $L$ is encoded into a code word of length $n \times (L + m)$, where the final $n \times m$ outputs are generated after the last non zero information block has entered the encoder. That is, an information sequence is terminated with all zero blocks in order to clear the encoder memory. (To appreciate this fact, examine the calculations of $v_1^{(j)}$ for the Example 8.1 and 8.3). The terminating sequence of $m$-zeros is called the "Tail of the message". Viewing the convolutional-code as a linear block code, with generator matrix $G$, then the block code rate is given by $kL/n(L + m)$ - the ratio of the number of message bits to the length of the code word. If $L \gg m$, then $L/(L + m) \approx 1$ and the block code rate of a convolutional code and its rate when viewed as a block code would appear to be same. Infact, this is the normal mode of operation for convolutional codes and accordingly we shall not distinguish between the rate of a convolutional code and its rate when viewed as a block code. On the contrary, if $L'$ were small, the effective rate of transmission indeed is $kL/n (L + m)$ and will be below the block code rate by a fractional amount:

$$
\frac{k/n - kL/n(L + m)}{k/n} = \frac{m}{L+m} \quad \text{...................... (8.11)}
$$

and is called "fractional rate loss". Therefore, in order to keep the fractional rate loss at a minimum (near zero), $L'$ is always assumed to be much larger than $m$. For the information 'sequence of Example 8.1, we have $L = 5; m = 3$ and fractional rate loss $= 3/8 = 37.5\%$. If $L$ is made 1000, the fractional rate loss is only $3/1003 \approx 0.3\%$.

### 8.3 Encoding of Convolutional Codes; Transform Domain Approach:

In any linear system, we know that the time domain operation involving the convolution integral can be replaced by the more convenient transform domain operation, involving polynomial multiplication. Since a convolutional encoder can be viewed as a 'linear time invariant finite state machine, we may simplify computation of the adder outputs by applying appropriate transformation. As is done in cyclic codes, each 'sequence in the encoding equations can' be replaced by a corresponding polynomial and the convolution operation replaced by polynomial multiplication. For example, for a $(2, 1, m)$ code, the encoding equations become:

$$
v^{(1)}(X) = u(X) g^{(1)}(X) \quad \text{............... (8.12a)}
$$

$$
v^{(2)}(X) = u(X) g^{(2)}(X) \quad \text{............... (8.12b)}
$$

Where $u(X) = u_1 + u_2X + u_3X^2 + \ldots$ is the information polynomial,

$$
v^{(1)}(X) = v_1^{(1)} + v_2^{(1)}X + v_3^{(1)}X^2 + \ldots \quad \text{and}
$$

$$
v^{(2)}(X) = v_1^{(2)} + v_2^{(2)}X + v_3^{(2)}X^2 + \ldots
$$

are the encoded polynomials.

$$
g^{(1)}(X) = g_1^{(1)} + g_2^{(1)}X + g_3^{(1)}X^2 + \ldots \quad \text{and}
$$

$$
g^{(2)}(X) = g_1^{(2)} + g_2^{(2)}X + g_3^{(2)}X^2 + \ldots
$$
are the “generator polynomials” of the code; and all operations are modulo-2. After multiplexing, the code word becomes:

\[ v(X) = v^{(1)}(X^2) + X v^{(2)}(X^2) \]  \hspace{1cm} (8.13)

The indeterminate 'X' can be regarded as a “unit-delay operator”, the power of X defining the number of time units by which the associated bit is delayed with respect to the initial bit in the sequence.

Example 8.5:

For the (2, 1, 3) encoder of Fig 8.3, the impulse responses were: \( g^{(1)} = (1, 0, 1, 1) \), and \( g^{(2)} = (1, 1, 1, 1) \)

The generator polynomials are: \( g^{(1)}(X) = 1 + X^2 + X^3 \), and \( g^{(2)}(X) = 1 + X + X^2 + X^3 \)

For the information sequence \( u = (1, 0, 1, 1, 1) \); the information polynomial is: \( u(X) = 1 + X^2 + X^3 + X^4 \)

The two code polynomials are then:

\[ v^{(1)}(X) = u(X) g^{(1)}(X) = (1 + X^2 + X^3 + X^4) (1 + X^2 + X^3) = 1 + X^7 \]

\[ v^{(2)}(X) = u(X) g^{(2)}(X) = (1 + X^2 + X^3 + X^4) (1 + X + X^2 + X^3) = 1 + X + X^3 + X^4 + X^5 + X^7 \]

From the polynomials so obtained we can immediately write:

\[ v^{(1)} = (1 1 0 0 0 0 1), \text{ and } v^{(2)} = (1 1 0 1 1 0 1) \]

Pairing the components we then get the code word \( v = (11, 01, 00, 01, 01, 00, 11) \).

We may use the multiplexing technique of Eq (8.13) and write:

\[ v^{(1)}(X^2) = 1 + X^{14} \text{ and } v^{(2)}(X^2) = 1 + X^2 + X^6 + X^8 + X^{10} + X^{14}; X v^{(2)}(X^2) = X + X^3 + X^7 + X^9 + X^{11} + X^{15}; \]

and the code polynomial is: \( v(X) = v^{(1)}(X^2) + X v^{(2)}(X^2) = 1 + X + X^3 + X^7 + X^9 + X^{11} + X^{14} + X^{15} \)

Hence the code word is: \( v = (1 1, 0 1, 0 0, 0 1, 0 1, 0 0, 1 1) \); this is exactly the same as obtained earlier.

The generator polynomials of an encoder can be determined directly from its circuit diagram. Specifically, the co-efficient of \( X^l \) is a '1' if there is a "connection" from the \( l \)-th shift register stage to the input of the adder of interest and a '0' otherwise. Since the last stage of the shift register in an \((n, l)\) code must be connected to at least one output, it follows that at least one generator polynomial should have a degree equal to the shift register length 'm', i.e.

\[ m = \max_{1 \leq j \leq n} \{ \text{deg } g^{(j)}(X) \} \]  \hspace{1cm} (8.14)

In an \((n, k)\) code, where \( k > 1 \), there are \( n \)-generator polynomials for each of the \( k \)-inputs, each set representing the connections from one of the shift registers to the \( n \)-outputs. Hence, the length \( k_l \) of the \( l \)-th shift register is given by:
\[ K_l = \max_{1 \leq j \leq n} \{ \deg g_l^{(j)}(X) \}, \quad 1 \leq l \leq k \]  \hspace{1cm} (8.15)\\

Where \( g_l^{(j)}(X) \) is the generator polynomial relating the \( l \)-th input to the \( j \)-th output and the encoder memory order \( m \) is:

\[ m = \max_{1 \leq l \leq k} K_l = \max_{1 \leq j \leq \deg g_l^{(j)}(X)} \]  \hspace{1cm} (8.16)\\

Since the encoder is a linear system and \( u_l^{(i)}(X) \) represents the \( l \)-th input sequence and \( v_l^{(j)}(X) \) represents the \( j \)-th output sequence the generator polynomial \( g_l^{(j)}(X) \) can be regarded as the ‘encoder transfer function’ relating the input - \( l \) to the output – \( j \). For the \( k \)-input, \( n \)-output linear system there are a total of \( k \times n \) transfer functions which can be represented as a \( (k \times n) \) "transfer function matrix".

\[ G(X) = \begin{bmatrix} g_1^{(1)}(X), & g_1^{(2)}(X), & \cdots & g_1^{(n)}(X) \\ g_2^{(1)}(X), & g_2^{(2)}(X), & \cdots & g_2^{(n)}(X) \\ \vdots & \vdots & \ddots & \vdots \\ g_k^{(1)}(X), & g_k^{(2)}(X), & \cdots & g_k^{(n)}(X) \end{bmatrix} \]  \hspace{1cm} (8.17)\\

Using the transfer function matrix, the encoding equations for an \((n, k, m)\) code can be expressed as

\[ V(X) = U(X) G(X) \]  \hspace{1cm} (8.18)\\

\[ U(X) = [u_1^{(1)}(X), u_1^{(2)}(X) \ldots u_1^{(k)}(X)] \] is the \( k \)-vector, representing the information polynomials, and.
\[ V(X) = [v_1^{(1)}(X), v_1^{(2)}(X) \ldots v_1^{(n)}(X)] \] is the \( n \)-vector representing the encoded sequences. After multiplexing, the code word becomes:

\[ v(X) = v_1^{(1)}(X^n) + X v_1^{(2)}(X^n) + X^2 v_1^{(3)}(X^n) + \ldots + X^{n-l} v_1^{(n)}(X^n) \]  \hspace{1cm} (8.19)\\

**Example 8.6:**

For the encoder of Fig 8.4, we have:

\[ g_1^{(1)}(X) = 1 + X, \quad g_1^{(2)}(X) = X \]
\[ g_1^{(3)}(X) = 1, \quad g_1^{(3)}(X) = 1 + X \]
\[ \therefore G(X) = \begin{bmatrix} 1 + X & 1 & 1 \\ X & 1 + X & 0 \end{bmatrix} \]

For the information sequence \( u^{(1)} = (1 0 1), u^{(2)} = (1 1 0) \), the information polynomials are:

\[ u_1^{(1)}(X) = 1 + X^2, u_1^{(2)}(X) = 1 + X \]

Then

\[ V(X) = [v_1^{(1)}(X), v_1^{(2)}(X), v_1^{(3)}(X)] = [I + X^2, 1 + X, 1 + X] \]
Hence the code word is:

\[ v(X) = v^{(1)}(X^3) + Xv^{(2)}(X^3) + X^2v^{(3)}(X^3) \]

\[ = (1 + X^9) + X(0) + X^2(1 + X^6) \]

\[ = 1 + X^2 + X^8 + X^9 \]

\[ \therefore v = (1 \ 0 \ 1, \ 0 \ 0 \ 0, \ 0 \ 0 \ 1, \ 1 \ 0 \ 0). \]

This is exactly the same as that obtained in Example 8.3.

From Eq (8.17) and (8.18) it follows that:

8.5.1 State Diagrams:

The state of an encoder is defined as its shift register contents. For an \((n, k, m)\) code with \(k > 1\), \(i\)-th shift register contains \(K_i\) previous information bits. Defining \(K = \sum_{i=1}^{k} K_i\) as the total encoder memory (\(m\) - represents the memory order which we have defined as the maximum length of any shift register), the encoder state at time unit \(T'\) when the encoder inputs are, \(\{u_{l-1}^{(1)}, u_{l-1}^{(2)} \cdots u_{l-1}^{(k)}\}\), are the binary \(k\)-tuple of inputs:

\(\{u_{l-1}^{(1)}u_{l-2}^{(1)}u_{l-3}^{(1)} \cdots u_{l-k}^{(1)}; u_{l-1}^{(2)}u_{l-2}^{(2)}u_{l-3}^{(2)} \cdots u_{l-k}^{(2)}; \cdots; u_{l-1}^{(k)}u_{l-2}^{(k)}u_{l-3}^{(k)} \cdots u_{l-k}^{(k)}\}\),

and there are a total of \(2^k\) different possible states. For a \((n, 1, m)\) code, \(K = K_1 = m\) and the encoder state at time unit \(l\) is simply \(\{u_{l-1}, u_{l-2} \cdots u_{l-m}\}\).

Each new block of \(k\)-inputs causes a transition to a new state. Hence there are \(2^k\) branches leaving each state, one each corresponding to the input block. For an \((n, 1, m)\) code there are only two branches leaving each state. On the state diagram, each branch is labeled with the \(k\)-inputs causing the transition and the \(n\)-corresponding outputs. The state diagram for the convolutional encoder of Fig 8.3 is shown in Fig 8.10. A state table would be, often, more helpful while drawing the state diagram and is as shown.

**State table for the (2, 1, 3) encoder of Fig 8.3**

<table>
<thead>
<tr>
<th>State Binary Description</th>
<th>S₀</th>
<th>S₁</th>
<th>S₂</th>
<th>S₃</th>
<th>S₄</th>
<th>S₅</th>
<th>S₆</th>
<th>S₇</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>100</td>
<td>010</td>
<td>110</td>
<td>001</td>
<td>101</td>
<td>011</td>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>
Recall (or observe from Fig 8.3) that the two output sequences are:

\[ v^{(1)} = u_l + u_{l-2} + u_{l-3} \quad \text{and} \quad v^{(2)} = u_l + u_{l-1} + u_{l-2} + u_{l-3} \]

Till the reader gains some experience, it is advisable to first prepare a transition table using the output equations and then translate the data on to the state diagram. Such a table is as shown below:

**State transition table for the encoder of Fig 8.3**

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Binary Description</th>
<th>Input</th>
<th>Next State</th>
<th>Binary Description</th>
<th>( u_l )</th>
<th>( u_{l-1} )</th>
<th>( u_{l-2} )</th>
<th>( u_{l-3} )</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_0 )</td>
<td>0 0 0</td>
<td>0</td>
<td>( S_0 )</td>
<td>0 0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>( S_1 )</td>
<td>1 0 0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( S_1 )</td>
<td>1 0 0</td>
<td>0</td>
<td>( S_2 )</td>
<td>0 1 0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>( S_3 )</td>
<td>1 1 0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>0 1 0</td>
<td>0</td>
<td>( S_4 )</td>
<td>0 0 1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>( S_5 )</td>
<td>1 0 1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( S_3 )</td>
<td>1 1 0</td>
<td>0</td>
<td>( S_6 )</td>
<td>0 1 1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>( S_7 )</td>
<td>1 1 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( S_4 )</td>
<td>0 0 1</td>
<td>0</td>
<td>( S_0 )</td>
<td>0 0 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>( S_1 )</td>
<td>1 0 0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( S_5 )</td>
<td>1 0 1</td>
<td>0</td>
<td>( S_2 )</td>
<td>0 1 0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>( S_3 )</td>
<td>1 1 0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( S_6 )</td>
<td>0 1 1</td>
<td>0</td>
<td>( S_4 )</td>
<td>0 0 1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>( S_5 )</td>
<td>1 0 1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( S_7 )</td>
<td>1 1 1</td>
<td>0</td>
<td>( S_6 )</td>
<td>0 1 1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>( S_7 )</td>
<td>1 1 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

For example, if the shift registers were in state \( S_5 \), whose binary description is \( 101 \), an input ‘1’ causes this state to change over to the new state \( S_3 \) whose binary description is \( 110 \) while producing an output (0 1). Observe that the inputs causing the transition are shown first, followed by
the corresponding output sequences shown with in parenthesis.

Assuming that the shift registers are initially in the state $S_0$ (the all zero state) the code word corresponding to any information sequence can be obtained by following the path through the state diagram determined by the information sequence and noting the corresponding outputs on the branch labels. Following the last nonzero block, the encoder is returned to state $S_0$ by a sequence of $m$-all-zero block appended to the information sequence. For example, in Fig 8.10, if $u = (11101)$, the code word is $v = (11, 10, 01, 11, 10, 11, 10)$ the path followed is shown in thin gray lines with arrows and the input bit written along in thin gray. The $m = 3$ zeros appended are indicated in gray which is much lighter compared to the information bits.

**Example 8.12: A (2, 1, 2) Convolutional Encoder:**

Consider the encoder shown in Fig 8.15. *We shall use this example for discussing further graphical representations viz. Trees, and Trellis.*

![Convolutional encoder diagram](Fig 8.15 A (2, 1, 2) Convolutional encoder)

For this encoder we have: $v_1^{(1)} = u + u_{l-1} + u_{l-2}$ and $v_1^{(2)} = u + u_{l-2}$

The state transition table is as follows.

<table>
<thead>
<tr>
<th>Previous state</th>
<th>Binary description</th>
<th>Input</th>
<th>Next State</th>
<th>Binary description</th>
<th>$u_1$ $u_{l-1}$ $u_{l-2}$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S0$</td>
<td>0 0</td>
<td>0</td>
<td>$S0$</td>
<td>0 0</td>
<td>0 0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>$S1$</td>
<td>1 0</td>
<td>0</td>
<td>$S1$</td>
<td>1 0</td>
<td>1 0 0</td>
<td>1 1</td>
</tr>
<tr>
<td>$S2$</td>
<td>0 1</td>
<td>0</td>
<td>$S0$</td>
<td>0 0</td>
<td>0 0 1</td>
<td>1 1</td>
</tr>
<tr>
<td>$S3$</td>
<td>1 1</td>
<td>1</td>
<td>$S1$</td>
<td>1 0</td>
<td>1 0 1</td>
<td>0 0</td>
</tr>
</tbody>
</table>

The state diagram and the augmented state diagram for computing the ‘complete path enumerator function’ for the encoder are shown in Fig 8.16.
8.5.3 Catastrophic Code:

We shall re-consider the catastrophic codes considered earlier. The state diagram of a \((2, 1, 2)\) code is shown in Fig 8.17. Notice that for a catastrophic code "the state diagram consists of a loop of zero weight other than the self loop around the state \(S_0\)."

It is the characteristic of a catastrophic code that an information sequence of infinite weight produces a finite weight code word.

In a non-catastrophic code which contains no zero weight loops other than the self loop around the all zero state \(S_0\), all infinite weight information sequences must generate infinite weight code words, and minimum weight code word always has a finite length.

The best achievable \(d_{\text{free}}\) for a convolutional code with a given rate and constraint length has not been determined exactly. However, results are available with respect to the lower and upper bounds on \(d_{\text{free}}\) for the best code, obtained using random coding approach. It is observed that more free distance is available with non-systematic codes of a given rate and constraint length compared to the systematic codes and thus has important consequences when a code with large \(d_{\text{free}}\) must be selected for use with either the Viterbi or Sequential decoding.

8.5.4 Tree and Trellis Diagrams:
Let us now consider other graphical means of portraying convolutional codes. The state diagram can be re-drawn as a 'Tree graph'. The convention followed is: If the input is a '0', then the upper path is followed and if the input is a '1', then the lower path is followed. A vertical line is called a 'Node' and a horizontal line is called 'Branch'. The output code words for each input bit are shown on the branches. The encoder output for any information sequence can be traced through the tree paths. The tree graph for the (2, 1, 2) encoder of Fig 8.15 is shown in Fig 8.18. The state transition table can be conveniently used in constructing the tree graph.

Following the procedure just described we find that the encoded sequence for an information sequence (10011) is (11, 10, 11, 11, 01) which agrees with the first 5 pairs of bits of the actual
encoded sequence. Since the encoder has a memory = 2 we require two more bits to clear and re-set the encoder. Hence to obtain the complete code sequence corresponding to an information sequence of length $kL$, the tree graph is to extended by $n(m-l)$ time units and this extended part is called the "Tail of the tree", and the $2kL$ right most nodes are called the "Terminal nodes" of the tree. Thus the extended tree diagram for the $(2, 1, 2)$ encoder, for the information sequence $(10011)$ is as in Fig 8.19 and the complete encoded sequence is $(11, 10, 11, 11, 01, 01, 11)$.

![Fig 8.19 Illustration of the 'Tail of the tree'](image)

At this juncture, a very important clue for the student in drawing tree diagrams neatly and correctly, without wasting time appears pertinent. As the length of the input sequence $L$ increases the number of right most nodes increase as $2^L$. Hence for a specified sequence length, $L$, compute $2^L$. Mark $2^L$ equally spaced points at the rightmost portion of your page, leaving space to complete the $m$ tail branches. Join two points at a time to obtain $2^{L-1}$ nodes. Repeat the procedure until you get only one node at the left most portion of your page. The procedure is illustrated diagrammatically in Fig 8.20 for $L = 3$. Once you get the tree structure, now you can fill in the needed information either looking back to the state transition table or working out logically.

![Fig 8.20 Procedure for drawing neat tree diagram](image)
From Fig 8.18, observe that the tree becomes "repetitive' after the first three branches. Beyond the third branch, the nodes labeled $S_0$ are identical and so are all the other pairs of nodes that are identically labeled. Since the encoder has a memory $m = 2$, it follows that when the third information bit enters the encoder, the first message bit is shifted out of the register. Consequently, after the third branch the information sequences $(000u_3u_4\ldots)$ and $(100u_3u_4\ldots)$ generate the same code symbols and the pair of nodes labeled $S_0$ may be joined together. The same logic holds for the other nodes.

Accordingly, we may collapse the tree graph of Fig 8.18 into a new form of Fig 8.21 called a "Trellis". It is so called because Trellis is a tree like structure with re-merging branches (You will have seen the trusses and trellis used in building construction).

The Trellis diagram contain $(L + m + 1)$ time units or levels (or depth) and these are labeled from 0 to $(L + m)$ (0 to 7 for the case with $L = 5$ for encoder of Fig 8.15 as shown in Fig8.21. The convention followed in drawing the Trellis is that "a code branch produced by an input '0' is drawn as a solid line while that produced by an input '1' is shown by dashed lines". The code words produced by the transitions are also indicated on the diagram. Each input sequence corresponds to a specific path through the trellis. The reader can readily verify that the encoder output corresponding to the sequence $u = (10011)$ is indeed $v = (11, 10, 11, 11, 01, 01, 11)$ the path followed being as shown in Fig 8.22.
A trellis is more instructive than a tree graph in that the finite state behaviour of the encoder is explicitly brought-out. Assuming that the encoder always starts in state $S_0$ and returns to $S_0$, the first $m$ - time units (levels or depth) correspond to the encoder’s departure from $S_0$ and the last $m$-levels corresponds to its return to $S_0$. Clearly, not all the states can be reached in these two portions of the Trellis. However, in the centre portion of the Trellis, all states are possible and each level (time unit) contains a replica of the state diagram and is shown in Fig 8.23. There are two branches leaving and entering each state.

In the general case of an $(n, k, m)$ code and an information sequence of length $kL$, there are $2^k$ branches leaving and entering each state and $2^{kL}$ distinct paths through the trellis corresponding to the $2^{kL}$ code words.

The following observations can be made from the Trellis diagram.

1. There are no fundamental paths at distance 1, 2 or 3 from the all-zero path.

2. There is a single fundamental path at distance 5 from the all zero path. It diverges from the all-zero path three branches back and it differs from the all-zero path in the single input bit.

3. There are two fundamental paths at a distance 6 from the all zero path. One path diverges from the all zero path four branches back and the other five branches back. Both paths differ from the all-zero path in two input bits. The above observations are depicted in Fig 8.24(a).

4. There are four fundamental paths at a distance 7 from the all-zero path. One path diverges from the all zero path five branches back, two other paths six branches back and the fourth path diverges seven branches back as shown in Fig 8.24(b). They all differ from the all zero path in three input bits. This information can be compared with those obtained from the complete path enumerator function found earlier.
For the (2, 1, 2) code we have, accordingly, \( d_{\text{free}} = 5 \). This implies that up to two errors in the received sequence are correctable. For two or fewer transmission errors the received sequence will be at most a Hamming distance of 2 from the transmitted sequence but at least a Hamming distance of 3 from any other code sequence in the code. That is, the received sequence remains closer to the transmitted sequence in spite of the presence of pairs of transmission errors than any other possible sequence.

**8.6 Maximum Likely-hood decoding of Convolutional Codes:**

Hitherto we were concentrating on the structural properties and graphical portrayal of convolutional codes. Now we shall turn our attention to some of the decoding techniques. Suppose that an information sequence \( u = (u_1, u_2, u_3 \ldots u_L) \) of length \( L \) is encoded into a code word \( v = (v_1, v_2, v_3 \ldots v_{L+m}) \) of length \( N = n \ (L+m) \) and that a \( Q \)-ary sequence \( r = (r_1, r_2, r_3 \ldots r_{L+m}) \) is received over a binary input - \( Q \)-ary output discrete memory less channel (DMC).

Alternatively; these sequences can be represented by \( u = (u_1, u_2, u_3 \ldots u_L) \), \( v = (v_1, v_2, v_3 \ldots v_N) \) and \( r = (r_1, r_2, r_3 \ldots r_N) \) where the subscripts now simply represent the ordering of the symbols in each sequence. The decoding problem then is to make an estimate \( \hat{u} \) of the information sequence. Since there is a one-one correspondence between \( u \) and \( v \), the decoding can be equivalently the one of making an estimate \( \hat{v} \) of the code vector. Then we may put \( \hat{u} = u \) if and only if \( \hat{v} = v \). Otherwise there exists a, "decoding error". The "decoding rule" for choosing the estimate \( \hat{v} \) given the received vector \( r \), is optimum when the "probability of decoding error" is made minimum. For equi-probable messages, it follows that the probability of decoding error is minimum if the estimate \( \hat{v} \) is chosen to maximize the "log-likely hood function". Let \( P(r \mid v) \) represent conditional probability of receiving \( r \) given that \( v \) was transmitted. The log-likely hood function is then \( \ln P(r \mid v) \). The "maximum likely hood decoder" or "decision rule" is to “choose the estimate \( \hat{v} \) when the likely hood function, ‘\( \ln P(r \mid v) \)’, is a maximum”.

For a DMC, the two sequences \( v \) and \( r \) may differ in some locations because of the channel noise. With \( v \) and \( r \) as assumed we then have:

\[
P(r \mid v) = \prod_{i=1}^{N} p(r_i \mid v_i)
\]

Accordingly, the log-likely hood function becomes

\[
\ln P(r \mid v) = \sum_{i=1}^{N} \ln p(r_i \mid v_i)
\]

Let \( p(r_i \mid v_i) = \begin{cases} p & \text{.... if } r_i \neq v_i \\ (1-p) & \text{.... if } r_i = v_i \end{cases} \)

Further, let the received vector \( r \) differ from the transmitted vector \( v \) in exactly ‘d’ positions. Notice that the number ‘d’ is nothing but the Hamming distance between the vectors \( r \) and \( v \). Then Eq (8.38) can be re-formulated as:

\[
\ln P(r \mid v) = d \ln p + (N-d) \ln (1-p)
\]

\[
= d \ln \left( p / (1-p) \right) + N \ln (1-p)
\]

\[
\text{or } \ln P(r \mid v) = - A d + B
\]

where \( A = - \ln \left( p / (1-p) \right) = \ln \left( (1-p) / p \right) \), \( B=N \ln (1-p) \) are constants.
However, in general, the probability of an error occurring is low and without loss of generality we may, assume $p < 1/2$ and it follows that the constant $A$ is positive. Hence from Eq (8.40) we conclude that the, log-likely, hood function becomes maximum when $d'$ becomes minimum.

Thus we may state the maximum likely hood decoding rule for the DMC (Binary symmetric channel here) as follows: "Choose the estimate $\hat{v}$ that minimizes the Hamming distance between the vectors $r$ and $v$".

Thus, for a BSC the MLD (maximum likely hood Decoder) reduces to a "minimum distance decoder". In such a decoder the received vector $r$ is compared with each possible code vector $v$ and the particular one "closest" to $r$ is chosen as the correct transmitted code vector.

Notice that the above procedure ensures minimum error probability when all code words are equally likely.

The log likely hood function $\ln P(r|v)$ is called the "Metric" associated with the path $v$ and is denoted by $M(r|v)$. The terms $\ln P(r_i|v_i)$ are called "Branch metrics" and are denoted by $M(r_i|v_i)$, $i = 1, 2, ..., L + m$, where as the terms, $\ln P(r_i|v_i)$, $i = 1, 2, ..., N$, are called "Bit Metrics". Hence the path metric can be written as:

$$M(r|v) = \sum_{i=1}^{L+m} M(r_i|v_i) = \sum_{i=1}^{N} M(r_i|v_i) \quad \text{................. (8.41)}$$

A partial path metric for the first $j$ branches of the path can be expressed as:

$$M(r|v_j) = \sum_{i=1}^{j} M(r_i|v_j) \quad \text{................. (8.42)}$$

### 8.6.1 The Viterbi Algorithm:

The Viterbi algorithm, when applied to the received sequence $r$ from a DMC finds the path through the trellis with the largest metric. At each step, it compares the metrics of all paths entering each state and stores the path with the largest metric called the "survivor" together with its metric.

**The Algorithm:**

**Step: 1.** Starting at level (i.e. time unit) $j = m$, compute the partial metric for the single path entering each node (state). Store the path (the survivor) and its metric for each state.

**Step: 2.** Increment the level $j$ by 1. Compute the partial metric for all the paths entering a state by adding the branch metric entering that state to the metric of the connecting survivor at the preceding time unit. For each state, store the path with the smallest metric (the survivor), together with its metric and eliminate all other paths.

**Step: 3.** If $j < (L + m)$, repeat **Step 2**. Otherwise stop.

Notice that although we can use the Tree graph for the above decoding, the number of nodes at any level of the Trellis does not continue to grow as the number of incoming message bits increases, instead it remains a constant at $2^m$. 
There are $2^k$ survivors from time unit ‘$m$’ up to time unit $L$, one for each of the $2^k$ states. After $L$ time units there are fewer survivors, since there are fewer states while the encoder is returning to the all-zero state. Finally, at time unit $(L + m)$ there is only one state, the all-zero state and hence only one survivor and the algorithm terminates.

![Trellis Diagram](image)

**Fig 8.25** Survivor after time unit ‘$j$’.

Suppose that the maximum likely hood path is eliminated by the algorithm at time unit $j$ as shown in Fig 8.25. This implies that the partial path metric of the survivor exceeds that of the maximum likely hood path at this point. Now, if the remaining portion of the maximum likely hood path is appended onto the survivor at time unit $j$, then the total metric of this path will exceed the total metric of the maximum likely hood path. But this contradicts the definition of the 'maximum likely hood path' as the 'path with largest metric'. Hence the maximum likely hood path cannot be eliminated by the algorithm and it must be the final survivor and it follows $M(r | \hat{v}) \geq M(r | v)$, $\forall v \neq \hat{v}$. Thus it is clear that the Viterbi algorithm is optimum in the sense that it always finds the maximum likely hood path through the Trellis. From an implementation point of view, however, it would be very inconvenient to deal with fractional numbers. Accordingly, the bit metric $M(r_i | v_i) = \ln P(r_i | v_i)$ can be replaced by "$C_2 \{\ln P(r_i | v_i) + C_1\}$", $C_1$ is any real number and $C_2$ is any positive real number so that the metric can be expressed as an integer. Notice that a path $v$ which maximizes $M(r | v) = \sum_{i=1}^{N} M(r_i | v_i) = \sum_{i=1}^{N} \ln P(r_i | v_i)$ also maximizes $\sum_{i=1}^{N} C_2 [\ln P(r_i | v_i) + C_1]$. Therefore, it is clear that the modified metrics can be used without affecting the performance of the Viterbi algorithm. Observe that we can always choose $C_1$ to make the smallest metric as zero and then $C_2$ can be chosen so that all other metrics can be approximated by nearest integers. Accordingly, there can be many sets of integer metrics possible for a given DMC depending on the choice of $C_2$. The performance of the Viterbi algorithm now becomes slightly sub-optimal due to the use of modified metrics, approximated by nearest integers. However the degradation in performance is typically very low.
Notice that “the final m-branches in any trellis path always corresponds to ‘0’ inputs and hence not considered part of the information sequence”.

As already mentioned, the MLD reduces to a ‘minimum distance decoder’ for a BSC (see Eq 8.40). Hence the distances can be reckoned as metrics and the algorithm must now find the path through the trellis with the smallest metric (i.e. the path closest to \( r \) in Hamming distance). The details of the algorithm are exactly the same, except that the Hamming distance replaces the log likely hood function as the metric and the survivor at each state is the path with the smallest metric. The following example illustrates the concept.

**Example 8.14:**

Suppose the code word \( r = (01, 10, 10, 11, 01, 01, 11) \), from the encoder of Fig 8.15 is received through a BSC. The path traced is shown in Fig 8.29 as dark lines.

![Viterbi algorithm for a BSC.](image)

The estimate of the transmitted code word is

\[
\hat{v} = (11, 10, 11, 11, 01, 01, 11)
\]

and the corresponding information sequence is: \( \hat{u} = (1 0 0 1 1) \)

Notice that the distances of the code words of each branch with respect to the corresponding received words are indicated in brackets. Note also that at some states neither path is crossed out indicating a tie in the metric values of the two paths entering that state. If the final survivor goes
through any of these states there is more than one maximum likely hood path (i.e. there may be more than one path whose distance from \( r \) is a minimum). From an implementation point of view whenever a tie in metric values occur, one path is arbitrarily selected as survivor, because of the non-practicability of storing a variable number of paths. However, this arbitrary resolution of ties has no effect on the decoding error probability. Finally, the Viterbi algorithm cannot give fruitful results when more errors in the transmitted code word than permissible by the \( d_{\text{free}} \) of the code occur. For the example illustrated, the reader can verify that the algorithm fails if there are three errors. Discussion and details about the performance bounds, convolutional code construction, implementation of the Viterbi algorithm etc are beyond the scope of this book.
The coding and decoding schemes discussed so far are designed to combat random or independent errors. We have assumed, in other words, the channel to be “Memory less”. However, practical channels have ‘memory’ and hence exhibit mutually dependent signal transmission impairments. In a ‘fading channel’, such impairment is felt, particularly when the fading varies slowly compared to one symbol duration. The ‘multi-path’ impairment involves signal arrivals at the receiver over two or more paths of different lengths with the effect that the signals “arrive out of phase” with each other and the cumulative received signal are distorted. High-Frequency (HF) and troposphere propagation in radio channels suffer from such a phenomenon. Further, some channels suffer from switching noise and other burst noise (Example: Telephone channels or channels disturbed by pulse jamming impulse noise in the communication channel causes transmission errors to cluster into ‘bursts’). All of these time-correlated impairments results in statistical dependence among successive symbol transmissions. The disturbances tend to cause errors that occur in bursts rather than isolated events.

Once the channel is assumed to have memory, the errors that occur can no longer be characterized as single randomly distributed errors whose occurrence is independent from bit to bit. Majority of the codes: Block, Cyclic or Convolutional codes are designed to combat such random or independent errors. They are, in general, not efficient for correcting burst errors. The channel memory causes degradation in the error performance.

Many coding schemes have been proposed for channels with memory. Greatest problem faced is the difficulty in obtaining accurate models of the frequently time-varying statistics of such channels. We shall briefly discuss some of the basic ideas regarding such codes. (A detailed discussion of burst error correcting codes is beyond the scope of this book). We start with the definition of burst length, ‘b’ and requirements on a (n, k) code to correct error burst. “An error burst of length ‘b’ is defined as a sequence of error symbols confined to ‘b’ consecutive bit positions in which the first and the last bits are non-zero”

For example, an error vector (0010101101100) is a burst of b = 10. The error vector (001000110100) is a burst of b = 8. A code that is capable of correcting all burst errors of length ‘b’ or less is called a “b-burst-error-correcting code”. Or the code is said to have a burst error correcting capability = b. Usually, for proper decoding, the b-symbol bursts are separated by a guard space of ‘g’ symbols. Let us confine, for the present, ourselves for the construction of an (n, k) code for a given n and b with as small a redundancy (n - k) as possible. Then one can make the following observations.

Start with a code vector V with an error burst of length 2b or less. This code vector then may be expressed as a linear combination (vector sum) of the vectors V₁ and V₂ of length b or less. Therefore in the standard array of the code both V₁ and V₂ must be in the same co-set. Further, if one of these is assumed to be the co-set leader (i.e. a correctable error pattern), then the other vector which is in the same co-set turns out to be an un-correctable error pattern. Hence, this code will not be able to correct all error bursts of length b or less. Thus we have established the following assertion:
Concepts of Information Theory & Coding

Assertion-1: “A necessary condition for a \((n, k)\) linear code to be able to correct all error bursts of length \(b\) or less is that no error burst of length \(2b\) or less be a code vector”.

Next let us investigate the structure of code vectors whose non zero components are confined to the first ‘\(b\)’ bits. There are, clearly, \(2^b\) such code vectors. No two such vectors can be in the same co-set of the standard array; otherwise their sum, which is again a burst of \(b\) or less, would be a code vector. Therefore these \(2^b\) vectors must be in \(2^b\) distinct co-sets. For an \((n, k)\) code we know that there are \(2^{(n-k)}\) co-sets. This means that \((n-k)\) must be at least equal to ‘\(b\)’. Thus we have established another important assertion.

Assertion-2: “The number of parity check bits of an \((n, k)\) linear code that has no bursts of length \(b\) or less as a code vector is at least ‘\(b\)’, i.e. \((n-k) \geq b\)

Combining the two assertions, now we can conclude that: “The number of parity check bits of a \(b\)-burst error correcting code must be at least \(2b\)”

\[
\text{i.e. } (n-k) \geq 2b \hspace{2cm} (9.1)
\]

From Eq. (9.1) it follows that the burst-error-correcting capability of an \((n, k)\) code is at most \((n-k)/2\). That is, the upper bound on the burst-error-correcting capability of an \((n, k)\) linear code is governed by:

\[
b \geq \frac{(n-k)}{2} \hspace{2cm} (9.2)
\]

This bound is known by the name “Reiger Bound” and it is used to define the burst correcting efficiency, \(z\), of an \((n, k)\) codes as

\[
z = \frac{2b}{(n-k)} \hspace{2cm} (9.3)
\]

Whereas most useful random error correcting codes have been devised using analytical techniques, for the reasons mentioned at the beginning of this section, the best burst- error correcting codes have to be found through computer aided search procedures. A short list of high rate burst-error-correcting cyclic codes found by computer search is listed in Table-9.1.

If the code is needed for ‘detecting’ error bursts of length ‘\(b\)’, then the number of check bits must satisfy:

\[
(n-k) \geq b \hspace{2cm} (9.4)
\]

Some of the famous block/cyclic and convolution codes designed for correcting burst errors are Burton, Fire, R-S, Berlekemp - Preparata-Massey, Iwadare and Adaptive Gallager codes. Of these Fire codes have been extensively used in practice. A detailed discussion on these codes is available in Shu-Lin et-all and J.Das et-all. (Refer – Bibliography)

9.1 Burst and Random Error Correcting Codes:

In most practical systems, error occurs neither independently, at random, nor in well-defined bursts. As a consequence codes designed for random error correction or single-burst-error correction will become either inefficient or in adequate for tackling a mixture of random and burst errors. For channels in which both types of error occur, it is better to design codes that can correct both types of
Concepts of Information Theory & Coding

errors. One technique, which only requires knowledge of the duration or span of channel memory, not its exact statistical characterization, is the use of “Time diversity or Interleaving”.

Interleaving the code vectors before transmission and de-interleaving after reception causes the burst errors to be spread out in time and the decoder can handle them as if they were random errors. Since, in all practical cases, the channel memory decreases with time separation, the idea behind interleaving is only to separate the code word symbols in time. The interleaving times are similarly filled by symbols of other code words. “Separating the symbols in time effectively transforms a channel with memory to a ‘memory less’ channel”, and there by enable the random error correcting codes to be useful in a bursty-noise channel.

The function of an interleaver is to shuffle the code symbol over a span of several block lengths (for block codes) or several constraint lengths (for convolutional codes). The span needed is usually determined from the knowledge of the burst length. Further the details of the bit-redistribution pattern must be known at the receiver end to facilitate de-interleaving and decoding. Fig 9.1 illustrates the concept of interleaving.

The un-interleaved code words shown in Fig 9.1(a) are assumed to have a single error correcting capability in each six-symbol sequence. If the memory span of the channel is one code word in duration, a six symbol time noise burst could destroy the information contained in one or two code words. On the contrary, suppose encoded data were interleaved as shown in Fig 9.1(b), such that each code symbol of each code word is separated from its pre-interleaved neighbors by a span of six symbol times. The result of an error burst as marked in Fig.9.1 is to affect one code symbol from each of the original six code words. Upon reception, the stream is de-interleaved and decoded as though a single-random error has occurred in each code word. Clearly the burst noise has no degrading effect on the final sequence.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>A₂</td>
<td>A₃</td>
<td>A₄</td>
<td>A₅</td>
<td>A₆</td>
</tr>
</tbody>
</table>

(a)

1 2 3 4 5 6

A₁ B₁ C₁ D₁ E₁ F₁ A₂ B₂ C₂ D₂ E₂ F₂ A₃ B₃ C₃ D₃ E₃ F₃ A₄ B₄ C₄ D₄ E₄ F₄ A₅ B₅ C₅ D₅ E₅ F₅ A₆ B₆ C₆ D₆ E₆ F₆

Fig 9.1 Illustrating Interleaving (a) Original Un-Interleaved code words made up of six code symbols. (b) Interleaved code symbols.

9.1.1 Block Interleaving:

Given an \((n, k)\) cyclic code, it is possible to construct a \((\lambda n, \lambda k)\) cyclic “interlaced code” by simply arranging \(\lambda\) code vectors of the original code into \(\lambda\) rows of a rectangular array and then transmitting them column by column. The parameter ‘\(\lambda\)’ is called “Degree of Interlacing”. By such an arrangement, a burst of length \(\lambda\) or less will affect no more than one symbol in each row since transmission is done on a column by column basis. If the original code (whose code words are the rows of the \((\lambda n \times \lambda k)\) matrix) can correct single errors, then the interlaced code can correct single
bursts of length \( \lambda \) or less. On the other hand if the original code has an error correcting capability of \( t \), \( t > 1 \), then the interlaced code is capable of correcting any combination of \( t \)-error bursts of length \( \lambda \) or less. The performance of the \((\lambda n, \lambda k)\) interleaved cyclic code against purely random errors is identical to that of the original \((n, k)\) cyclic code from which it was generated.

The block interleaver accepts the coded symbols in blocks from the encoder, permutes the symbols and then feeds the re-arranged symbols to the modulator. The usual permutation of block is accomplished by ‘filling the rows’ of a \(' \lambda '\) row by \(' n '\)-column ‘array’ with the encoded sequence. After the array is completely filled, the symbols are then fed to the modulator ‘one column at a time’ and transmitted over the channel. At the receiver the code words are re-assembled in a complimentary manner. The de-interleaver accepts the symbols from the de-modulator, de-interleaves them and feeds them to the decoder - symbols are entered into the de-interleaver array by columns and removed by rows. The most important characteristics of block interleaver may be summarized as follows:

1) Any burst of length less than \( \lambda \) contiguous channel symbol errors result in isolated errors at the de-interleaver output that are separated from each other by at least \( n \)-symbols.

2) Any \( q, \lambda \) burst of errors, where \( q > 1 \), results in output bursts from the de-interleaver of not more than \( \lceil q \rceil \) – symbol errors. Each output burst is separated from the other burst by not less than \( n - \lceil q \rceil \) symbols. The notation \( \lceil q \rceil \) means the smallest integer not less than \( q \) and \( \lfloor q \rfloor \) means the largest integer not greater than \( q \).

3) A periodic sequence of single errors spaced \( \lambda \)-symbols apart results in a single burst of errors of length \( 'n' \) at the de-interleaver output.

4) The interleaver/de-interleaver end – to – end delay is approximately \( 2 \lambda n \) symbol time units to be filled at the receiver before decoding begins. Therefore, the minimum end- to- end delay is \((2 \lambda n-2n+2)\) symbol time units. This does not include any channel propagation delay.

5) The memory requirement, clearly, is \( \lambda n \) symbols for each location (interleaver and de-interleaver). However, since the \( \lambda \times n \) array needs to be (mostly) filled before it can be read out, a memory of \( 2 \lambda n \) symbols is generally implemented at each location to allow the emptying of one \( \lambda \times n \) array while the other is being filled , and vice versa.

Finally a note about the simplest possible implementation aspect- If the original code is cyclic then the interleaved code is also cyclic. If the original code has a generator polynomial \( g(X) \), the interleaved code will have the generator polynomial \( g(X^{\lambda}) \). Hence encoding and decoding can be done using shift registers as was done for cyclic codes. The modification at the decoder for the interleaved codes is done by replacing each shift register stage of the original decoder by \( \lambda \) – stages without changing other connections. This modification now allows the decoder to look at successive rows of the code array on successive decoder cycles. It then follows that if the decoder for the original cyclic code is simple so will it be for the interleaved code. “Interleaving technique is indeed an effective tool for deriving long powerful codes from short optimal codes”.

**Example 9.1:**

Let us consider an interleaver with \( n = 4 \) and \( \lambda = 6 \). The corresponding \((6 \times 4)\) array is shown in Fig. 9.2(a). The symbols are numbered indicating the sequence of transmission. In Fig 9.2(b) is
Concepts of Information Theory & Coding

shown an error burst of five-symbol time units. The symbols shown encircled suffer transmission errors. After de-interleaving at the receiver, the sequence is:

<table>
<thead>
<tr>
<th>← One Code word →</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 7 13 19</td>
</tr>
<tr>
<td>2 8 14 20</td>
</tr>
<tr>
<td>3 9 15 21</td>
</tr>
<tr>
<td>4 10 16 22</td>
</tr>
<tr>
<td>5 11 17 23</td>
</tr>
<tr>
<td>6 12 18 24</td>
</tr>
</tbody>
</table>

Observe that in the de-interleaved sequence, each code word does not have more than one error. The smallest separation between symbols in error is $n = 4$.

Next, with $q = 1.5$, $q\lambda = 9$. Fig 9.2(c) illustrates an example of 9-symbol error burst. After de-interleaving at the receiver, the sequence is:

<table>
<thead>
<tr>
<th>← One Code word →</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 7 13 19</td>
</tr>
<tr>
<td>2 8 14 20</td>
</tr>
<tr>
<td>3 9 15 21</td>
</tr>
<tr>
<td>4 10 16 22</td>
</tr>
<tr>
<td>5 11 17 23</td>
</tr>
<tr>
<td>6 12 18 24</td>
</tr>
</tbody>
</table>

The encircled symbols are in error. It is seen that the bursts consists of no more than $\lceil 1.5 \rceil = 2$ contiguous symbols per code word and they are separated by at least $n - \lfloor 1.5 \rfloor = 4 - 1 = 3$ symbols.

Fig. 9.2(d) illustrates a sequence of single errors spaced by $\lambda = 6$ symbols apart. After de-interleaving at the receiver, the sequence is:

<table>
<thead>
<tr>
<th>← One Code word →</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 7 13 19</td>
</tr>
<tr>
<td>2 8 14 20</td>
</tr>
<tr>
<td>3 9 15 21</td>
</tr>
<tr>
<td>4 10 16 22</td>
</tr>
<tr>
<td>5 11 17 23</td>
</tr>
<tr>
<td>6 12 18 24</td>
</tr>
</tbody>
</table>

Each Row is a 4 bit Code word.
It is seen that de-interleaved sequence has a single error burst of length $n = 4$ symbols. The minimum end-to-end delay due to interleaver and de-interleaver is $(2n - 2n+2) = 42$ symbol time units. Storage of $\lambda n = 24$ symbols required at each end of the channel. As said earlier, storage for $2\lambda n = 48$ symbols would generally be implemented.

**Example 9.2: Interleaver for a BCH code.**

Consider a $(15, 7)$ BCH code generated by $g(X) = 1 + X + X^2 + X^4 + X^8$. For this code $d_{min} = 5$, $t = \frac{d_{min} - 1}{2} = 2$. With $\lambda = 5$, we can construct a $(75, 35)$ interleaved code with a burst error correcting capability of $b = \lambda t = 10$. The arrangement of code words, similar to Example 9.1, is shown in Fig 9.3. A 35-bit message block is divided into five 7-bit message blocks and five code words of length 15 are generated using $g(X)$. These code words are arranged as 5-rows of a $5 \times 15$ matrix. The columns of the matrix are transmitted in the sequence shown as a 75-bit long code vector.

![Fig 9.3 Block Interleaver for a (15, 7) BCH code.](image)

To illustrate the burst and random error correcting capabilities of this code, we have put the bit positions 9, 32 to 38, 66 and 70 in parenthesis, indicating errors occurred in these positions. The de-interleaver now feeds the rows of Fig 9.3 to the decoder. Clearly each row has a maximum of two errors and since the $(15, 7)$ BCH code, from which the rows were constructed, is capable of correcting up to two errors per row. Hence the error pattern shown in parenthesis in the Figure can be corrected. The isolated errors in bit positions 9, 66 and 70 may be thought of as random errors while the cluster of errors in bit positions 32 to 38 as a burst error.

**9.1.2 Convolutional Interleaving:**

Convolution interleave are somewhat simpler and more effective compared to block interleavers. A $(b \times n)$ periodic (convolutional) interleaver is shown in Fig 9.4. The code symbols are shifted sequentially into the bank of $n$-shift registers. Each successive register introduces a delay ‘$b$’. i.e., the successive symbols of a codeword are delayed by \{0, b, 2b \ldots (n-1) b\} symbol units respectively. Because of this, the symbols of one codeword are placed at distances of $b$-symbol units in the channel stream and a burst of length ‘$b$’ separated by a guard space of $(n-1)b$ symbol units only affect one symbol per codeword. In the receiver, the code words are reassembled through complementary delay units and decoded to correct single errors so generated. If the burst length $l > b$ but $l \leq 2b$, then the $(n, k)$ code should be capable of correcting two errors per code words. To
economize on the number of SR’s as shown in Fig 9.4 and clock them at a period of $nT_o$, where $T_o$ = symbol duration. This would then ensure the required delays of $\{b, 2b \ldots (n-1) b\}$ symbol units.

As illustrated in Example 9.2, if one uses a (15, 7) BCH code with $t=2$, then a burst of length $\leq 2b$ can be corrected with a guard space of $(n-1)b = 14b$. This 14 – to – 2 – guard space to burst length ratio is too large, and hence codes with smaller values of $n$ are preferable. Convolutional codes with interleaver may also be used. The important advantage of convolutional interleaver over block interleaver is that, with convolutional interleaving the end-to-end delay is $(n-1)b$ symbol units and the memory required at both ends of the channel is $b(n-1)/2$. This means, there is a reduction of one half in delay and memory over the block interleaving requirements.