- Data Link Layer

Chapter 10

Error Detection and Correction

Data Link Layer

- ***Between network layer and physical layer in Internet Model***
- ***Receives services from physical layer***
- ***Provides services to network layer***

Responsible:

- **Carry a packet from one hop (computer or router) to next**
- **Local responsibility (as opposed to network layer) b/w 2 hops**
- **Make sure that Packet arrives safe and sound**
- **Error Detection and Correction**
- **Flow Control: flow of data not too much to the next hop**
- **Medium Access Control:shared medium, who has the right to send**

Data Link Layer

Duties:

- ***Packetizing***
- ***Addressing***
- ***Error Control***
- ***Flow Control***
- ***Medium Access Control***

Data(one or more bits) can be corrupted during transmission.

Some applications

- Need reliable communication
- require that errors be detected and corrected.

Error Detection and Correction

Let us first discuss some issues related, directly or indirectly, to error detection and correction.

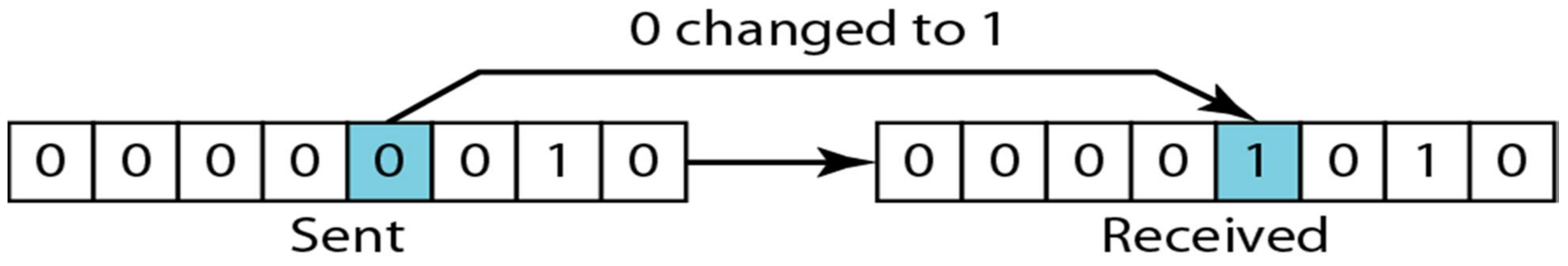
Topics discussed in this section:

Types of Errors
Redundancy
Detection Versus Correction
Forward Error Correction Versus Retransmission
Coding

Types of Errors

Single-bit Error:

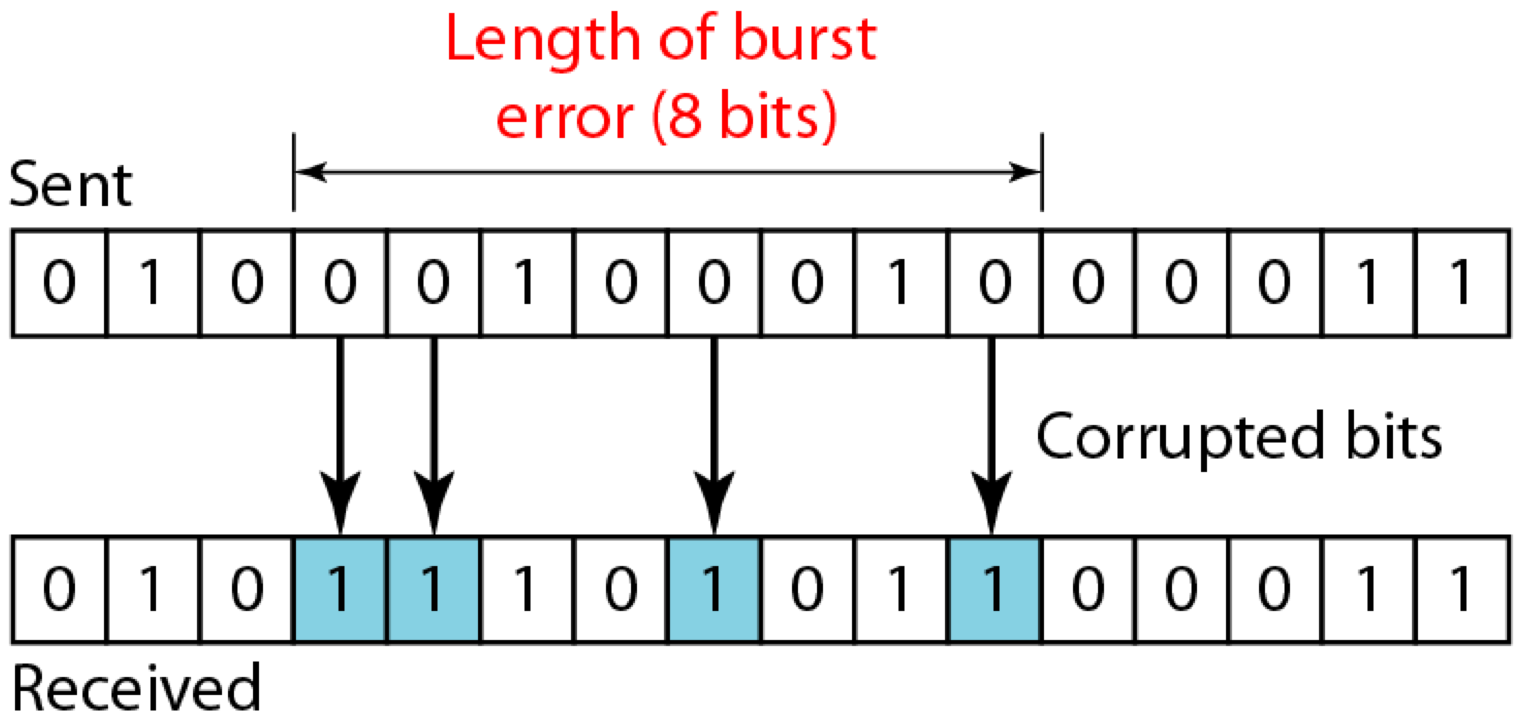
- **In a single-bit error, only 1 bit in the data unit has changed.**



Burst Error:

- **In a burst error, 2 or more bits in the data unit have changed (from 1 to 0 or 0 to 1)**

Figure 10.2 *Burst error of length 8*

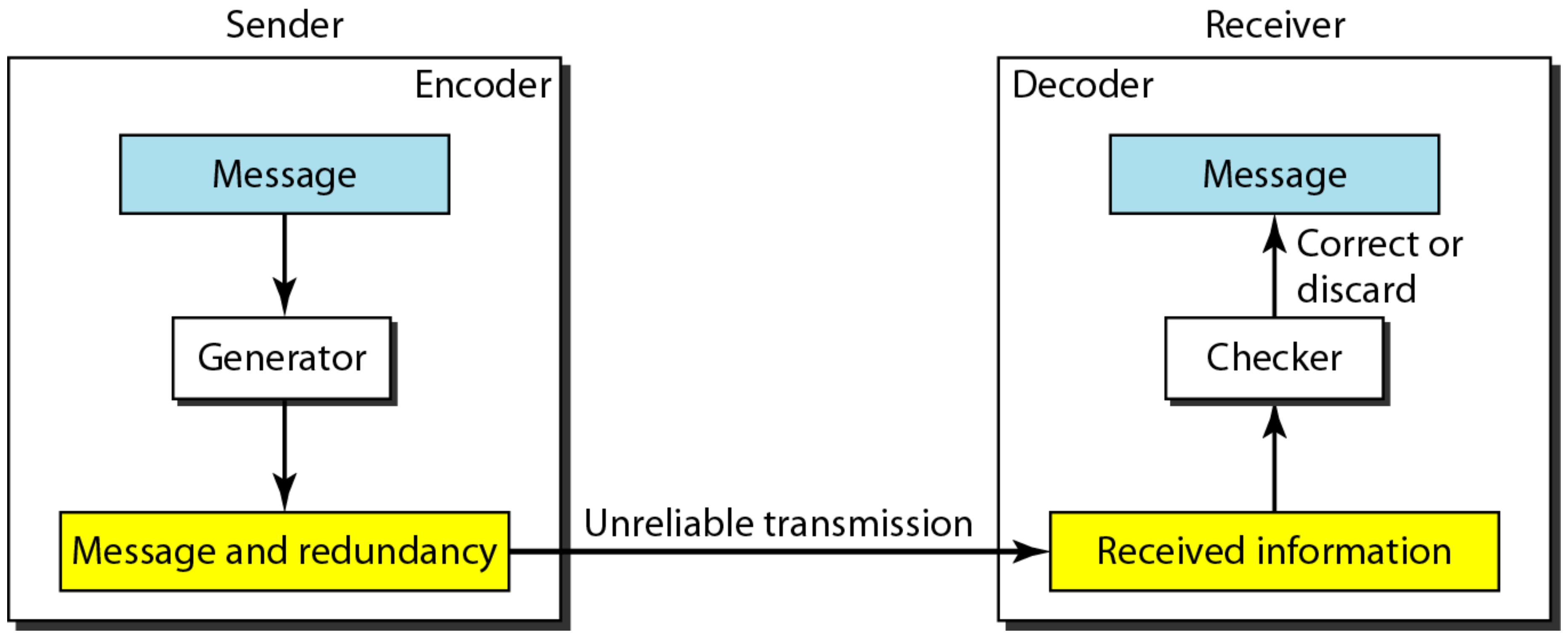


Error Detection

- One mechanism: send every data unit twice
- Receiver: bit-for-bit comparison b/w 2 data units
    - Double trans time, expensive comparison

- To detect or correct errors, we need to send extra (redundant) bits with data

Redundancy in the structure of encoder and decoder



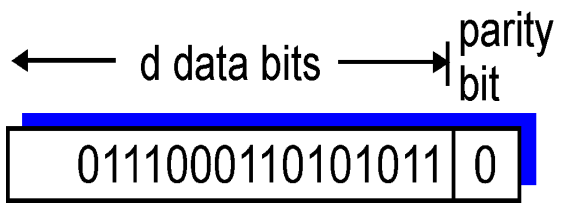
Detection Methods

- Parity Check
  - Simple
  - Two-dimensional
- Cyclic Redundancy Check(CRC)
- Checksum

Parity Checking:

Single Bit Parity:

Detect single bit errors

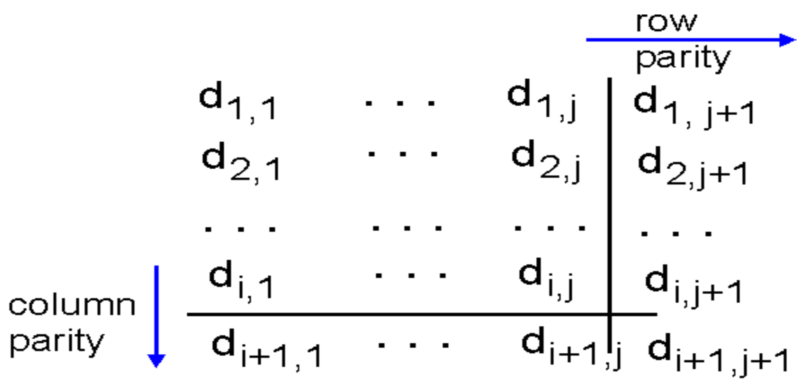


Based on addition. A redundant bit, parity bit added to every data unit so that total number of 1s in the unit becomes even(odd)

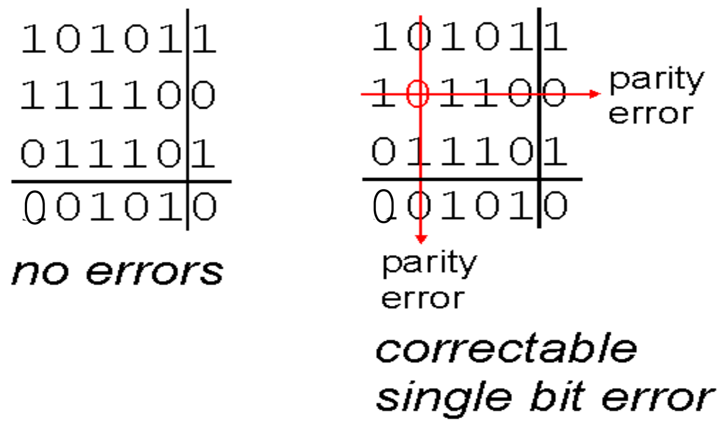
Two Dimensional Bit Parity:

A redundant row of bits is added to the whole block.

Detect *and correct* single bit errors



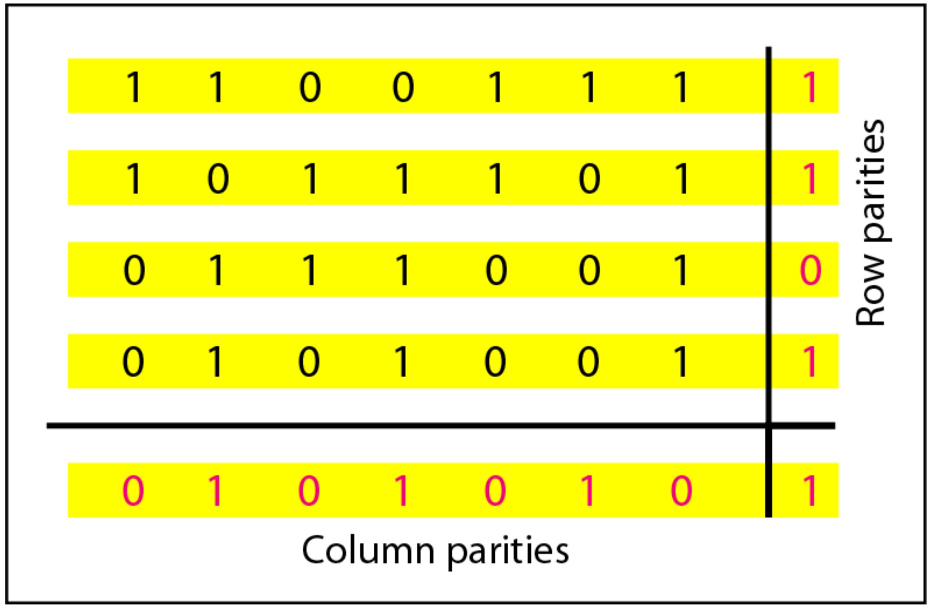
row parity
column parity
$d_{1,1}$ . . . $d_{1,j}$ $d_{1,j+1}$
$d_{2,1}$ . . . $d_{2,j}$ $d_{2,j+1}$
. . . .
$d_{i,1}$ . . . $d_{i,j}$ $d_{i,j+1}$
$d_{i+1,1}$ . . . $d_{i+1,j}$ $d_{i+1,j+1}$



Note

A simple parity-check code can detect an odd number of errors.

Figure 10.11 *Two-dimensional parity-check code*

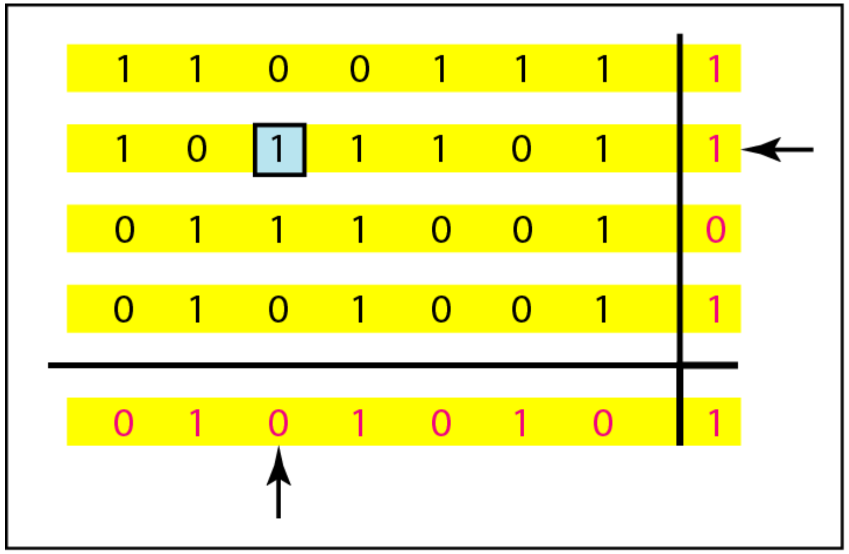


							Row parities
1	1	0	0	1	1	1	1
1	0	1	1	1	0	1	1
0	1	1	1	0	0	1	0
0	1	0	1	0	0	1	1
0	1	0	1	0	1	0	1

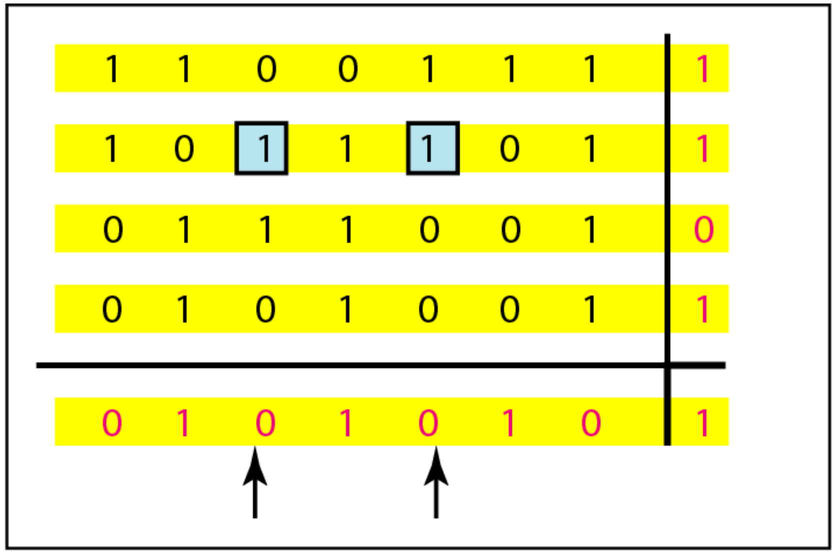
Column parities

a. Design of row and column parities

Figure 10.11 *Two-dimensional parity-check code*

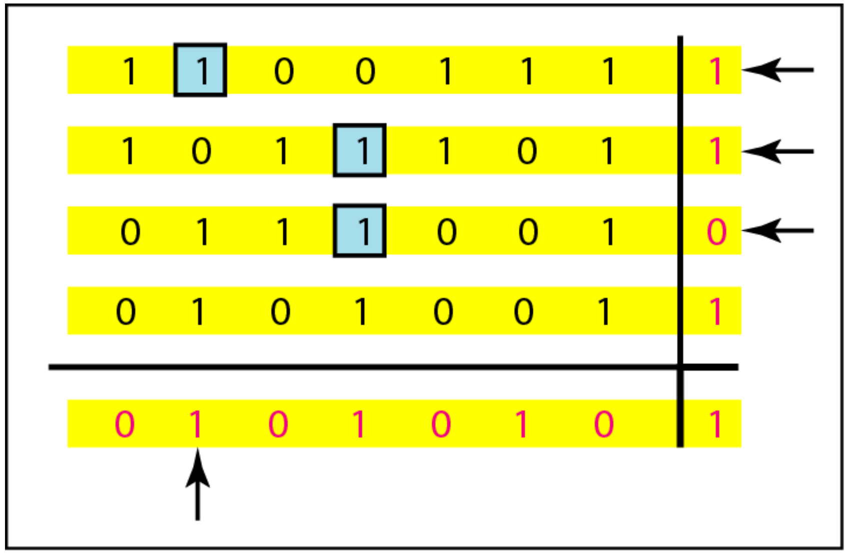


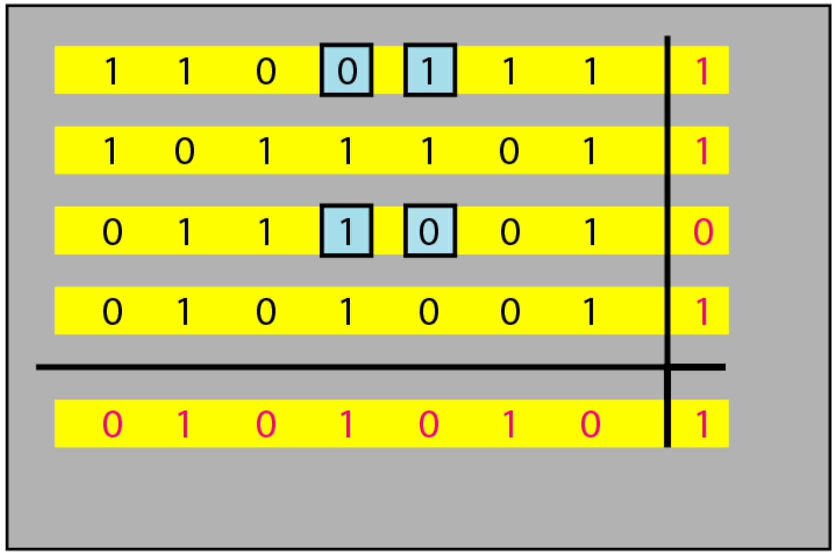
b. One error affects two parities



c. Two errors affect two parities

Figure 10.11 *Two-dimensional parity-check code*


d. Three errors affect four parities


e. Four errors cannot be detected

Cyclic Redundancy Check(CRC)

- Based on binary division
- view data bits, D, as a binary number
- choose r+1 bit pattern (divisor, generator), G
- goal: choose r CRC bits, R, such that
  - <D,R> exactly divisible by G (modulo 2)
  - receiver knows G, divides <D,R> by G. If non-zero remainder: error detected!
  - can detect all burst errors less than r+1 bits
- widely used in practice (ATM, HDCL)

Cyclic Redundancy Check(CRC)

- Sender: D: Data bits, n 0s
- <D, 0…0> → Divisor G(n+1) bits → CRC (Remainder n bits)

- Send <D, CRC> to Receiver
- Receiver:
- <D, CRC> → Divisor G(n+1) bits → Remainder; if zero accept; If non-zero remainder: error detected!
    - can detect all burst errors less than r+1 bits

CRC Example (Generator)

```
Divisor 1101:   100100000   (data+0s)
                1101
```

```
Remainder: 001
Sent: 100100001 (Data+CRC)
```

CRC Example (Checker)

```
Divisor 1101:   100100001   (data+CRC)
                1101
```

```
Result: 000
Result is all 0s, data is accepted
```

Figure 10.14 *CRC encoder and decoder*

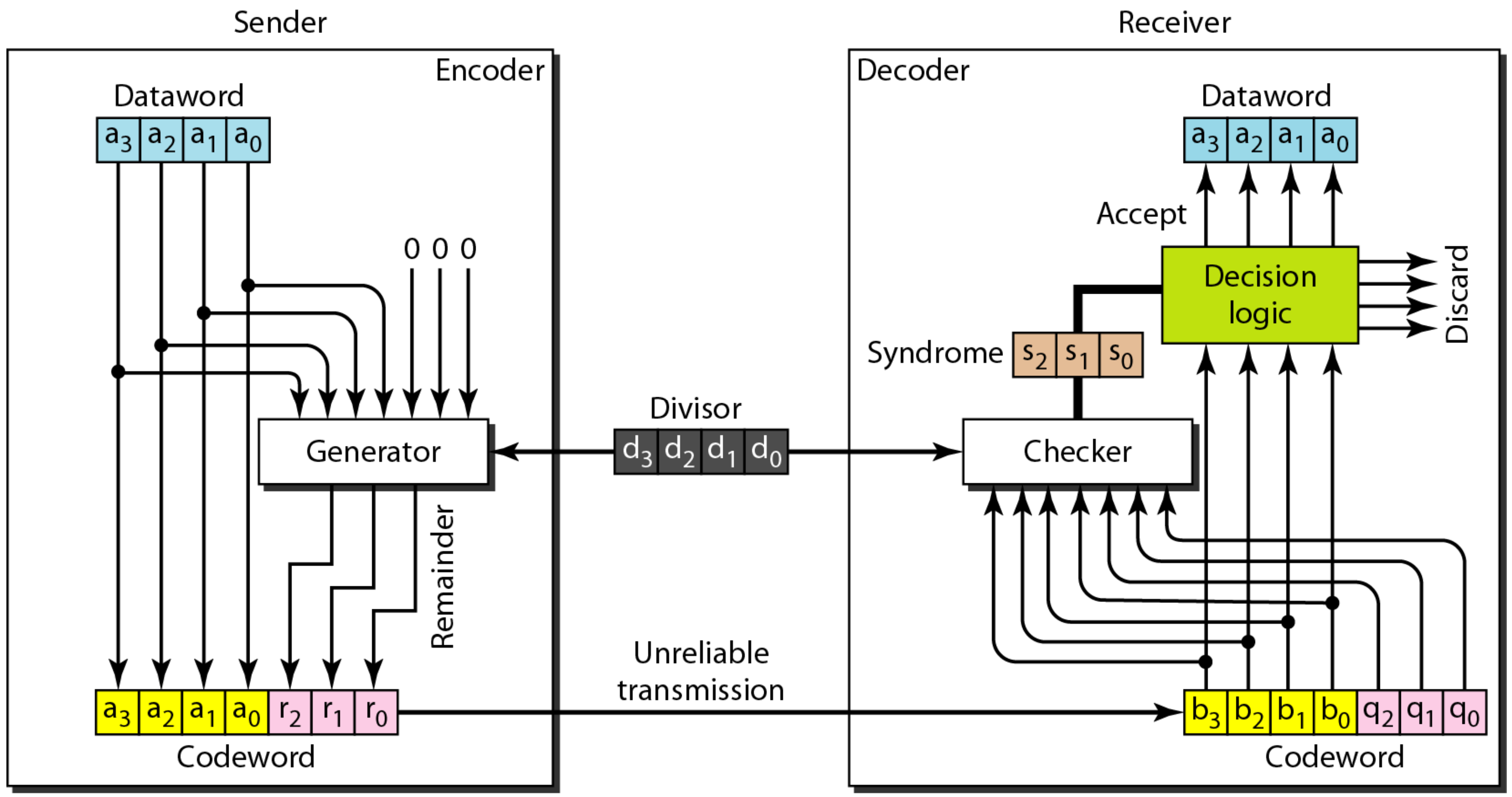


Figure 10.15 *Division in CRC encoder*

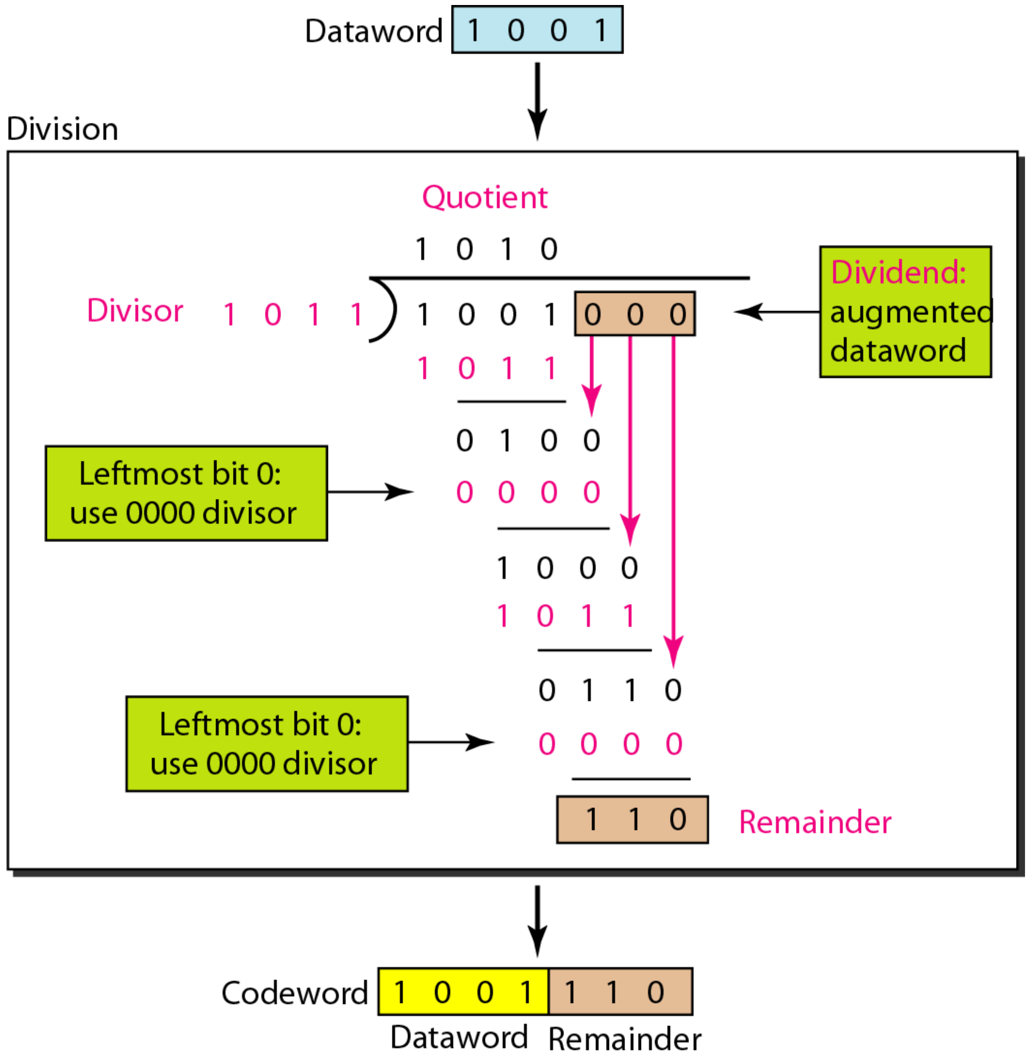
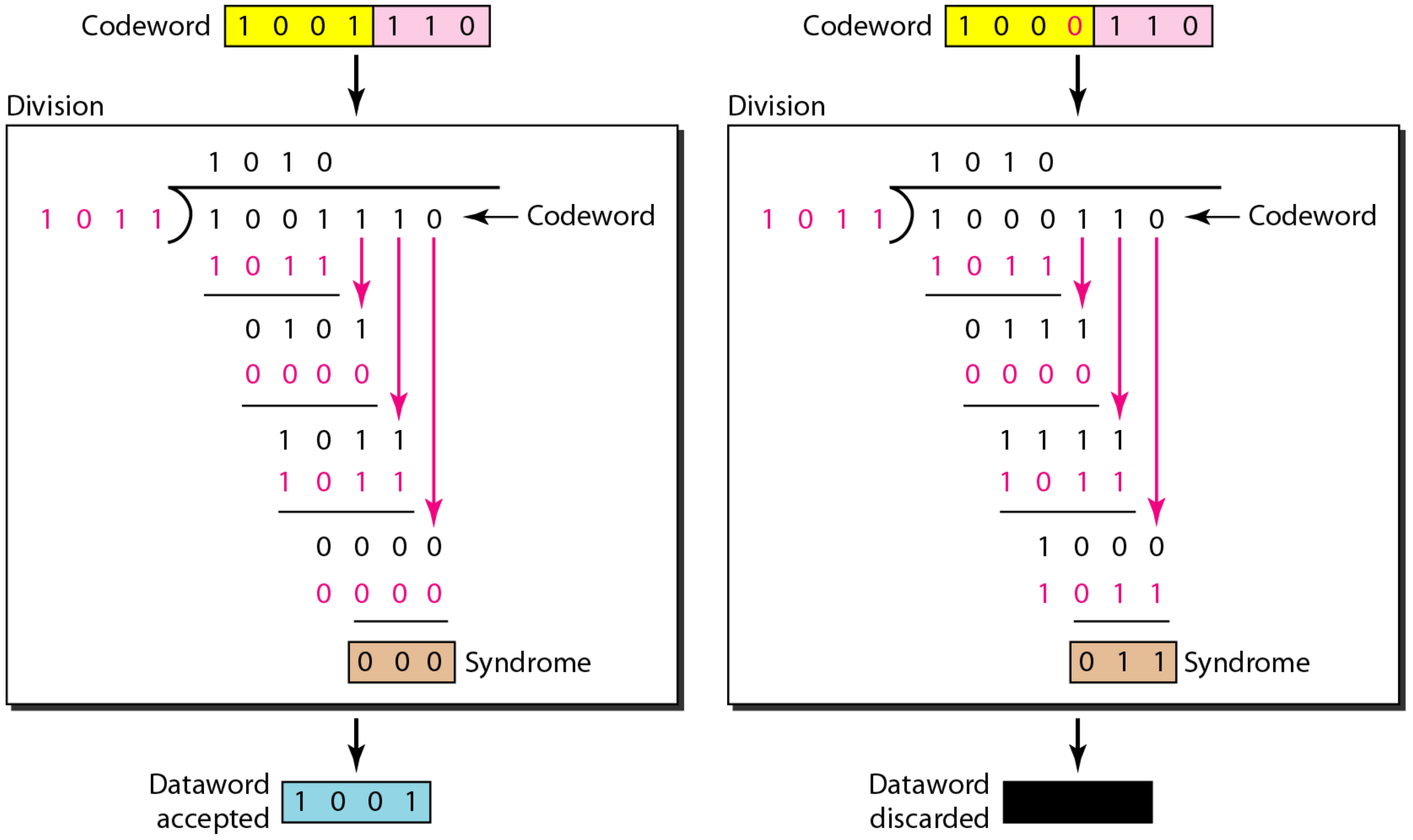


Figure 10.16 *Division in the CRC decoder for two cases*



Note

In a cyclic code,

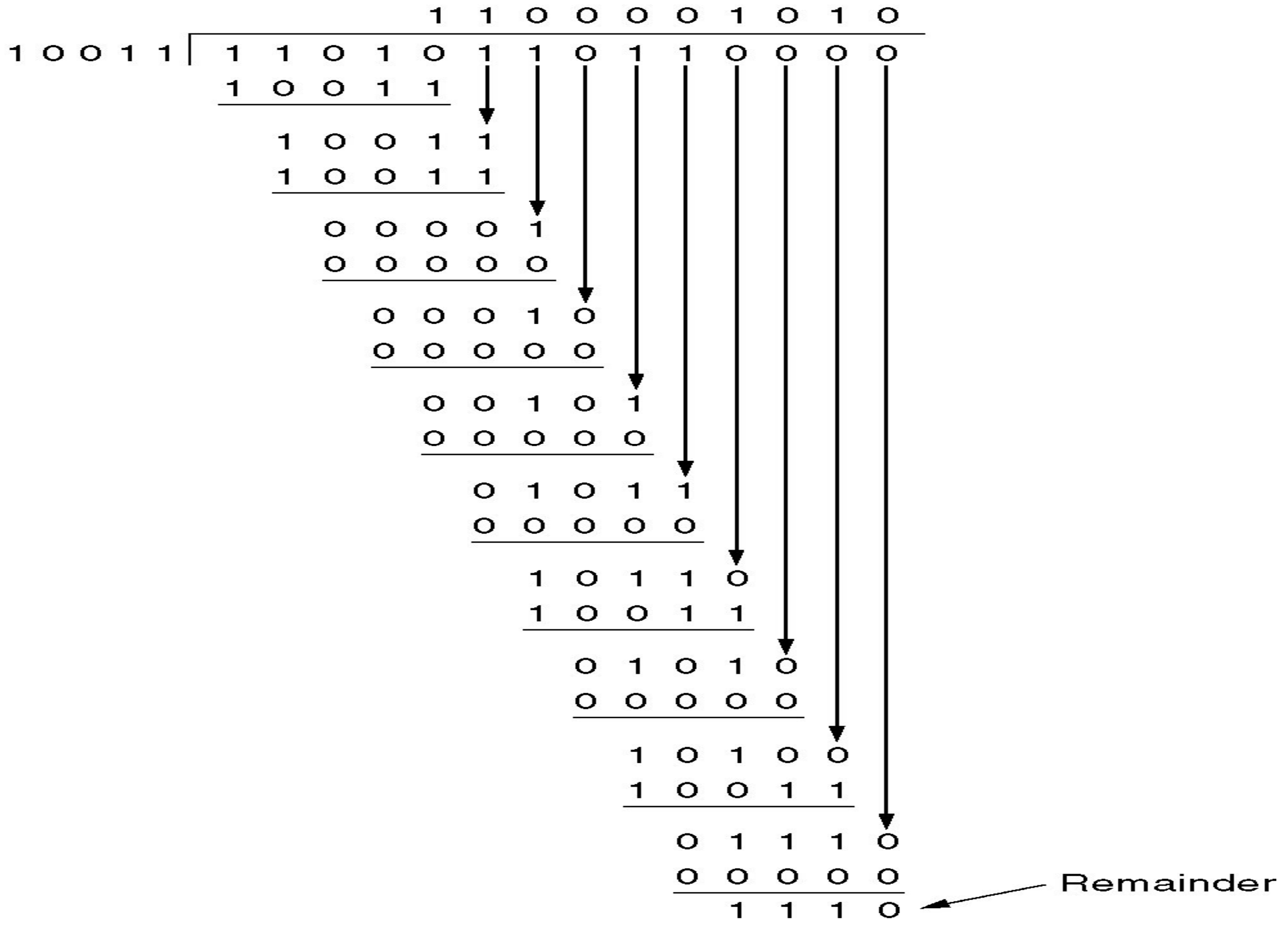
If $s(x) \neq 0$, one or more bits is corrupted.

If $s(x) = 0$, either

a. No bit is corrupted. or

b. Some bits are corrupted, but the decoder failed to detect them.

Frame : 1 1 0 1 0 1 1 0 1 1
Generator: 1 0 0 1 1
Message after 4 zero bits are appended: 1 1 0 1 0 1 1 0 1 1 0 0 0 0

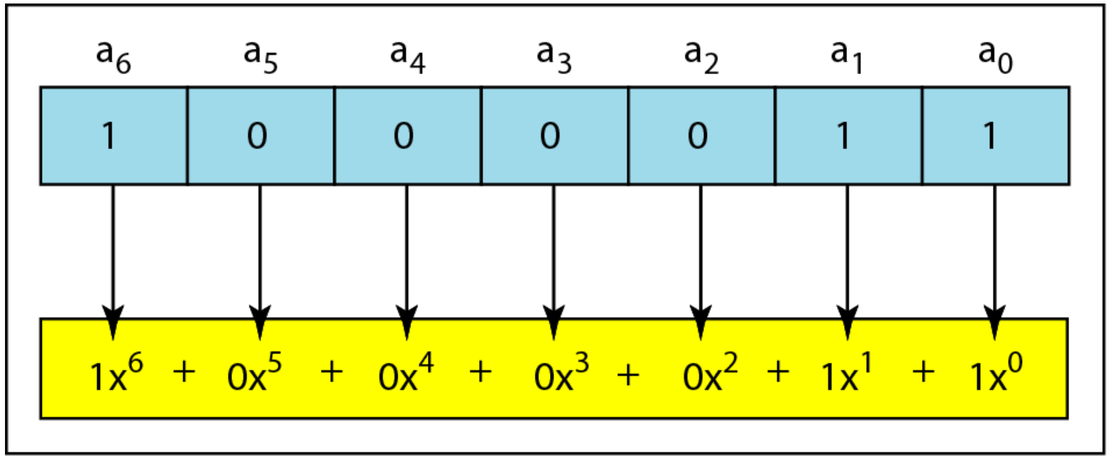


Transmitted frame: 1 1 0 1 0 1 1 0 1 1 1 1 1 0

Polynomials

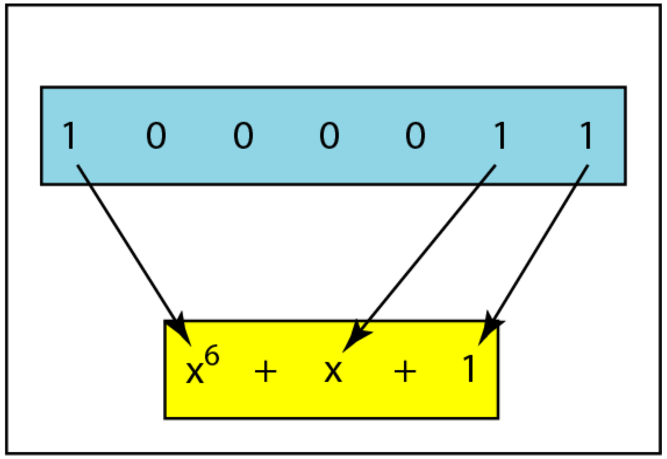
- Divisor G(n+1) bits
- Instead of 0s and 1s, represent it as polinomial
- Satisfy 2 properties:
  - Not divisible by x
  - Divisible by x + 1
- Divisor: 10100111
- Polinomial:

Figure 10.21 *A polynomial to represent a binary word*



a_6 a_5 a_4 a_3 a_2 a_1 a_0
1 0 0 0 0 1 1
$1x^6 + 0x^5 + 0x^4 + 0x^3 + 0x^2 + 1x^1 + 1x^0$

a. Binary pattern and polynomial



b. Short form

Note

A generator that contains a factor of $x + 1$ can detect all odd-numbered errors.

Note

A good polynomial generator needs to have the following characteristics:

1. It should have at least two terms.
2. The coefficient of the term x^0 should be 1.
3. It should not divide $x^t + 1$, for t between 2 and $n - 1$.
4. It should have the factor $x + 1$.

Table 10.7 ***Standard polynomials***

Name	*Polynomial*	*Application*
CRC-8	$x^8 + x^2 + x + 1$	ATM header
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x^2 + 1$	ATM AAL
CRC-16	$x^{16} + x^{12} + x^5 + 1$	HDLC
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	LANs

Checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender: (k many segments)

- treat segment contents as sequence of n-bits
- checksum: addition (1's complement sum) of segment contents
- Make total n-bits long
- Sum is complemented (checksum)
- puts checksum value at the end of data field

Receiver:

- treat segment contents as sequence of n-bits
- checksum: addition (1's complement sum) of segment contents
- Sum is complemented
- If result is all 0z
    - NO error detected
    - Else, error detected.

Note

Sender site:

1. **The message is divided into 16-bit words.**
2. **The value of the checksum word is set to 0.**
3. **All words including the checksum are added using one's complement addition.**
4. **The sum is complemented and becomes the checksum.**
5. **The checksum is sent with the data.**

Note

Receiver site:

1. **The message (including checksum) is divided into 16-bit words.**
2. **All words are added using one's complement addition.**
3. **The sum is complemented and becomes the new checksum.**
4. **If the value of checksum is 0, the message is accepted; otherwise, it is rejected.**

Error Correction

1. Error Correction by Retransmission
   1. When an error is detected, receiver have the sender retransmit the entire data unit

2. Forward Error Correction (FEC)
   1. Use error-correcting codes

Error-Correcting Code

In some cases it is sufficient to detect an error and in some, it requires the errors to be corrected also.

- On a reliable medium : detection is sufficient where the error rate is low and asking for retransmission after detection would work efficiently
- In contrast, on an unreliable medium : Retransmission after error detection may result in another error and still another and so on. Hence Error Correction is desirable.

Error-Correcting Code

- m data bits
- r error check bits(redundant)
- form an n = (m + r) bit codeword
- What r? r bits must indicate at least m+r+1 states
- m+r+1 states:
    - no error,
    - m+r : location of error in m+r posistions
- m+r+1 states must be identifiable by r bits
- r bits: can indicate 2^r different states. Therefore,

  $$2^r \geq m + r + 1$$

(m=1,r=2; m=2,r=3; m=5,r=4; m=7,r=4)

Table 10.4 ***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 10.12 *The structure of the encoder and decoder for a Hamming code*

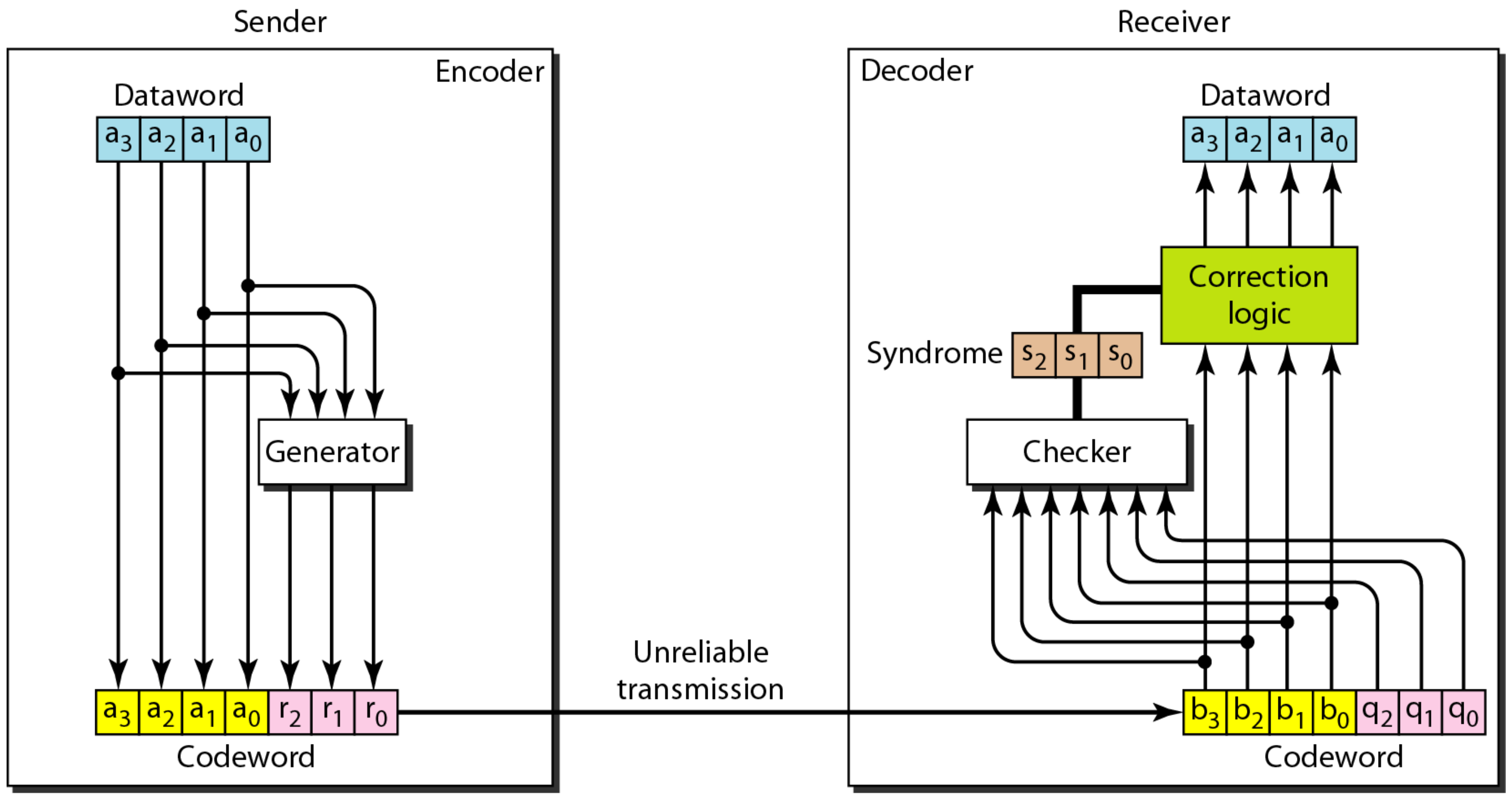
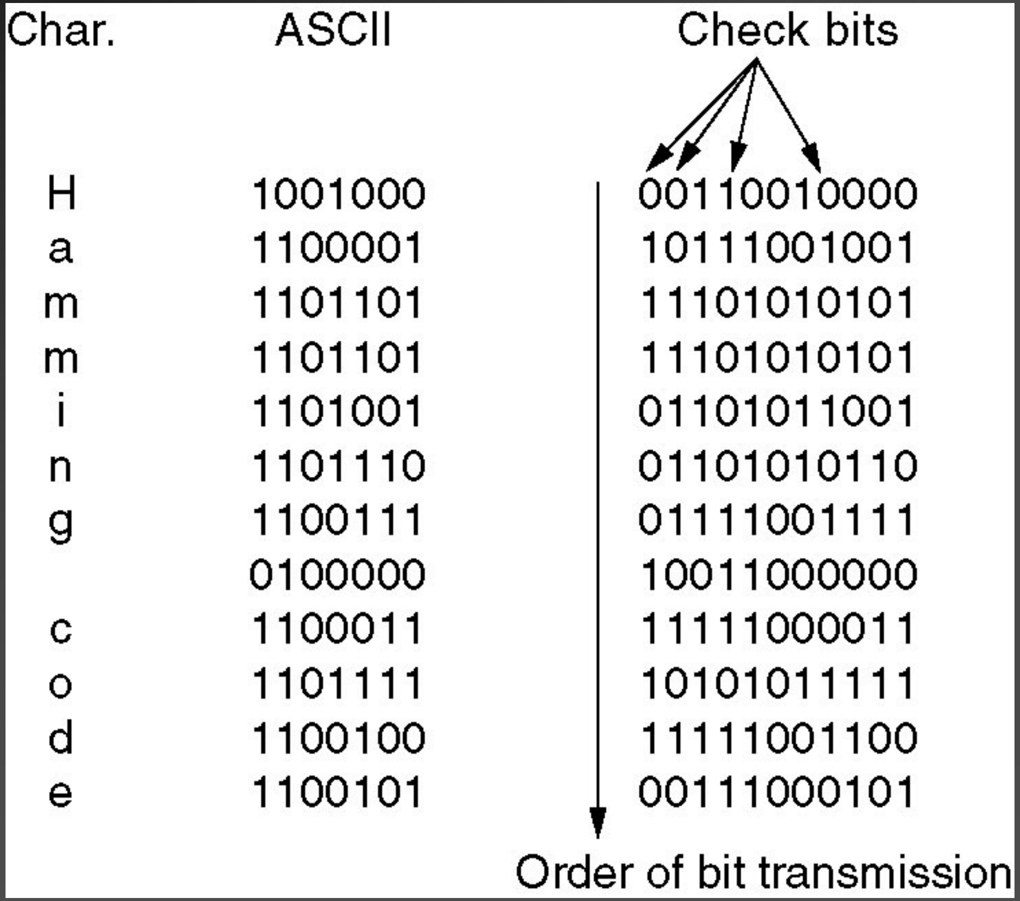


Table 10.5 *Logical decision made by the correction logic analyzer*

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

Hamming Code to correct burst errors



Char.	ASCII	Check bits
H	1001000	00110010000
a	1100001	10111001001
m	1101101	11101010101
m	1101101	11101010101
i	1101001	01101011001
n	1101110	01101010110
g	1100111	01111001111
	0100000	10011000000
c	1100011	11111000011
o	1101111	10101011111
d	1100100	11111001100
e	1100101	00111000101

Order of bit transmission

Pattern for Redundant Check Bits

Pos		11	10	9	8	7	6	5	4	3	2	1
Data		d	d	d		d	d	d		d		
Check					r_8				r_4		r_2	r_1

- r_n : skip n-1 bits, check n bits, skip n bits, check n bits, skip n bits, …
- Bits are numbered from 1 (not zero)
- Check bits are placed in every bit position that is a power of 2,
- For r_i , position is power of 2.
  - $r_1=1$, $r_2=2$, $r_4=4$, $r_8=8$, $r_{16}=16$,… (Positions)
  - 7=1+2+4, so pos 7 is checked by r_1,r_2,r_4
  - 9=1+8, so pos 9 is checked by r_1,r_8

Pattern for Redundant Check Bits

Pos		11	10	9	8	7	6	5	4	3	2	1
Data		d	d	d		d	d	d		d		
Check					r_8				r_4		r_2	r_1

- r_n : skip n-1 bits, check n bits, skip n bits, check n bits, skip n bits, …
- r_1 will take care of these bits

Pattern for Redundant Check Bits

Pos		11	10	9	8	7	6	5	4	3	2	1
Data		d	d	d		d	d	d		d		
Check					r_8				r_4		r_2	r_1

- r_n : skip n-1 bits, check n bits, skip n bits, check n bits, skip n bits, …
- r_2: will take care of these bits

Pattern for Redundant Check Bits

Pos		11	10	9	8	7	6	5	4	3	2	1
Data		d	d	d		d	d	d		d		
Check					r_8				r_4		r_2	r_1

- r_n : skip n-1 bits, check n bits, skip n bits, check n bits, skip n bits, …
- r_4 will take care of these bits

Pattern for Redundant Check Bits

Pos		11	10	9	8	7	6	5	4	3	2	1
Data		d	d	d		d	d	d		d		
Check					r_8				r_4		r_2	r_1

- r_n : skip n-1 bits, check n bits, skip n bits, check n bits, skip n bits, …
- r_8 will take care of these bits

Example Data to Send:1001101

Pos		11	10	9	8	7	6	5	4	3	2	1
Data		1	0	0		1	1	0		1		
Check					r_8				r_4		r_2	r_1

- Data: 1001101
- Place each data bit
- Compute r bits (even parity)
- r1= 1
- r2= 0
- r4= 0
- r8= 1
- Place each r bit

Codeword to Send

Pos		11	10	9	8	7	6	5	4	3	2	1
Data		1	0	0	1	1	1	0	0	1	0	1
Check					r_8				r_4		r_2	r_1

- Codeword: 1001 1100 101

Received Codeword (bit in pos 7 is in err)

Pos		11	10	9	8	7	6	5	4	3	2	1
Data		1	0	0	1	0	1	0	0	1	0	1
Check					r_8				r_4		r_2	r_1

- Correct Codeword: 1001 1100 101
- Compute check bits (even parity)
- r1= 1 (pos 1, 3, 5, 7, 9, 11)
- r2= 1 (pos 2,3, 6,7, 10, 11)
- r4= 1 (pos 4,5,6,7)
- r8= 0 (pos 8,9,10,11)

- r8 r4 r2 r1=0111=pos 7 is in error