DATALINK LAYER

ANNU JAMES

Error Detection and Correction

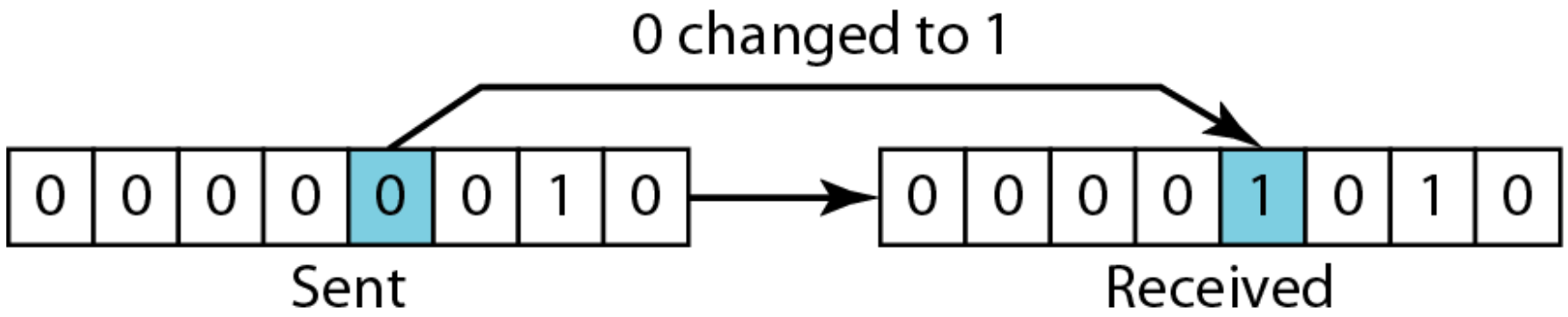
- Data can be corrupted during transmission.
- Some applications require that errors be detected and corrected.

Types of Errors

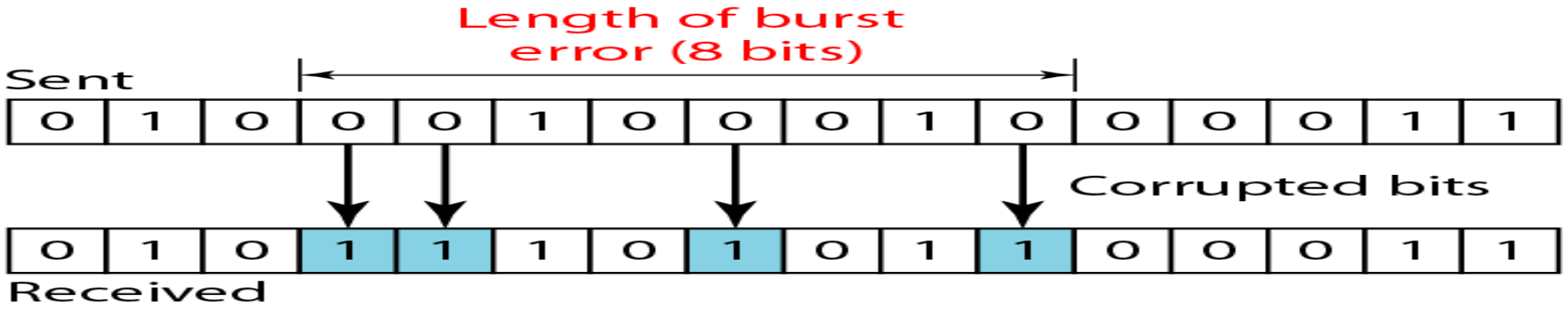
Whenever bits flow from one point to another, they are subject to unpredictable changes because of interference. This interference can change the shape of the signal. In a single-bit error, a 0 is changed to a 1 or a 1 to a O. In a burst error, multiple bits are changed. For example, a 11100 s burst of impulse noise on a transmission with a data rate of 1200 bps might change all or some of the 12 bits of information.

- ***Single-Bit Error***

The term *single-bit error* means that only 1 bit of a given data unit (such as a byte, character, or packet) is changed from 1 to 0 or from 0 to 1.



- ***Burst Error***
- The term *burst error* means that 2 or more bits in the data unit have changed from 1 to 0 or from 0 to 1.
- Figure shows the effect of a burst error on a data unit. In this case, 0100010001000011 was sent, but 0101110101100011 was received. Note that a burst error does not necessarily mean that the errors occur in consecutive bits. The length of the burst is measured from the first corrupted bit to the last corrupted bit. Some bits in between may not have been corrupted.



Redundancy

- The central concept in detecting or correcting errors is redundancy. To be able to detect or correct errors, we need to send some extra bits with our data. These redundant bits are added by the sender and removed by the receiver. Their presence allows the receiver to detect or correct corrupted bits.

Detection Versus Correction

- The correction of errors is more difficult than the detection. In error detection, we are looking only to see if any error has occurred. The answer is a simple yes or no. We are not even interested in the number of errors. A single-bit error is the same for us as a burst error.
- In error correction, we need to know the exact number of bits that are corrupted and more importantly, their location the message. The number of the errors and the size of the message are important factors. If we need to correct one single error an 8-bit data unit, we need to consider eight possible error locations; if we need to correct two errors in a data unit of the same size, we need to consider 28 possibilities.

Forward Error Correction Versus Retransmission

- There are two main methods of error correction. Forward error correction is the process in which the receiver tries to guess the message by using redundant bits. This is possible, as we see later, if the number of errors is small.
- Correction by retransmission is a technique in which the receiver detects the occurrence of an error and asks the sender to resend the message. Resending is repeated until a message arrives that the receiver believes is error-free (usually, not all errors can be detected).

Coding

- Redundancy is achieved through various coding schemes. The sender adds redundant bits through a process that creates a relationship between the redundant bits and the actual data bits. The receiver checks the relationships between the two sets of bits to detect or correct the errors. The ratio of redundant bits to the data bits and the robustness of the process are important factors in any coding scheme. Figure shows the general idea of coding.
- We can divide coding schemes into two broad categories: **block coding and convolution coding.** In this book, we concentrate on block coding; convolution coding is more complex and beyond the scope of this book.

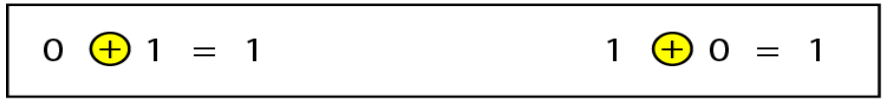
Modular Arithmetic

- In modular arithmetic, we use only a limited range of integers. We define an upper limit, called a modulus *N*. We then use only the integers 0 to *N* - I, inclusive. This is *modulo-N* arithmetic. For example, if the modulus is 12, we use only the integers 0 to 11, inclusive. An example of modulo arithmetic is our clock system. It is based on modulo-12 arithmetic, substituting the number 12 for O. In a *modulo-N* system, if a number is greater than *N*, it is divided by *N* and the remainder is the result.
- Addition and subtraction in modulo arithmetic are simple. There is no carry when you add two digits in a column. There is no carry when you subtract one digit from another in a column.

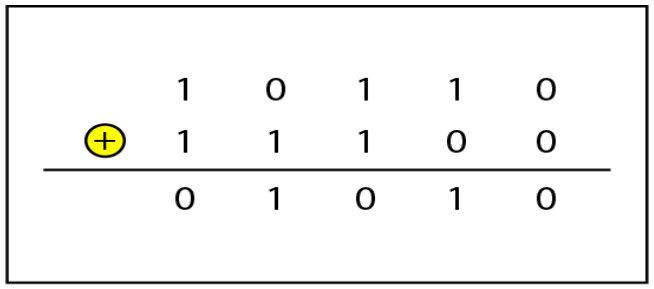
Modulo-2 Arithmetic

- Of particular interest is modulo-2 arithmetic. In this arithmetic, the modulus *N* is 2. We can use only 0 and 1. Operations in this arithmetic are very simple. The following shows how we can add or subtract 2 bits.
- Notice particularly that addition and subtraction give the same results. In this arithmetic we use the XOR (exclusive OR) operation for both addition and subtraction. The result of an XOR operation is 0 if two bits are the same; the result is I if two bits are different. Figure 10.4 shows this operation.
- Adding: 0+0=0 0+1=1 1+0=1 1+1=0
- Subtracting:0-0=0 0-1=1 1-0=1 1-1=0
- Notice particularly that addition and subtraction give the same results. In this arithmetic we use the XOR (exclusive OR) operation for both addition and subtraction. The result of an XOR operation is 0 if two bits are the same; the result is I if two bits are different. Figure 10.4 shows this operation.

a. Two bits are the same, the result is 0.



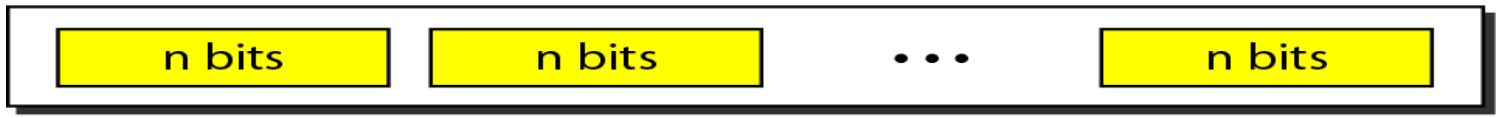
b. Two bits are different, the result is 1.



c. Result of XORing two patterns

- **BLOCK CODING**
- In block coding, we divide our message into blocks, each of *k* bits, called datawords. We add *r* redundant bits to each block to make the length $n = k + r$. The resulting *n-bit* blocks are called codewords. How the extra *r* bits is chosen or calculated is something we will discuss later. For the moment, it is important to know that we have a set of datawords, each of size *k*, and a set of codewords, each of size of *n*. With *k* bits, we can create a combination of 2^k datawords; with *n* bits, we can create a combination of 2^n codewords. Since $n > k$, the number of possible codewords is larger than the number of possible datawords. The block coding process is one-to-one; the same dataword is always encoded as the same codeword. This means that we have $2^n - 2^k$ codewords that are not used. We call these codewords invalid or illegal. Figure shows the situation.

2^k Datawords, each of k bits



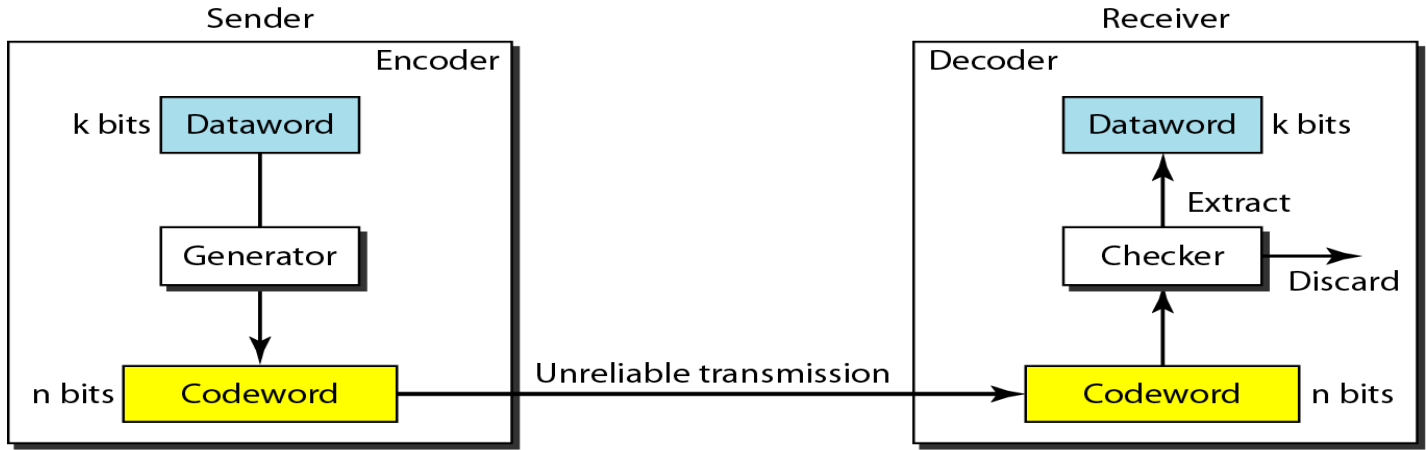
2^n Codewords, each of n bits (only 2^k of them are valid)

Error Detection

- How can errors be detected by using block coding? If the following two conditions are met, the receiver can detect a change in the original codeword.

1. The receiver has (or can find) a list of valid codewords.
2. The original codeword has changed to an invalid one.

- The sender creates codewords out of datawords by using a generator that applies the rules and procedures of encoding . Each codeword sent to the receiver may change during transmission. If the received codeword is the same as one of the valid codewords, the word is accepted; the corresponding dataword is extracted for use.
- If the received codeword is not valid, it is discarded. However, if the codeword is corrupted during transmission but the received word still matches a valid codeword, the error remains undetected. This type of coding can detect only single errors. Two or more errors may remain undetected.



- *Let us assume that k = 2 and n = 3. Table 10.1 shows the list of datawords and codewords. Later, we will see how to derive a codeword from a dataword.*
- *Assume the sender encodes the dataword 01 as 011 and sends it to the receiver. Consider the following cases:*

1. The receiver receives 011. It is a valid codeword. The receiver extracts the dataword 01 from it

2.The codeword is corrupted during transmission, and 111 is received (the leftmost bit is corrupted).This is not a valid codeword and is discarded.

3. The codeword is corrupted during transmission, and 000 is received (the right two bits are corrupted). This is a valid codeword. The receiver incorrectly extracts the dataword 00. Two correpted bits have made the error undetectable

An error-detecting code can detect only the types of errors for which it is designed;other types of errors may remain undetected.

Datawords	Codewords
00	000
01	011
10	101
11	110

Table 1

Error Correction

- As we said before, error correction is much more difficult than error detection. In error detection, the receiver needs to know only that the received codeword is invalid; in error correction the receiver needs to find (or guess) the original codeword sent. We can say that we need more redundant bits for error correction than for error detection. Figure shows the role of block coding in error correction. We can see that the idea is the same as error detection but the checker functions are much more complex.
- An error-detecting code can detect only the types of errors for which it is designed; other types of errors may remain undetected.

- ***Example:*** Let us add more redundant bits to Example to see if the receiver can correct an error without knowing what was actually sent. We add 3 redundant bits to the 2-bit dataword to make 5-bit codewords. Again, later we will show how we chose the redundant bits. For the moment let us concentrate on the error correction concept. Table shows the datawords and codewords.
- Assume the dataword is 01. The sender consults the table (or uses an algorithm) to create the codeword 01 011. The codeword is corrupted during transmission, and 01 001 is received (error in the second bit from the right). First, the receiver finds that the received codeword is not in the table. This means an error has occurred. (Detection must come before correction.) The receiver, assuming that there is only 1 bit corrupted, uses the following strategy to guess the correct dataword

Dataword	Codeword
00	00000
01	01011
10	10101
11	11110

Table 2

1. Comparing the received codeword with the first codeword in the table (01001 versus 00000), the receiver decides that the first codeword is not the one that was sent because there are two different bits.
2. By the same reasoning, the original codeword cannot be the third or fourth one in the table.
3. The original codeword must be the second one in the table because this is the only one that differs from the received codeword by 1 bit. The receiver replaces 01001 with 01011 and consults the table to find the

Hamming Distance

- One of the central concepts in coding for error control is the idea of the Hamming distance. The Hamming distance between two words (of the same size) is the number of differences between the corresponding bits. We show the Hamming distance between two words *x* and *y* as *d(x, y)*.The Hamming distance can easily be found if we apply the XOR operation on the two words and count the number of Is in the result. Note that the Hamming distance is a value greater than zero.
- *Let us find the Hamming distance between two pairs of words.*
- *1. The Hamming distance d(000, 011) is 2 because*

$000 \oplus 011$ is 011 (two 1s)

- *2. The Hamming distance d(10101, 11110) is 3 because*

$10101 \oplus 11110$ is 01011 (three 1s)

Minimum Hamming Distance

- Although the concept of the Hamming distance is the central point in dealing with error detection and correction codes, the measurement that is used for designing a code is the minimum Hamming distance. In a set of words, the minimum Hamming distance is the smallest Hamming distance between all possible pairs. We use d min to define the minimum Hamming distance in a coding scheme. To find this value, we find the Hamming distances between all words and select the smallest one.
- *Find the minimum Hamming distance of the coding scheme in Table*
- *Solution*
- *We first find all Hamming distances.*

$d(000, 011) = 2$	$d(000, 101) = 2$	$d(000, 110) = 2$	$d(011, 101) = 2$
$d(011, 110) = 2$	$d(101, 110) = 2$		

- *The d_{min} in this case is 2. C(3,2)- C(n,k) n- codeword, data size k*
- *Find the minimum Hamming distance of the coding scheme in Table 10.2.*
- *Solution*
- *We first find all the Hamming distances.*

$d(00000, 01011) = 3$	$d(00000, 10101) = 3$	$d(00000, 11110) = 4$
$d(01011, 10101) = 4$	$d(01011, 11110) = 3$	$d(10101, 11110) = 3$

- *The d_{min} in this case is 3. C(5,2)*

Hamming Distance and Error

- Before we explore the criteria for error detection or correction, let us discuss the relationship between the Hamming distance and errors occurring during transmission. When a codeword is corrupted during transmission, the Hamming distance between the sent and received codewords is the number of bits affected by the error. In other words, the Hamming distance between the received codeword and the sent codeword is the number of bits that are corrupted during transmission.
- For example, if the codeword 00000 is sent and 01101 is received, 3 bits are in error and the Hamming distance between the two is *d(OOOOO,* 01101) =3.

Minimum Hamming Distance for Error correction and Error Detection

- To guarantee the detection of up to s errors in all cases, the minimum Hamming distance in a block code must be $d_{min} = s + 1$.
- To guarantee correction of up to *t* errors in all cases, the minimum Hamming distance in a block code must be $d_{min} = 2t + 1$.

LINEAR BLOCK CODES

- Almost all block codes used today belong to a subset called linear block codes. The use of nonlinear block codes for error detection and correction is not as widespread because their structure makes theoretical analysis and implementation difficult. We therefore concentrate on linear block codes.
- For our purposes, a linear block code is a code in which the exclusive OR (addition modulo-2) of two valid codewords creates another valid codeword.
- Let us see if the two codes we defined in Table 10.1 and Table belong to the class of linear block codes.

1. The scheme in Table 1 is a linear block code because the result of XOR ing any codeword with any other codeword is a valid codeword. For example, the XOR ing of the second and third codewords creates the fourth one.

2. The scheme in Table 2 is also linear block code. We can create all four codewords by XOR ing two other codewords.

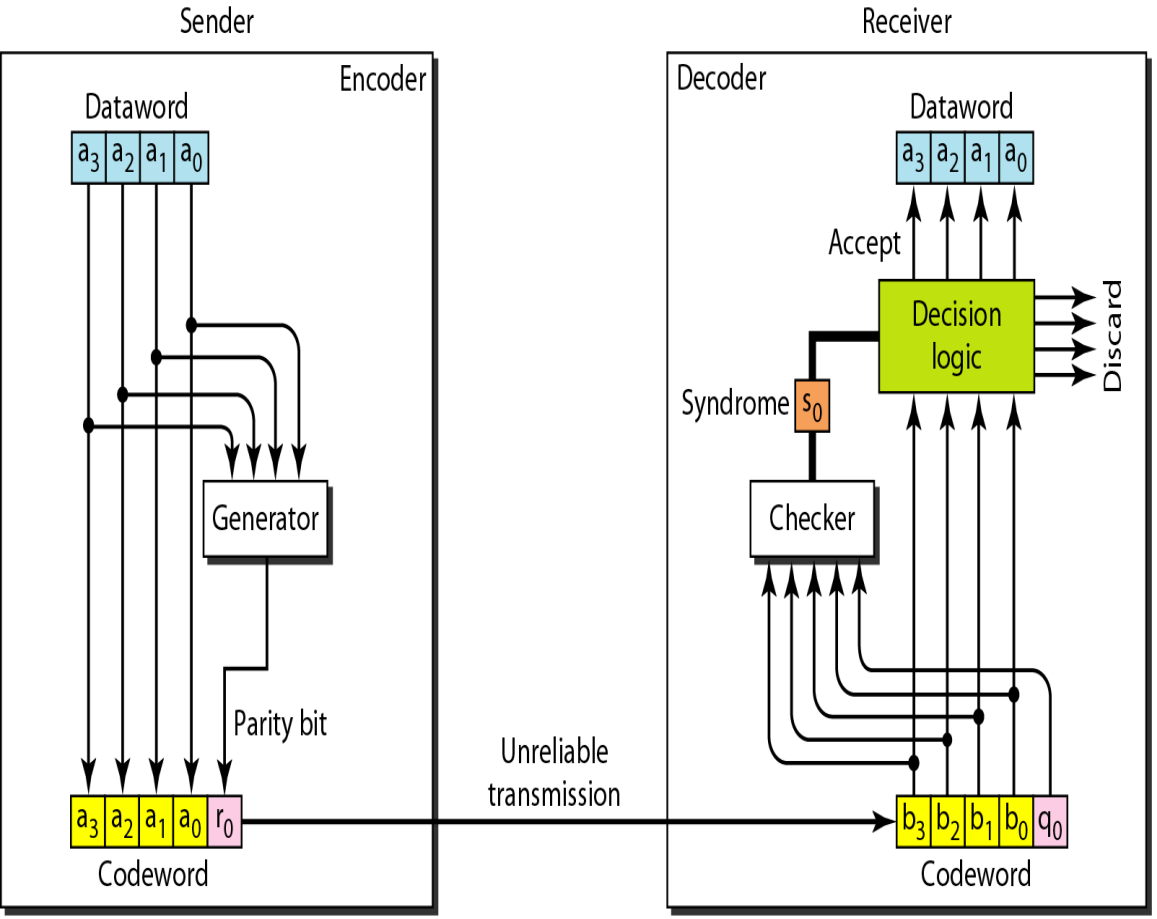
Minimum Distance for Linear Block Codes

- It is simple to find the minimum Hamming distance for a linear block code. The minimum Hamming distance is the number of Is in the nonzero valid codeword with the smallest number of Is.
- *Example* In our first code (Table 1), the numbers of Is in the nonzero codewords are 2, 2, and 2. So the minimum Hamming distance is $d_{min} = 2$.
- In our second code (Table 2), the numbers of Is in the nonzero codewords are 3, 3, and 4. So in this code we have $d_{min} = 3$.

Simple Parity-Check Code(Vertical Redundancy Check)

- Perhaps the most familiar error-detecting code is the simple parity-check code. In this code, a *k-bit* dataword is changed to an n-bit codeword where $n = k + 1$. The extra bit, called the parity bit, is selected to make the total number of Is in the codeword even. Although some implementations specify an odd number of Is, we discuss the even case. The minimum Hamming distance for this category is dmin =2, which means that the code is a single-bit error-detecting code; it cannot correct any error.
- Our first code (Table 1) is a parity-check code with $k = 2$ and $n = 3$. The code in Table 3 is also a parity-check code with $k = 4$ and $n = 5$.
- Figure shows a possible structure of an encoder (at the sender) and a decoder (at the receiver).
- The encoder uses a generator that takes a copy of a 4-bit dataword *(a0,* a1,*a2,* and *a3)* and generates a parity bit *ro*. The dataword bits and the parity bit create the 5-bitcodeword. The parity bit that is added makes the number of Is in the codeword even.

Encoder and decoder for simple parity-check code



Datawords	*Codewords*	*Datawords*	*Codewords*
0000	00000	1000	10001
0001	00011	1001	10010
0010	00101	1010	10100
0011	00110	1011	10111
0100	01001	1100	11000
0101	01010	1101	11011
0110	01100	1110	11101
0111	01111	1111	11110

Table 3

- This is normally done by adding the 4 bits of the dataword (modulo-2); the result is the parity bit. In other words,

$$r_0 = a_3 + a_2 + a_1 + a_0 \qquad \text{(modulo-2)}$$

- If the number of 1′s is even, the result is 0; if the number of 1 ′ s is odd, the result is 1. In both cases, the total number of 1s in the codeword is even.
- The sender sends the codeword which may be corrupted during transmission. The receiver receives a 5-bit word. The checker at the receiver does the same thing as the generator in the sender with one exception: The addition is done over all 5 bits.
- The result, which is called the syndrome, is just 1 bit. The syndrome is 0 when the number of 1 ′ s in the received codeword is even; otherwise, it is 1.

$$s_0 = b_3 + b_2 + b_1 + b_0 + q_0 \qquad \text{(modulo-2)}$$

- If the syndrome is 0, there is no error in the received codeword; the data portion of the received codeword is accepted as the dataword; if the syndrome is 1, the data portion of the received codeword is discarded. The dataword is not created.

Example: Let us look at some transmission scenarios. Assume the sender sends the dataword 1011. The codeword created from this dataword is 1011 1(a_3, a_2, a_1, a_0, r_0) which is sent to the receiver. We examine five cases:

1. No error occurs; the received codeword is 1011 1. The syndrome is O. The dataword 1011 is created.

2. One single-bit error changes *a1* The received codeword is 1001 1. The syndrome is 1. No dataword is created.

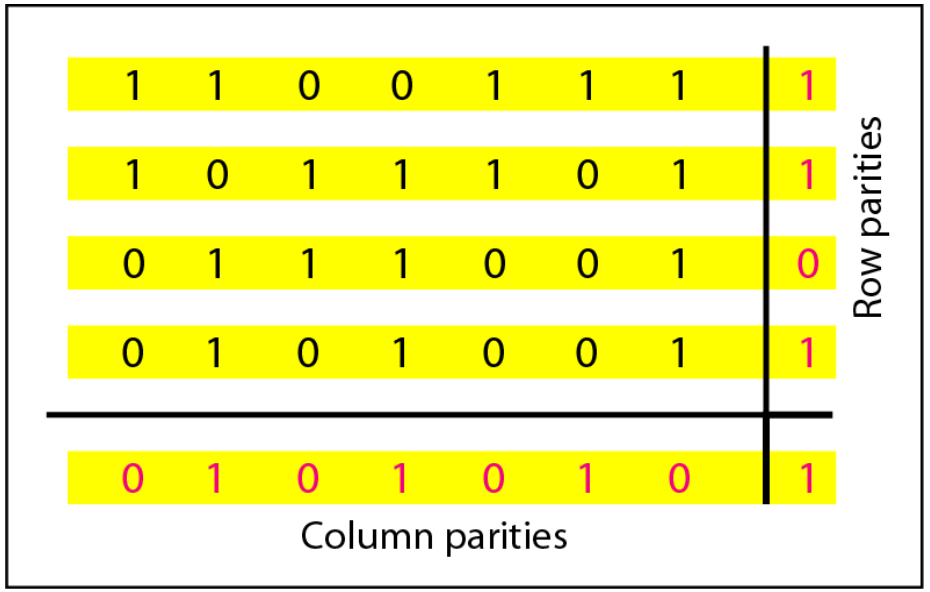
3. One single-bit error changes *ro* The received codeword is 10110. The syndrome is 1. No dataword is created. Note that although none of the dataword bits are corrupted, no dataword is created because the code is not sophisticated enough to show the position of the corrupted bit.

4. An error changes *ro* and a second error changes *a3* The received codeword is 0011 0. The syndrome is O. The dataword 0011 is created at the receiver. Note that here the dataword is wrongly created due to the syndrome value. The simple parity-check decoder cannot detect an even number of errors. The errors cancel each other out and give the syndrome a value of O.

5. Three bits-a3, *a2,* and a1-are changed by errors. The received codeword is 01011. The syndrome is 1. The dataword is not created. This shows that the simple parity check, guaranteed to detect one single error, can also find any odd number of errors.

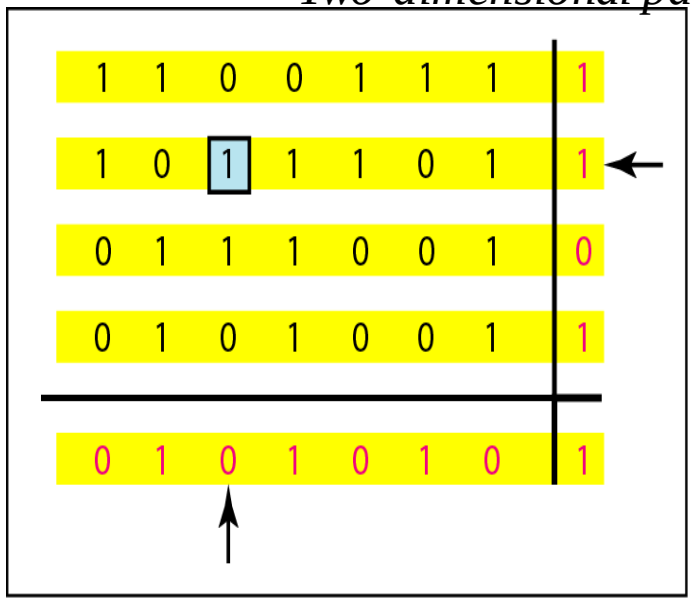
- A simple parity-check code can detect an odd number of errors. A better approach is the two-dimensional parity check. In this method, the dataword is organized in a table (rows and columns). In Figure, the data to be sent, five 7-bit bytes, are put in separate rows. For each row and each column, 1 parity-check bit is calculated. The whole table is then sent to the receiver, which finds the syndrome for each row and each column. As Figure below shows, the two-dimensional parity check can detect up to three errors that occur anywhere in the table (arrows point to the locations of the created nonzero syndromes). However, errors affecting 4 bits may not be detected.

Two-dimensional parity-check code

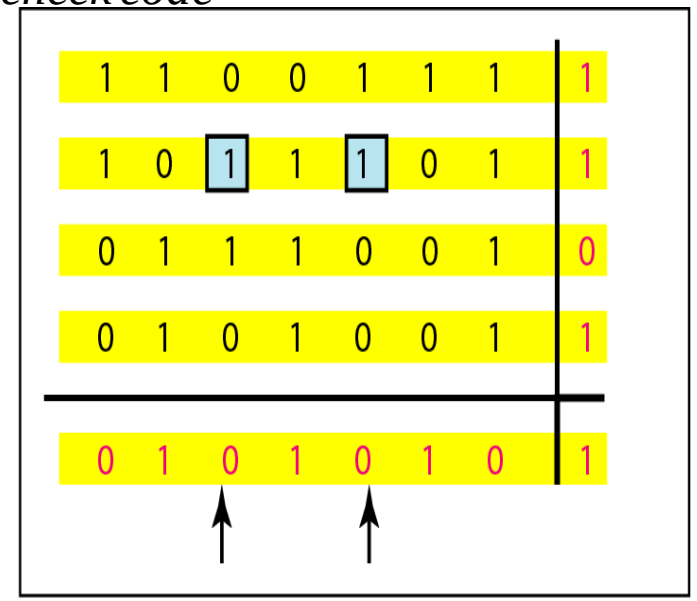


a. Design of row and column parities

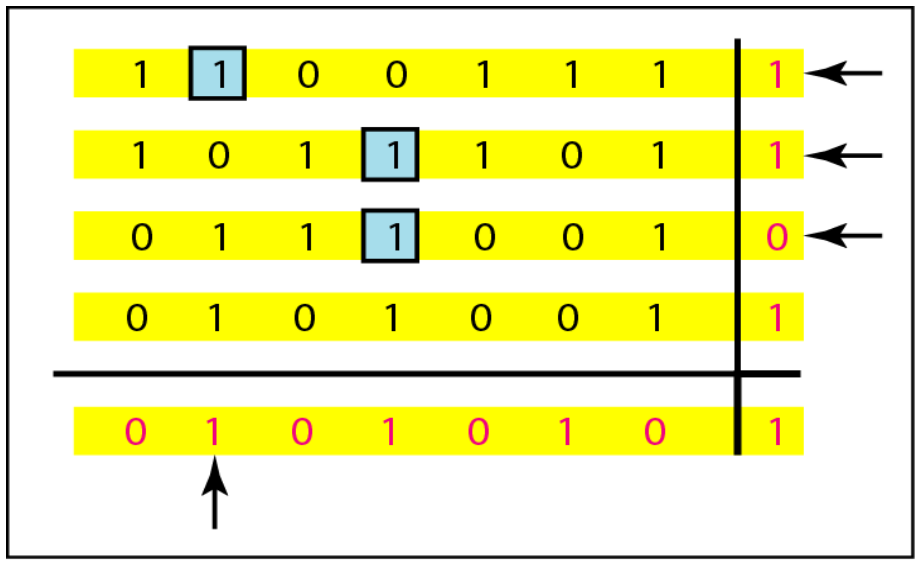
Two-dimensional parity-check code



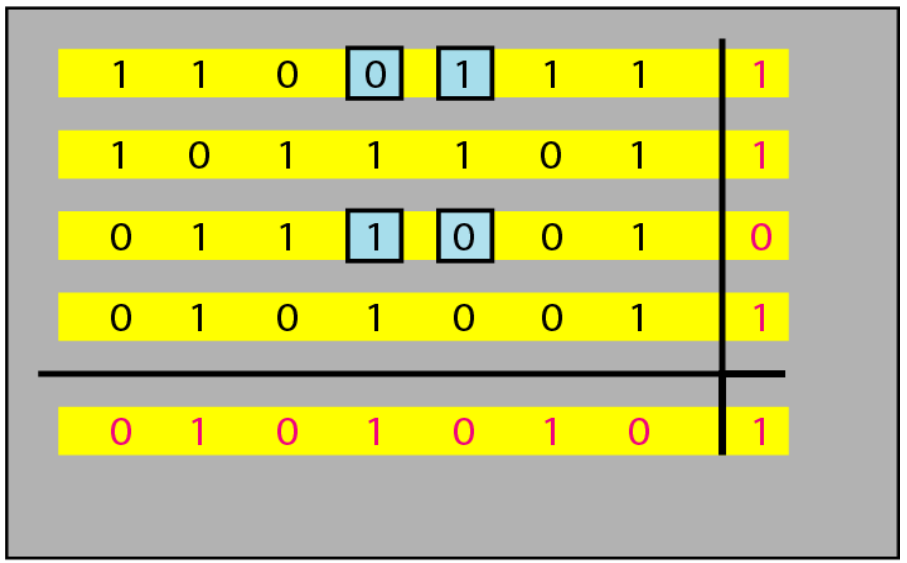
b. One error affects two parities



c. Two errors affect two parities



d. Three errors affect four parities



e. Four errors cannot be detected

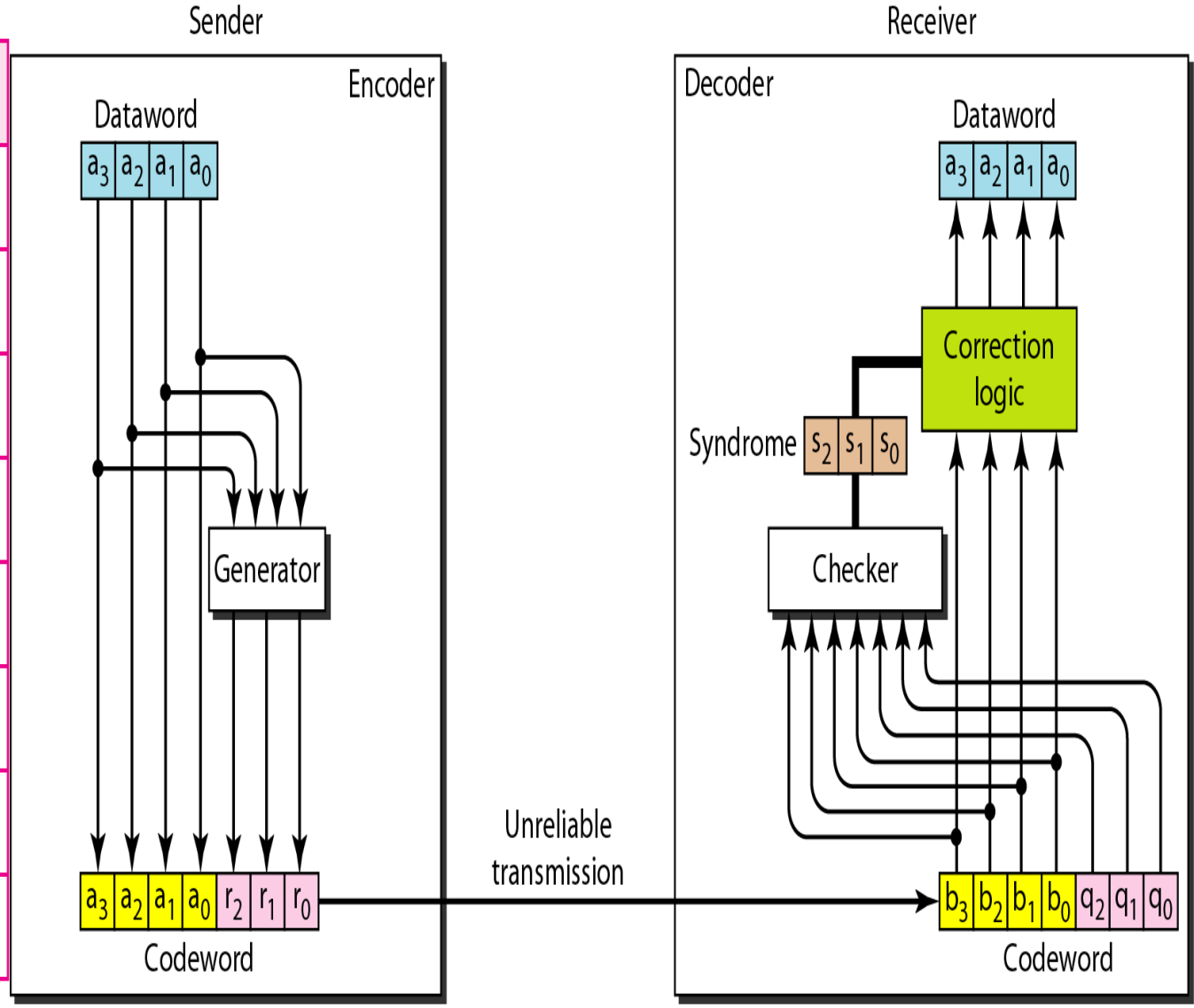
Hamming Codes

- Now let us discuss a category of error-correcting codes called Hamming codes. These codes were originally designed with *dmin* = 3, which means that they can detect up to two errors or correct one single error. Although there are some Hamming codes that can correct more than one error, our discussion focuses on the single-bit error-correcting code. First let us find the relationship between *n* and *k* in a Hamming code. We need to choose an integer $m >= 3$. The values of n and *k* are then calculated from mas $n = 2^m - 1$ and $k = n - m$. The number of check bits $r = m$.
- *{n=k+r (r must be able to indicate atleast k+r+1 different states of these one state means no error and k+r state indicate location of error in each of k+r states) ie, k+r+1 must be dicoverabe by r bits and r bits can indicate* 2^r different states $2^r > k+r+1$ *}*
- All Hamming codes discussed in this book have dmin = 3. The relationship between *m* and *n* in these codes is $n = 2^m - 1$.
- For example, if *m* =3, then *n* = 7 and *k*= 4. This is a Hamming code *C(7,* 4) with dmin =3.
- Table shows the datawords and codewords for this code.

- Hamming code C(7,4)

Datawords	Codewords	Datawords	Codewords
0000	0000000	1000	1000110
0001	0001101	1001	1001011
0010	0010111	1010	1010001
0011	0011010	1011	1011100
0100	0100011	1100	1100101
0101	0101110	1101	1101000
0110	0110100	1110	1110010
0111	0111001	1111	1111111

Figure ***The structure of the encoder and decoder for a Hamming code***

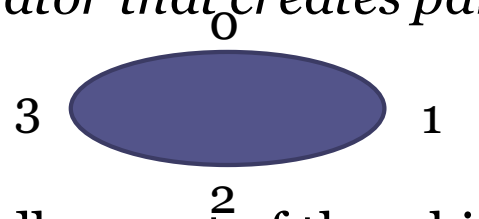


- *A copy of 4 bit(k) codeword is fed into the generator that creates parity check r0,r1,r2*

  r0=a2+a1+a0 modulo-2

  r1=a3 + a2+ a1 modulo-2

  r2=a1+a0+a3 modulo-2



- In other words, each of the parity-check bits handles 3 out of the 4 bits of the dataword. The total number of 1s in each 4-bit combination (3 dataword bits and 1 parity bit) must be even. any three equations that involve 3 of the 4 bits in the dataword and create independent equations(a combination of two cannot create the third) are valid.
- The checker in the decoder creates a 3-bit syndrome(S_2 S_1 S_0) in which each bit is the parity check for 4 out of the 7 bits in the received codeword:
- *s0=b2+b1+b0 +q0* modulo-2
- *s1=b3 + b2+* b1 +q1 modulo-2
- *s2=b1+b0+b3 +q2* modulo-2
- The equations used by the checker are the same as those used by the generator with the parity-check bits added to the right-hand side of the equation. The 3-bit syndrome creates eight different bit patterns (000 to 111) that can represent eight different conditions. These conditions define a lack of error or an error in 1 of the 7 bits of the received codeword, as shown in Table

	S_2 S_1 S_0	S_2 S_1 S_0						
Syndrome	000	001	010	011	100	101	110	111
Error	None	q_0	q_1	b_2	q_2	b_0	b_3	b_1

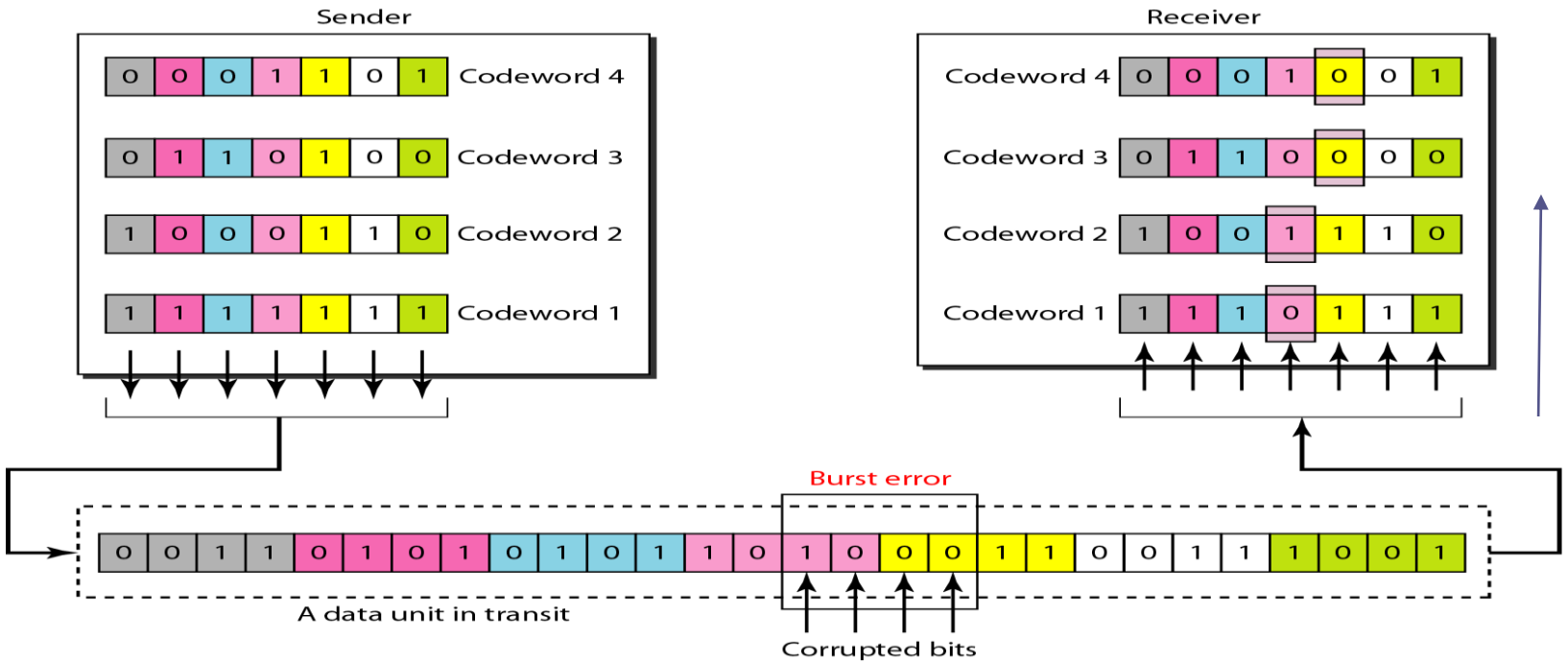
	$S_2 S_1 S_0$	$S_2 S_1 S_0$						
Syndrome	000	001	010	011	100	101	110	111
Error	None	q_0	q_1	b_2	q_2	b_0	b_3	b_1

- Note that the generator is not concerned with the four cases shaded in Table because there is either no error or an error in the parity bit. In the other four cases, 1 of the bits must be flipped (changed from 0 to 1 or 1 to 0) to find the correct dataword.
- The syndrome values in Table are based on the syndrome bit calculations. For example, if *qo* is in error, *So* is the only bit affected; the syndrome, therefore, is 001. If *b2* is in error, *So* and *S1* are the bits affected; the syndrome, therefore is O11. Similarly, if b1 is in error, all 3 syndrome bits are affected and the syndrome is 111.
- There are two points we need to emphasize here. First, if two errors occur during transmission, the created dataword might not be the right one. Second, if we want to use the above code for error detection, we need a different design.
- *Let us trace the path of three datawords from the sender to the destination:*

$a_3 a_2 a_1 a_0$ $b_3 b_2 b_1 b_0$ $q_2 q_1 q_0$

1. *The dataword 0100 becomes the codeword 0100 011.The codeword 0100 011 is received. The syndrome is 000, the final dataword is 0100.*
2. *The dataword 0111 becomes the codeword 0111 001. The codeword 0011 001 is received. The syndrome is 011. After flipping b_2 (changing the 1 to 0), the final dataword is 0111.*
3. *The dataword 1101 becomes the codeword 1101 000. The codeword 0001 000 is received* (two errors)*The syndrome is 101. After flipping b_0, we get 0000, the wrong dataword. This shows that our code cannot correct two errors.*

- *Performance*
- A Hamming code can only correct a single error or detect a double error. However, there is a way to make it detect a burst error, as shown in Figure. The key is to split a burst error between several codewords, one error for each codeword. In data communications, we normally send a packet or a frame of data. To make the Hamming code respond to a burst error of size *N*, we need to make *N* codewords out of our frame. Then, instead of sending one codeword at a time, we arrange the codewords in a table and send the bits in the table a column at a time. In Figure, the bits are sent column by column (from the left). In each column, the bits are sent from the bottom to the top. In this way, a frame is made out of the four codewords and sent to the receiver.



CYCLIC CODES

- Cyclic codes are special linear block codes with one extra property. In a cyclic code, if a codeword is cyclically shifted (rotated), the result is another codeword.

$a_6\ a_5\ a_4 a_3\ a_2 a_1\ a_0$ $b_6\ b_5 b_4\ b_3\ b_2 b_1\ b_0$

- For ex,if 1011000 is a codeword and we cyclically left-shift, then 0110001 is also a codeword.In this case, if we call the bits in the first word *ao* to *a6'* and the bits in the second word *bo* to *b6,* we can shift the bits by using the following:

$$b_1 = a_0 \quad b_2 = a_1 \quad b_3 = a_2 \quad b_4 = a_3 \quad b_5 = a_4 \quad b_6 = a_5 \quad b_0 = a_6$$

- In the rightmost equation, the last bit of the first word is wrapped around and becomes the first bit of the second word.

Cyclic Redundancy Check

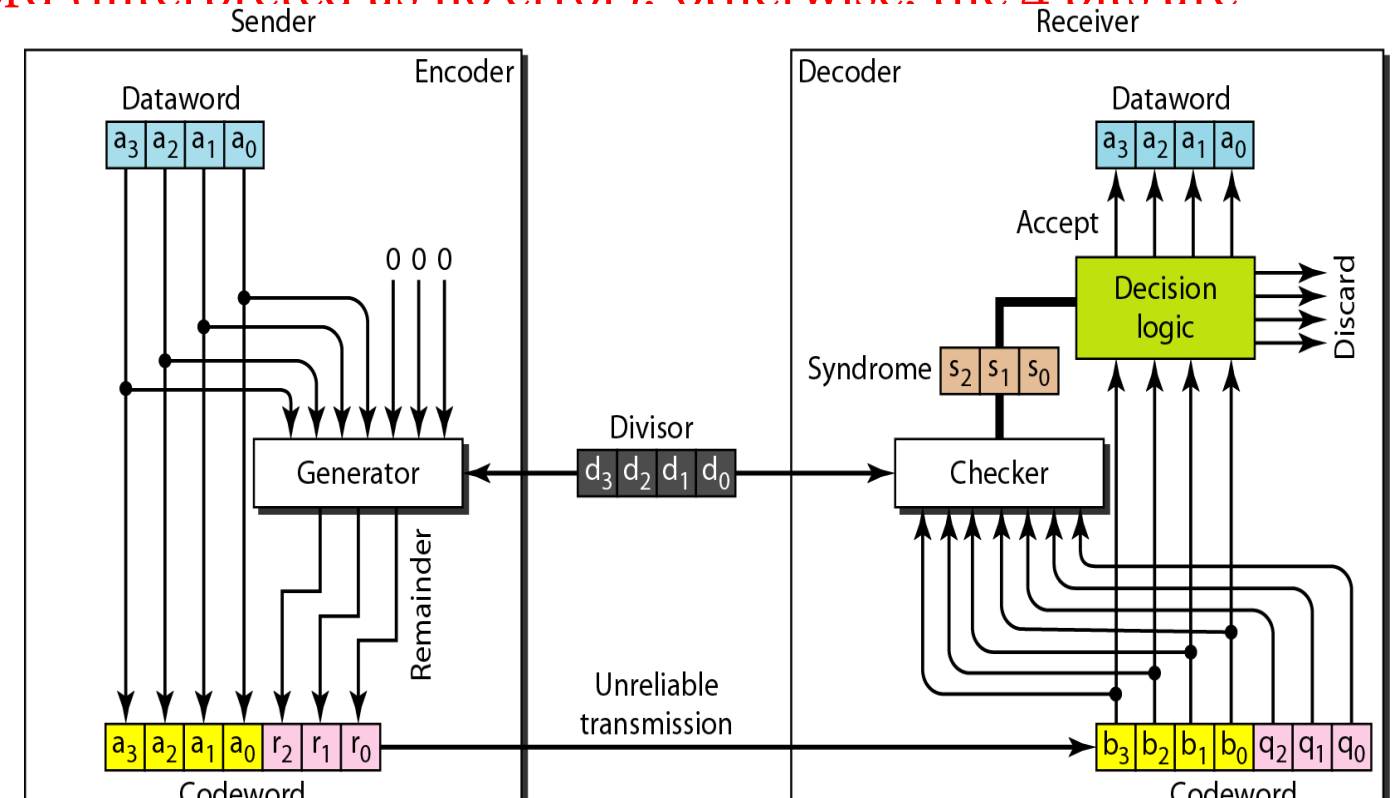
- We can create cyclic codes to correct errors. However, the theoretical backgroundrequired is beyond the scope of this book. In this section, we simply discuss a category of cyclic codes called the cyclic redundancy check (CRC) that is used in networks such as LANs and WANs.

Table 6 shows an example of a CRC code. We can see both the linear and cyclic properties of this code.

In the encoder, the dataword has *k* bits (4 here); the codeword has *n* bits (7 here). The size of the dataword is augmented by adding *n* - *k* (3 here) O's to the right-hand side of the word. The n-bit result is fed into the generator. The generator uses a divisor of size *n* - *k* + 1(4 here), predefined and agreed upon.

The generator divides the augmented dataword by the divisor (modulo-2 division). The quotient of the division is discarded; the remainder *(r2r1 ro)* is appended to the dataword to create the codeword. The decoder receives the possibly corrupted codeword. A copy of all *n* bits is fed to the checker which is a replica of the generator. The remainder produced by the checker is a syndrome of *n* - *k* (3 here) bits, which is fed to the decision logic analyzer. The analyzer has a simple function. If the syndrome bits are all 0's , the 4 leftmost bits of the codeword are accepted as the dataword (interpreted as no error); otherwise, the 4 bits are discarded (error).

Dataword	*Codeword*	*Dataword*	*Codeword*
0000	0000000	1000	1000101
0001	0001011	1001	1001110
0010	0010110	1010	1010011
0011	0011101	1011	1011000
0100	0100111	1100	1100010
0101	0101100	1101	1101001
0110	0110001	1110	1110100
0111	0111010	1111	1111111



Encoder

- Let us take a closer look at the encoder. The encoder takes the dataword and augments it with *n* - *k* number of as. It then divides the augmented dataword by the divisor, as shown in Figure The process of modulo-2 binary division is the same as the familiar division process we use for decimal numbers. In this case addition and subtraction are the same. We use the XOR operation to do both.
- As in decimal division, the process is done step by step. In each step, a copy of the divisor is XORed with the 4 bits of the dividend. The result of the XOR operation (remainder) is 3 bits (in this case), which is used for the next step after 1 extra bit is pulled down to make it 4 bits long. There is one important point we need to remember in this type of division.
- If the leftmost bit of the dividend (or the part used in each step)is 0, the step cannot use the regular divisor; we need to use an all-Os divisor. Each bit of the divisor is subtracted from the corresponding bit of the divided without disturbing the next bit. The leading 0 of the remainder is dropped off. The next unused bit from the dividend is then pulled down to make the number of bits in the remainder equal to the no of bits in the divisor. When there are no bits left to pull down, we have a result. The 3-bit remainder forms the check bits *(r2'r1'* and *ro)*. They are appended to the dataword to create the codeword.

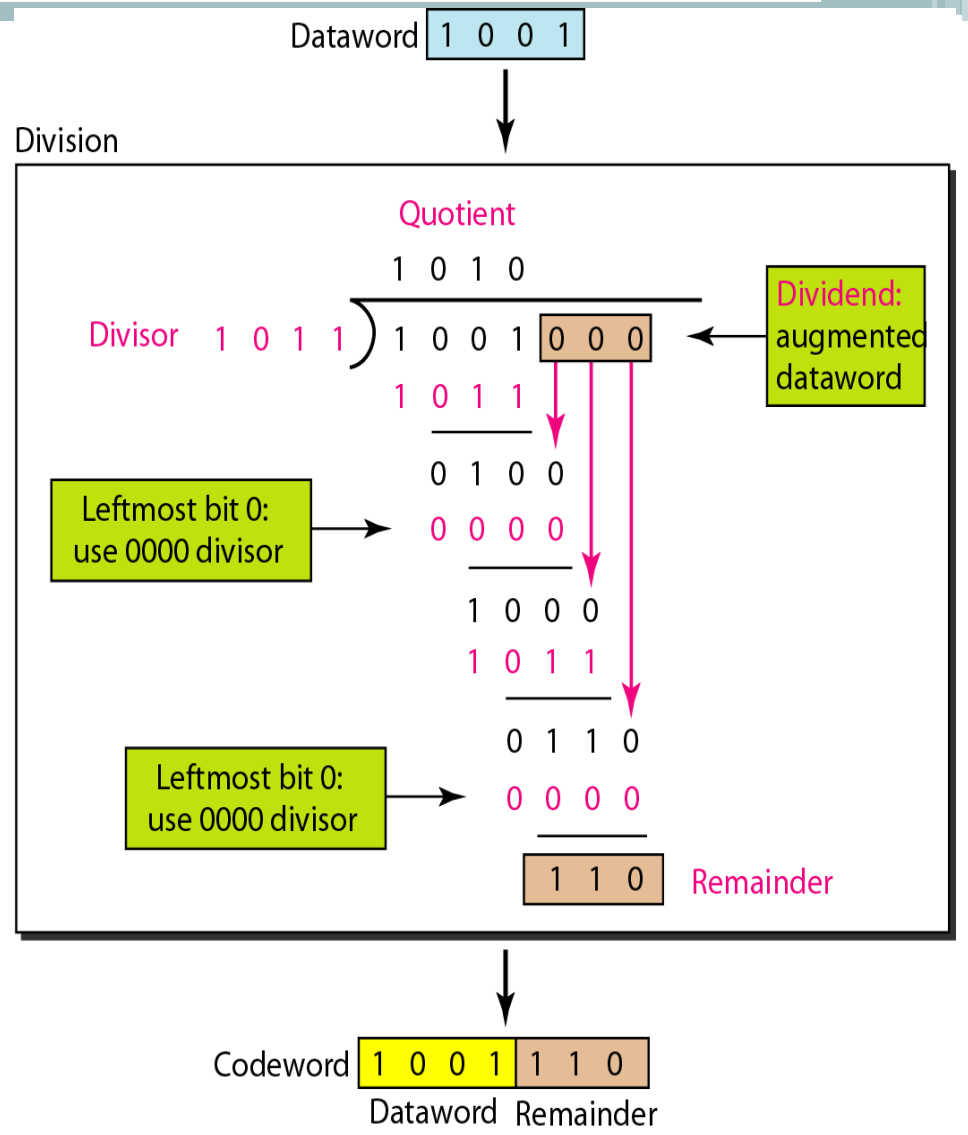
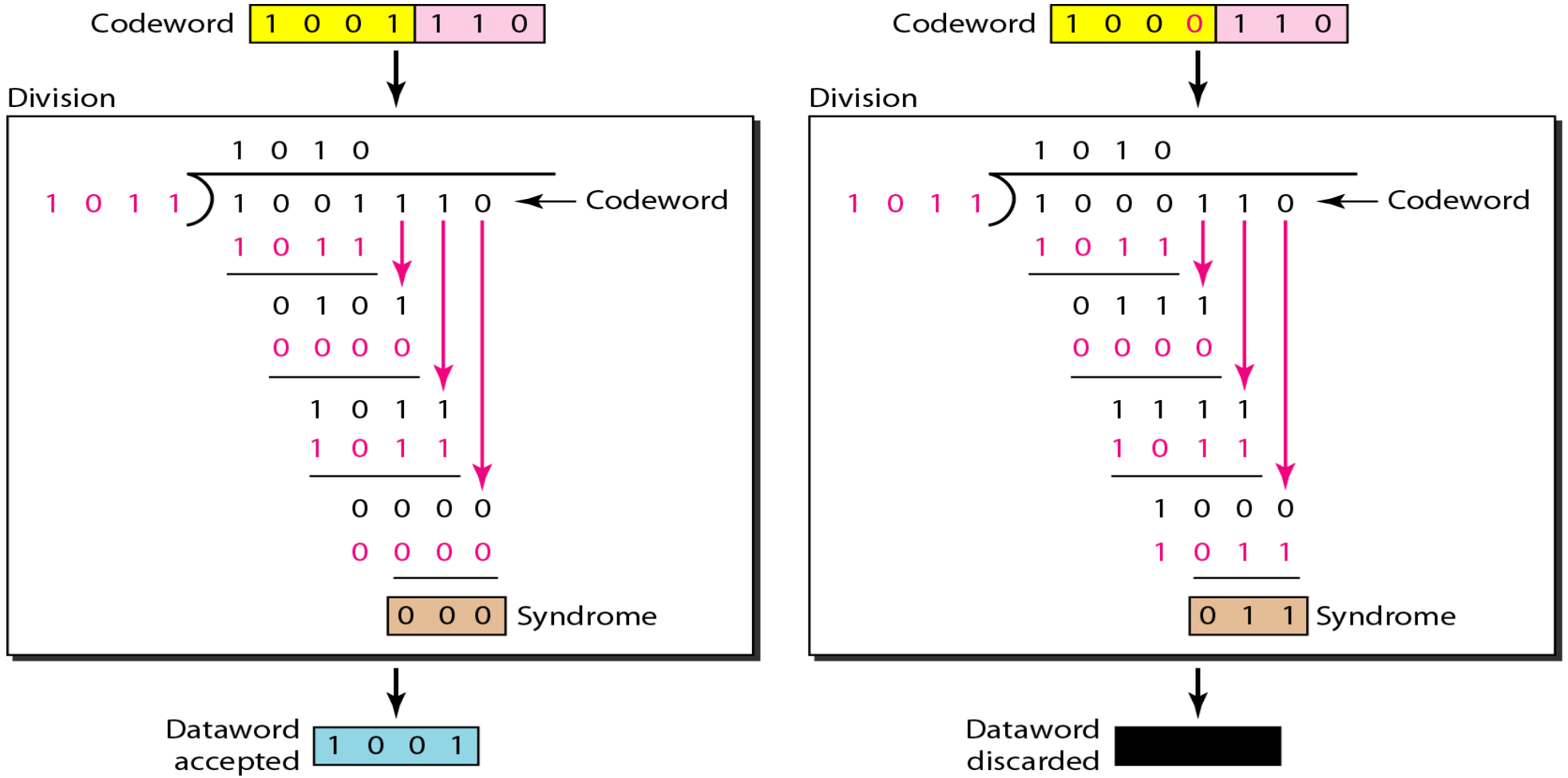


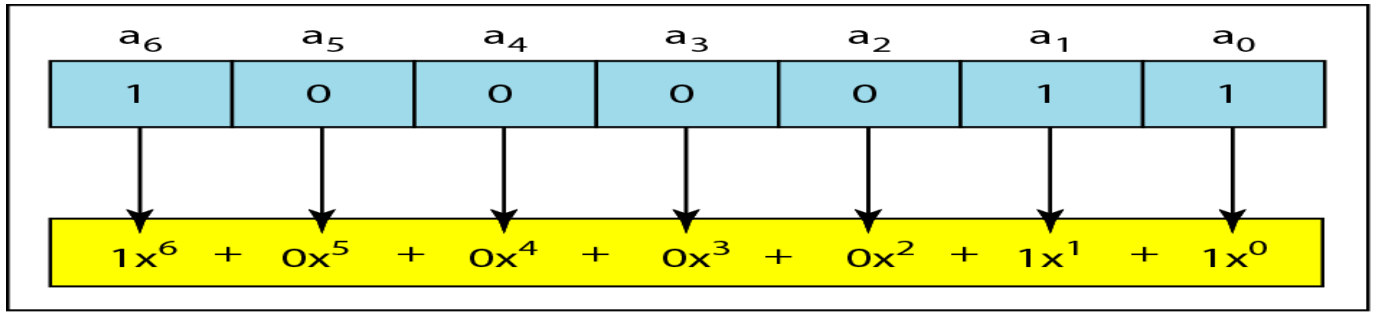
Figure ***Division in CRC encoder***

Decoder

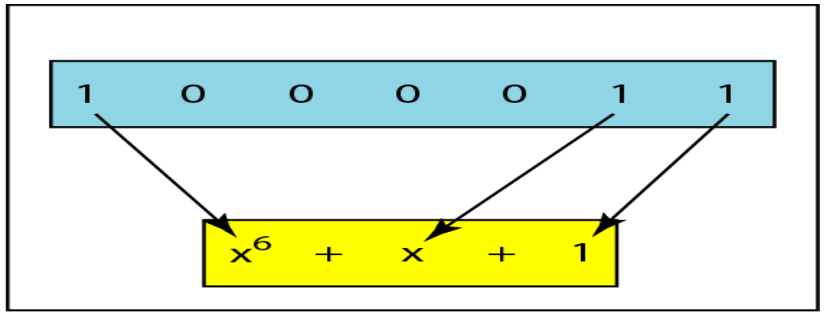
- The codeword can change during transmission. The decoder does the same division process as the encoder. The remainder of the division is the syndrome. If the syndrome is all Os, there is no error; the dataword is separated from the received codeword and accepted. Otherwise, everything is discarded. Figure 10.16 shows two cases: The lefthand figure shows the value of syndrome when no error has occurred; the syndrome is 000. The right-hand part of the figure shows the case in which there is one single error.
- The syndrome is not all Os (it is 011).

Division in CRC decoder

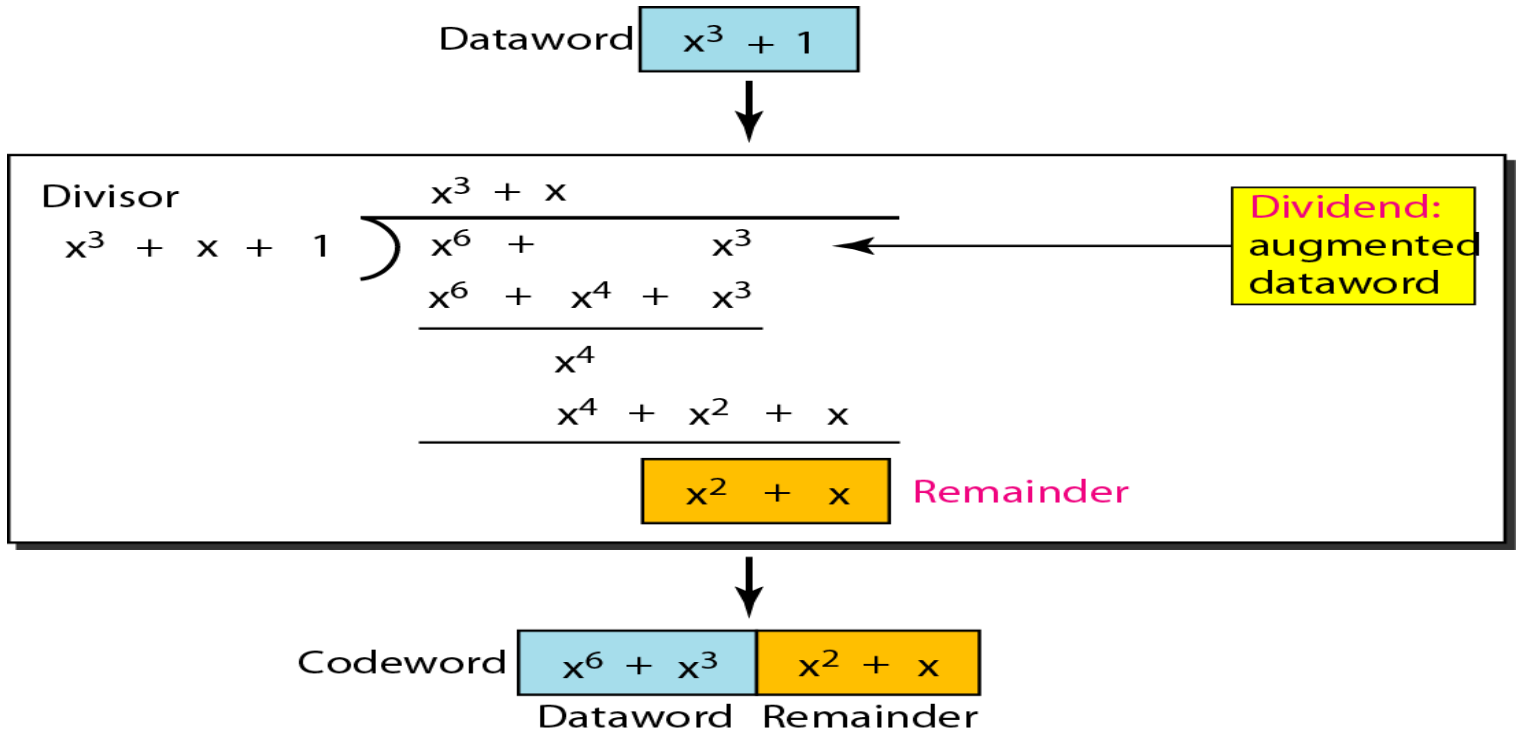


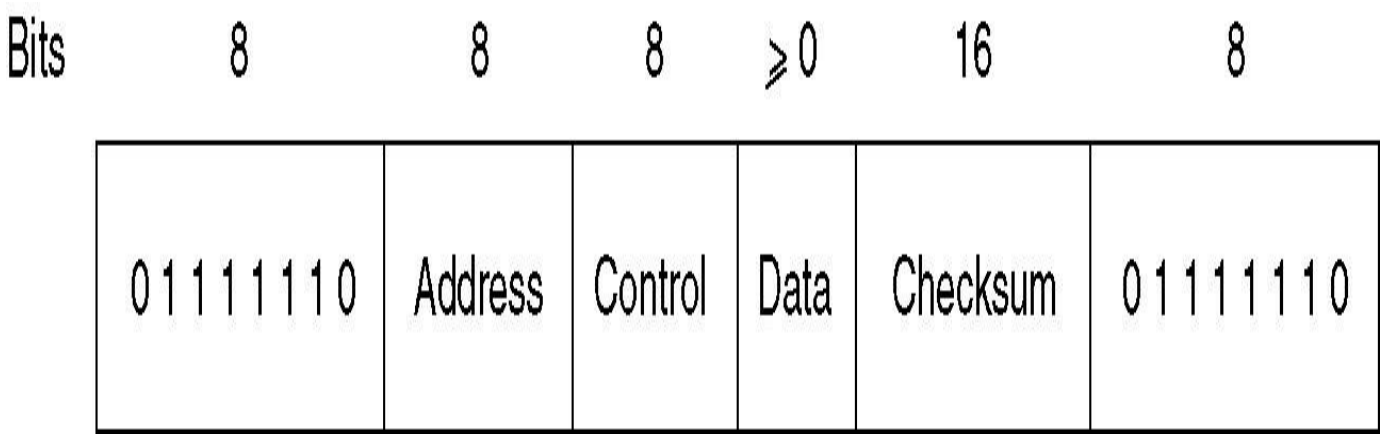


a. Binary pattern and polynomial



b. Short form




Bits
8
8
8
≥0
16
8
01111110
Address
Control
Data
Checksum
01111110