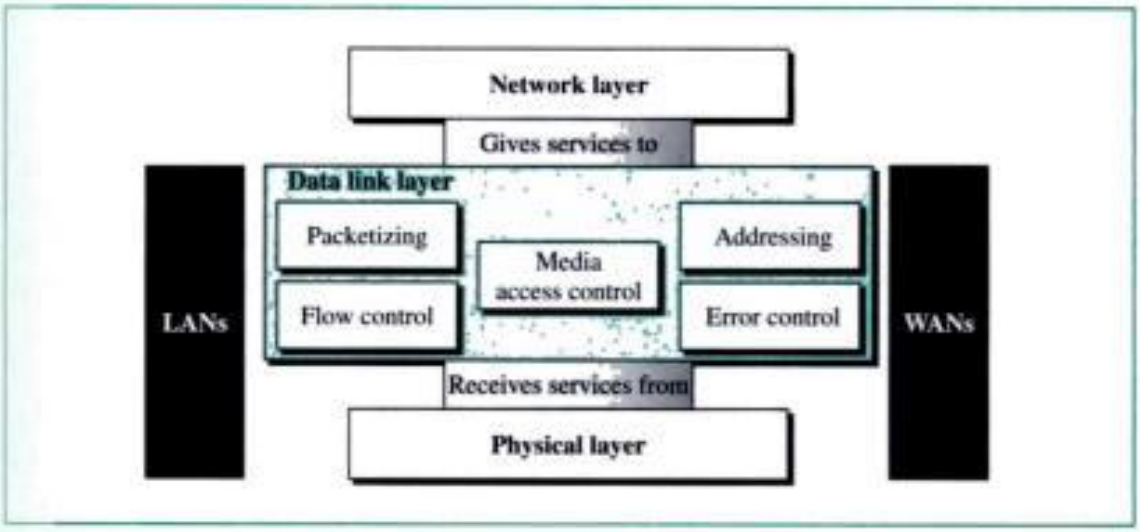
PART 3

Data Link Layer

The data link layer lies between the network layer and the physical layer in the Internet model. It receives services from the physical layer and provides services to the network layer. Figure 1 shows the position of the data link layer in the Internet model.

Figure 1 *Position of data link layer*



The **data link layer** is responsible for carrying a packet from one hop (computer or router) to the next hop. Unlike the network layer which has a global responsibility, the data link layer has a local responsibility. Its responsibility lies between two hops. In other words, because LANs and WANs in the Internet are delimited by hops, we can say that the responsibility of the data link layer is to carry a packet through a LAN or WAN.

The journey through a LAN or a WAN (between two hops) must preserve the integrity of the packet; the data link layer must make sure that the packet arrives safe and sound. If the packet is corrupted during the transmission, it must either be corrected or retransmitted. The data link layer must also make sure that the next hop is not overwhelmed with data by the previous hop; the flow of data must be controlled.

Access to a LAN or a WAN for the sending of data is also an issue. If several computers or routers are connected to a common medium (link), and more than one want to send data at the same time, which has the right to send? What is the access method?

allowed to overwhelm the receiver. The receiving device must be able to inform the sending device before some limit is reached and request that the transmitting device send fewer frames or stop temporarily. We discuss flow control as part of data link control in Chapter 11.

Medium Access Control

When computers use a shared medium (cable or air), there must be a method to control access to the medium at any moment. To prevent this conflict or collision on a network, there is a need for a medium access control (MAC) method. This method defines the procedure a computer follows when it needs to send a frame or frames. We devote two chapters to this issue, Chapter 12 and Chapter 13.

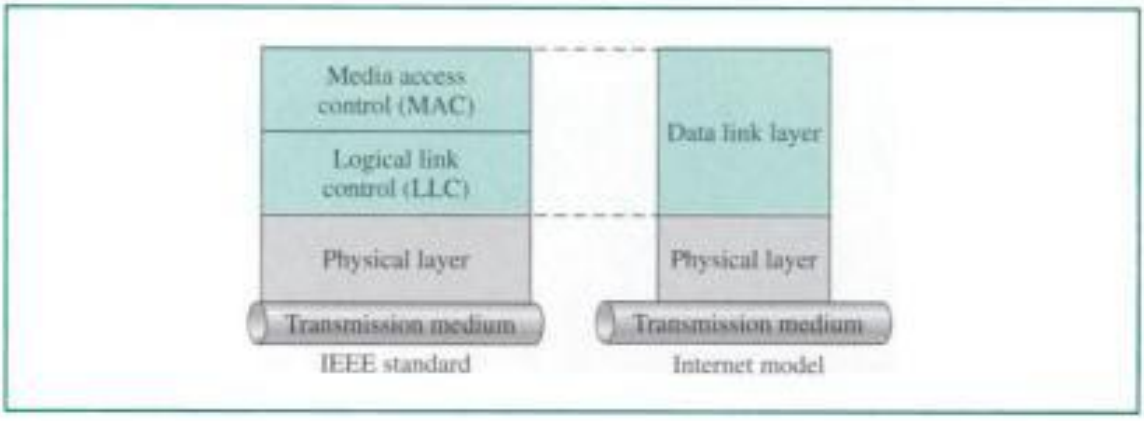
Local Area Networks

Local Area Networks (LANs) operate at the physical and data link layer. The obvious place to discuss local area networks is after these layers have been discussed. We have devoted Chapter 14 to Ethernet, the most common LAN today and Chapter 15 to wireless LANs, the most promising LAN today. After discussing these two subjects, we show how to connect LANs in Chapter 16.

IEEE Standards

The Internet does not spell out specifications for LANs or WANs. The Internet accepts any LAN as a communications pathway for transferring its network layer packet. There must other protocols to handle LANs. In 1985, the Computer Society of the IEEE started a project, called Project 802, to set standards to enable intercommunication between equipment from a variety of manufacturers. The IEEE has subdivided the data link layer into two sublayers: logical link control (LLC) and media access control (MAC) as shown in Figure 3. The LLC is nonarchitecture specific; that is, it is the same for all IEEE-defined LANs. It is not widely used today. The MAC sublayer, on the other hand, contains a number of distinct modules; each carries proprietary information specific to the LAN product being used. Figure 4 shows some IEEE 802 standards defined for specific LANs.

Figure 3 *LLC and MAC sublayers*



allowed to overwhelm the receiver. The receiving device must be able to inform the sending device before some limit is reached and request that the transmitting device send fewer frames or stop temporarily. We discuss flow control as part of data link control in Chapter 11.

Medium Access Control

When computers use a shared medium (cable or air), there must be a method to control access to the medium at any moment. To prevent this conflict or collision on a network, there is a need for a medium access control (MAC) method. This method defines the procedure a computer follows when it needs to send a frame or frames. We devote two chapters to this issue, Chapter 12 and Chapter 13.

Local Area Networks

Local Area Networks (LANs) operate at the physical and data link layer. The obvious place to discuss local area networks is after these layers have been discussed. We have devoted Chapter 14 to Ethernet, the most common LAN today and Chapter 15 to wireless LANs, the most promising LAN today. After discussing these two subjects, we show how to connect LANs in Chapter 16.

IEEE Standards

The Internet does not spell out specifications for LANs or WANs. The Internet accepts any LAN as a communications pathway for transferring its network layer packet. There must other protocols to handle LANs. In 1985, the Computer Society of the IEEE started a project, called Project 802, to set standards to enable intercommunication between equipment from a variety of manufacturers. The IEEE has subdivided the data link layer into two sublayers: logical link control (LLC) and media access control (MAC) as shown in Figure 3. The LLC is nonarchitecture specific; that is, it is the same for all IEEE-defined LANs. It is not widely used today. The MAC sublayer, on the other hand, contains a number of distinct modules; each carries proprietary information specific to the LAN product being used. Figure 4 shows some IEEE 802 standards defined for specific LANs.

Figure 3 *LLC and MAC sublayers*

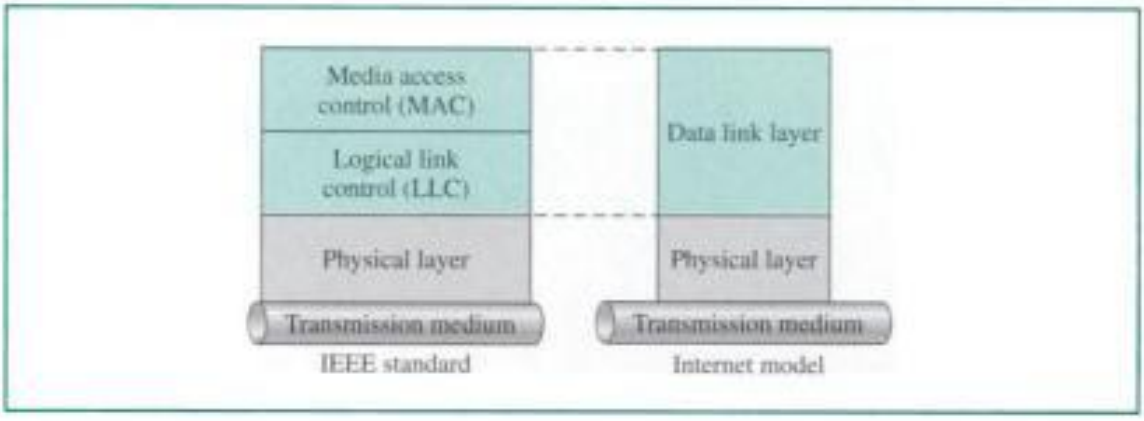
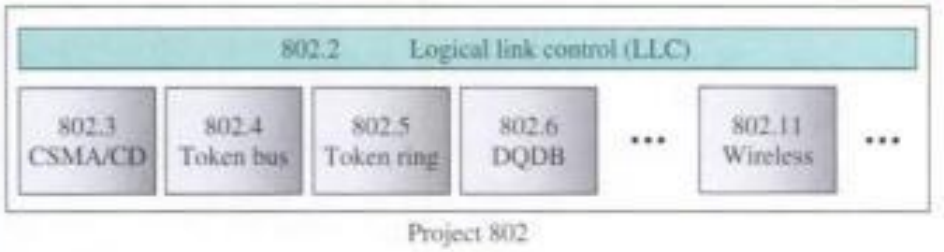


Figure 4 *IEEE standards for LANs*



Wide Area Networks

Wide Area Networks (WANs) also operate in the physical and data link layer discussed in this part of the text. We discuss the mobile telephone systems and lites, as wireless WANs, in Chapter 17. We discuss Frame Relay and ATM as sv WANs in Chapter 18.

Chapters

Part III of the book covers nine chapters 10–18. Chapters 10–13 discuss general s provided by the data link layer: error control, flow control, and media access. Cha is about error detection, a prelude to error control. Chapter 11 is about flow and er trol. Chapter 12 explains media access control for point-to-point connections. Cha does the same for multiple access connections.

Chapters 14 to 16 are devoted to LANs. Chapter 14 discuss the most c LAN, Ethernet. Chapter 15 discusses wireless LANs. Chapter 16 discusses the c tion of LANs.

Chapters 17 and 18 are devoted to WANs. Chapter 17 is about wireless mobile telephone networks and satellite networks. Chapter 18 is about sw WANs, Frame Relay, and ATM.

CHAPTER 10

Error Detection and Correction

Networks must be able to transfer data from one device to another with complete accuracy. A system that cannot guarantee that the data received by one device are identical to the data transmitted by another device is essentially useless. Yet anytime data are transmitted from one node to the next, they can become corrupted in passage. Many factors can alter or wipe out one or more bits of a given data unit. Reliable systems must have a mechanism for detecting and correcting such **errors.**

Data can be corrupted during transmission. For reliable communication, errors must be detected and corrected.

10.1 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 0. In a burst error, multiple bits are changed. For example, a 0.01-s burst of impulse noise on a transmission with a data rate of 1200 bps might change all or some of 12 bits of information.

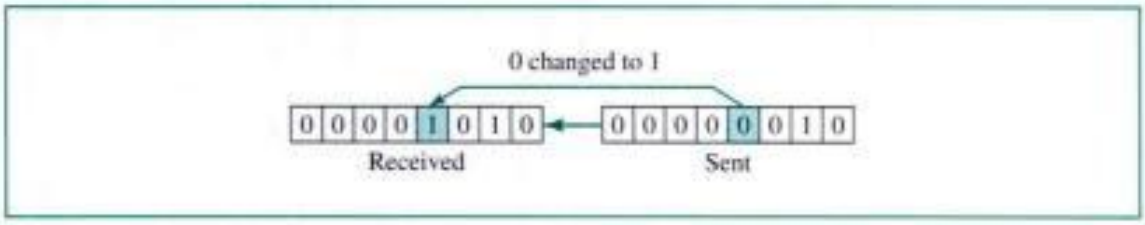
Single-Bit Error

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

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

Figure 10.1 shows the effect of a single-bit error on a data unit. To understand the impact of the change, imagine that each group of 8 bits is an ASCII character with a 0 bit added to the left. In the figure, 00000010 (ASCII *STX*) was sent, meaning *start of text*, but 00001010 (ASCII *LF*) was received, meaning *line feed*. (For more information about ASCII code, see Appendix A.)

Figure 10.1 *Single-bit error*



Single-bit errors are the least likely type of error in serial data transmission. To understand why, imagine a sender sends data at 1 Mbps. This means that each bit lasts only 1/1,000,000 s, or 1 μs. For a single-bit error to occur, the noise must have a duration of only 1 μs, which is very rare; noise normally lasts much longer than this.

However, a single-bit error can happen if we are sending data using parallel transmission. For example, if eight wires are used to send all 8 bits of 1 byte at the same time and one of the wires is noisy, one bit can be corrupted in each byte. Think of parallel transmission inside a computer, between CPU and memory, for example.

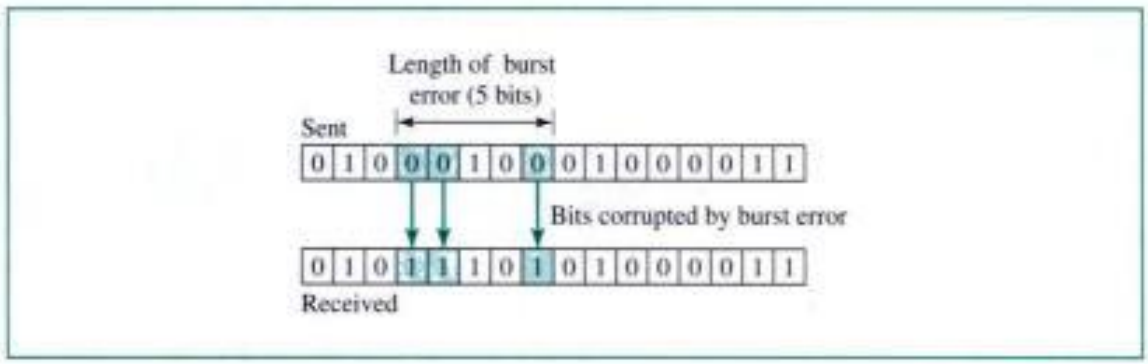
Burst Error

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.

A burst error means that 2 or more bits in the data unit have changed.

Figure 10.2 shows the effect of a burst error on a data unit. In this case, 0100010001000011 was sent, but 0101110101000011 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.

Figure 10.2 *Burst error of length 5*



Burst error is most likely to occur in a serial transmission. The duration of noise is normally longer than the duration of one bit, which means that when noise affects data, it affects a set of bits. The number of bits affected depends on the data rate and duration of noise. For example, if we are sending data at 1 Kbps, a noise of 1/100 s can affect 10 bits; if we are sending data at 1 Mbps, the same noise can affect 10,000 bits.

10.2 DETECTION

Although the goal of error checking is to correct errors, most of the time, we first need to detect errors. Error detection is simpler than error correction and is the first step in the error correction process.

Redundancy

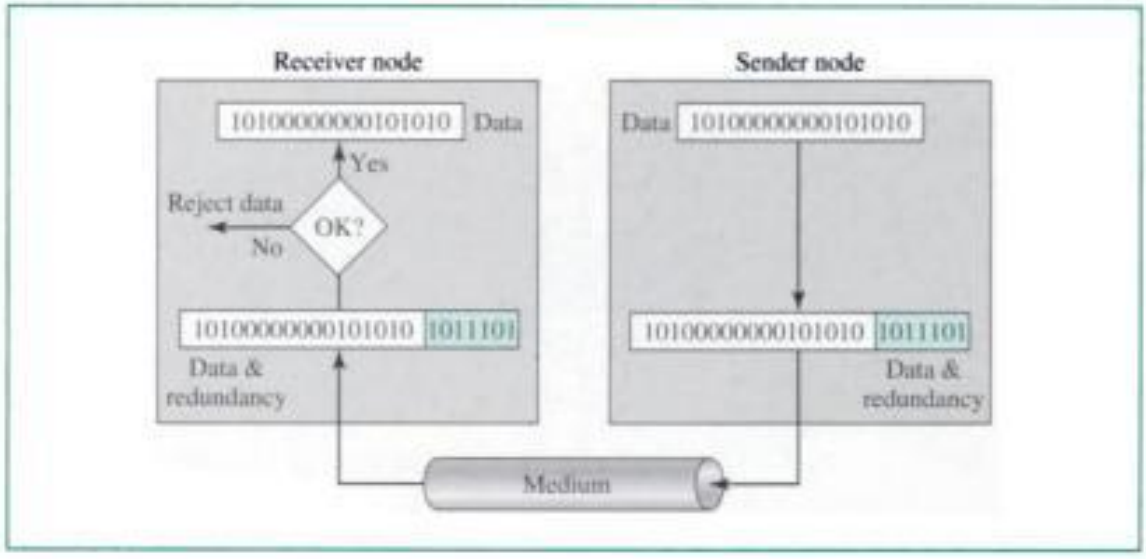
One **error detection** mechanism would be to send every data unit twice. The receiving device would then be able to do a bit-for-bit comparison between the two versions of the data. Any discrepancy would indicate an error, and an appropriate correction mechanism could be set in place. This system would be completely accurate (the odds of errors being introduced onto exactly the same bits in both sets of data are infinitesimally small), but it would also be insupportably slow. Not only would the transmission time double, but also the time it takes to compare every unit bit by bit must be added.

The concept of including extra information in the transmission for error detection is a good one. But instead of repeating the entire data stream, a shorter group of bits may be appended to the end of each unit. This technique is called **redundancy** because the extra bits are redundant to the information; they are discarded as soon as the accuracy of the transmission has been determined.

Error detection uses the concept of redundancy, which means adding extra bits for detecting errors at the destination.

Figure 10.3 shows the process of using redundant bits to check the accuracy of a data unit. Once the data stream has been generated, it passes through a device that analyzes it and adds on an appropriately coded redundancy check. The data unit, now enlarged by several bits, travels over the link to the receiver. The receiver puts

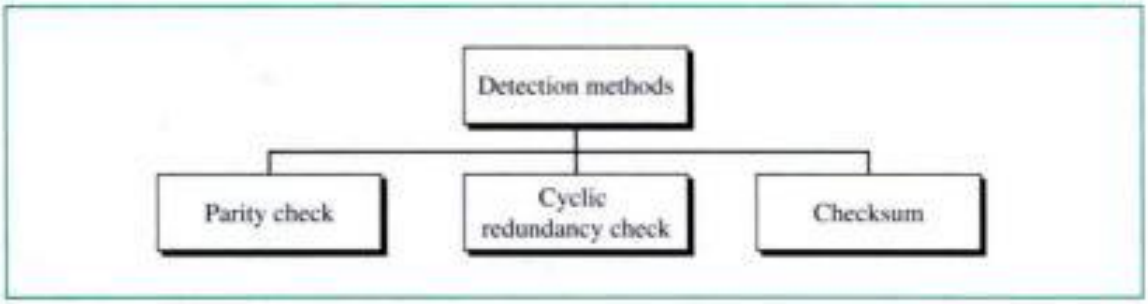
Figure 10.3 *Redundancy*



the entire stream through a checking function. If the received bit stream passes the checking criteria, the data portion of the data unit is accepted and the redundant bits are discarded.

Three types of redundancy checks are common in data communications: parity check, cyclic redundancy check (CRC), and checksum (see Fig. 10.4).

Figure 10.4 *Detection methods*



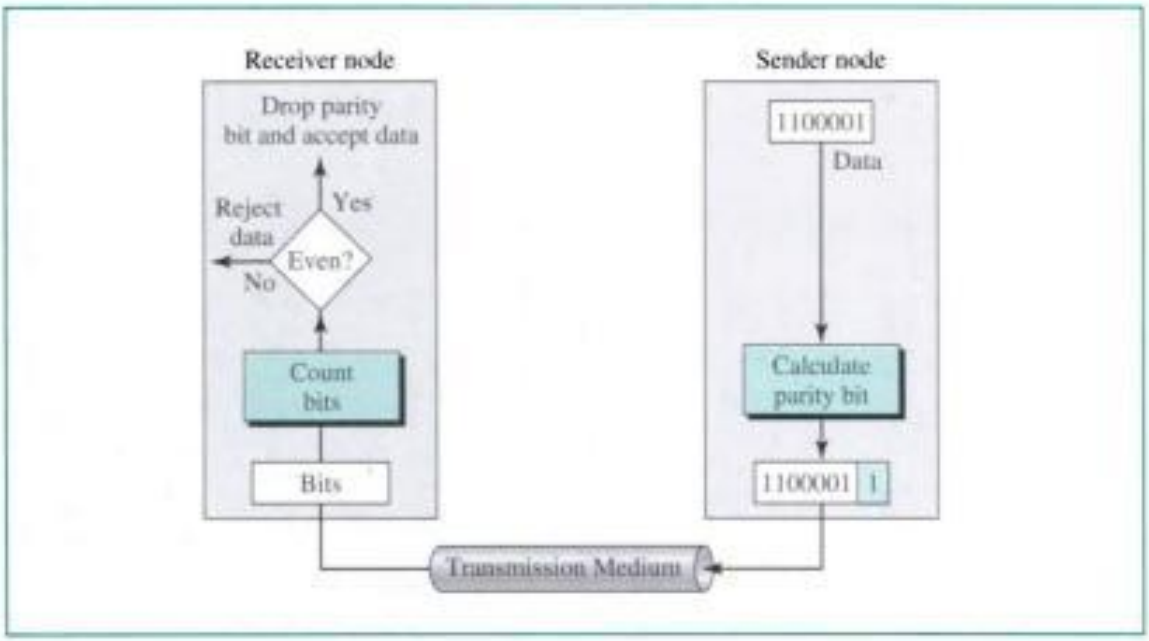
Parity Check

The most common and least expensive mechanism for error detection is the **parity check.** Parity checking can be simple or two-dimensional.

Simple Parity Check

In this technique, a redundant bit, called a **parity bit,** is added to every data unit so that the total number of 1s in the unit (including the parity bit) becomes even (or odd). Suppose we want to transmit the binary data unit 1100001 [ASCII *a* (97)]; see Figure 10.5.

Figure 10.5 *Even-parity concept*



Adding the number of 1s gives us 3, an odd number. Before transmitting, we pass the data unit through a parity generator. The parity generator counts the 1s and appends the parity bit (a 1 in this case) to the end. The total number of 1s is now 4, an even number. The system now transmits the entire expanded unit across the network link. When it reaches its destination, the receiver puts all 8 bits through an **even-parity** checking function. If the receiver sees 11000011, it counts four 1s, an even number, and the data unit passes. But what if the data unit has been damaged in transit? What if, instead of 11000011, the receiver sees 11001011? Then when the parity checker counts the 1s, it gets 5, an odd number. The receiver knows that an error has been introduced into the data somewhere and therefore rejects the whole unit. Note that for the sake of simplicity, we are discussing here even-parity checking, where the number of 1s should be an even number. Some systems may use **odd-parity** checking, where the number of 1s should be odd. The principle is the same.

> In parity check, a parity bit is added to every data unit so that the total number of 1s is even (or odd for odd-parity).

Example 1

Suppose the sender wants to send the word *world*. In ASCII (see Appendix A), the five characters are coded as

```
1110111  1101111  1110010  1101100  1100100
   w        o        r        l        d
```

Each of the first four characters has an even number of 1s, so the parity bit is a 0. The last character (d), however, has three 1s (an odd number), so the parity bit is a 1 to make the total number of 1s even. The following shows the actual bits sent (the parity bits are underlined).

```
11101110  11011110  11100100  11011000  11001001
```

Example 2

Now suppose the word *world* in Example 1 is received by the receiver without being corrupted in transmission.

```
11101110  11011110  11100100  11011000  11001001
```

The receiver counts the 1s in each character and comes up with even numbers (6, 6, 4, 4, 4). The data are accepted.

Example 3

Now suppose the word *world* in Example 1 is corrupted during transmission.

```
11111110  11011110  11101100  11011000  11001001
```

The receiver counts the 1s in each character and comes up with even and odd numbers (7, 6, 5, 4, 4). The receiver knows that the data are corrupted, discards them, and asks for retransmission.

Performance

Simple parity check can detect all single-bit errors. It can also detect burst errors as long as the total number of bits changed is odd (1, 3, 5, etc.). Let's say we have an even-parity data unit where the total number of 1s, including the parity bit, is 6: 1000111011. If any 3 bits change value, the resulting parity will be odd and the error will be detected: 1*111*111011:9, *011*0111011:7, 1*00010*011:5—all odd. The checker would return a result of 1, and the data unit would be rejected. The same holds true for any odd number of errors.

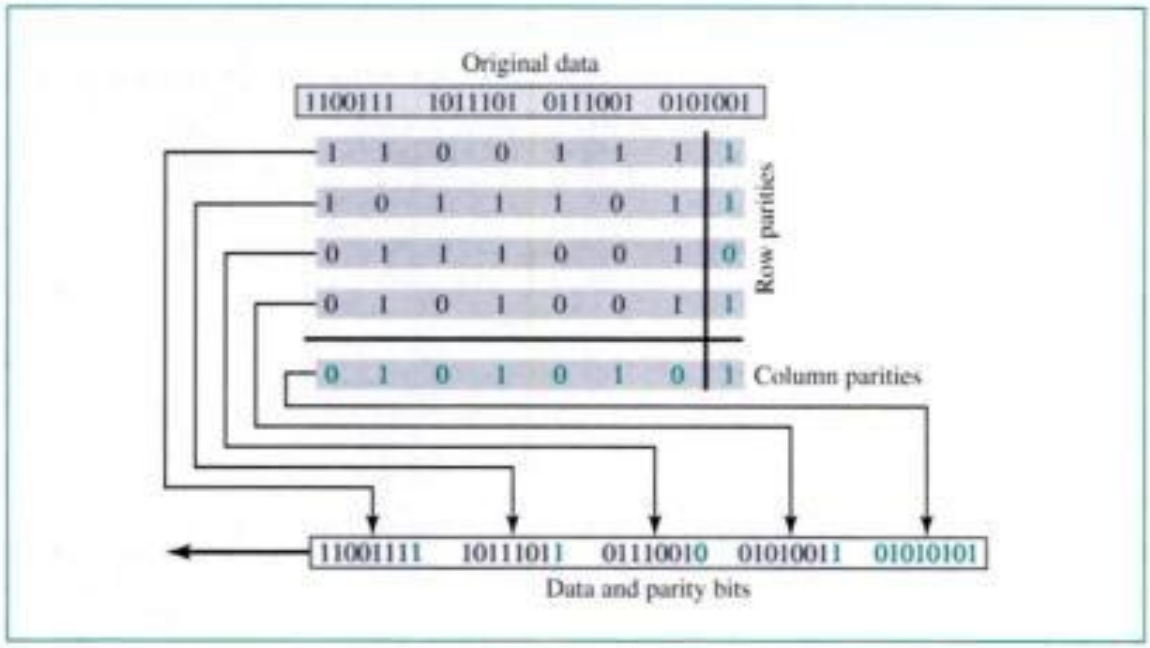
Suppose, however, that 2 bits of the data unit are changed: 1*11*0111011:8, 1*00*011011:6, 1000*11010*:4. In each case the number of 1s in the data unit is still even. The parity checker will add them and return an even number although the data unit contains two errors. This method cannot detect errors where the total number of bits changed is even. If any two bits change in transmission, the changes cancel each other and the data unit will pass a parity check even though the data unit is damaged. The same holds true for any even number of errors.

Simple parity check can detect all single-bit errors. It can detect burst errors only if the total number of errors in each data unit is odd.

Two-Dimensional Parity Check

A better approach is the **two-dimensional parity check.** In this method, a block of bits is organized in a table (rows and columns). First we calculate the parity bit for each data unit. Then we organize them into a table. For example, as shown in Figure 10.6, we have four data units shown in four rows and eight columns. We then calculate the parity bit for each column and create a new row of 8 bits; they are the parity bits for the whole block. Note that the first parity bit in the fifth row is calculated based on all first

Figure 10.6 *Two-dimensional parity*



bits; the second parity bit is calculated based on all second bits; and so on. We then attach the 8 parity bits to the original data and send them to the receiver.

Example 4

Suppose the following block is sent:

```
10101001 00111001 11011101 11100111 10101010
```

However, it is hit by a burst noise of length 8, and some bits are corrupted.

```
10100011 10001001 11011101 11100111 10101010
```

When the receiver checks the parity bits, some of the bits do not follow the even-parity rule and the whole block is discarded (the nonmatching bits are shown in bold).

```
10100011 10001001 11011101 11100111 10101010
                                     (parity bits)
```

In two-dimensional parity check, a block of bits is divided into rows and a redundant row of bits is added to the whole block.

Performance

Two-dimensional parity check increases the likelihood of detecting burst errors. As we showed in Example 4, a redundancy of n bits can easily detect a burst error of n bits. A burst error of more than n bits is also detected by this method with a very high probability. There is, however, one pattern of errors that remains elusive. If 2 bits in one data unit are damaged and two bits *in exactly the same positions* in another data unit are also damaged, the checker will not detect an error. Consider, for example, two data units: 11110000 and 11000011. If the first and last bits in each of them are changed, making the data units read *0*1110000*1* and *0*1000010, the errors cannot be detected by this method.

Cyclic Redundancy Check (CRC)

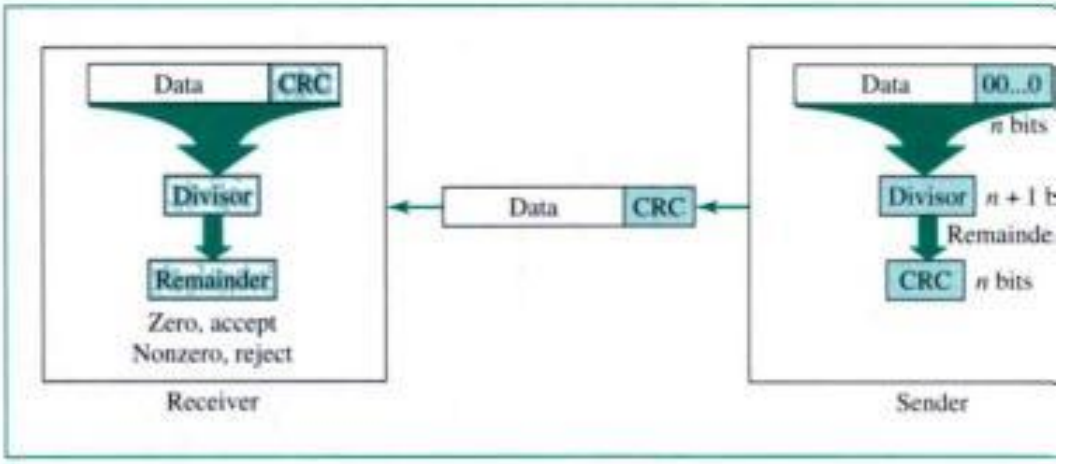
The third and most powerful of the redundancy checking techniques is the **cyclic redundancy check (CRC).** Unlike the parity check which is based on addition, CRC is based on binary division. In CRC, instead of adding bits to achieve a desired parity, a sequence of redundant bits, called the CRC or the CRC remainder, is appended to the end of a data unit so that the resulting data unit becomes exactly divisible by a second, predetermined binary number. At its destination, the incoming data unit is divided by the same number. If at this step there is no remainder, the data unit is assumed to be intact and is therefore accepted. A remainder indicates that the data unit has been damaged in transit and therefore must be rejected.

The redundancy bits used by CRC are derived by dividing the data unit by a predetermined divisor; the remainder is the CRC. To be valid, a CRC must have two qualities: It must have exactly one less bit than the divisor, and appending it to the end of the data string must make the resulting bit sequence exactly divisible by the divisor.

Both the theory and the application of CRC error detection are straightforward. The only complexity is in deriving the CRC. To clarify this process, we will start with

an overview and add complexity as we go. Figure 10.7 provides an outline of the basic steps.

Figure 10.7 *CRC generator and checker*



First, a string of n 0s is appended to the data unit. The number n is 1 less th number of bits in the predetermined divisor, which is $n + 1$ bits.

Second, the newly elongated data unit is divided by the divisor, using a p called binary division. The remainder resulting from this division is the CRC.

Third, the CRC of n bits derived in step 2 replaces the appended 0s at the the data unit. Note that the CRC may consist of all 0s.

The data unit arrives at the receiver data first, followed by the CRC. The re treats the whole string as a unit and divides it by the same divisor that was used the CRC remainder.

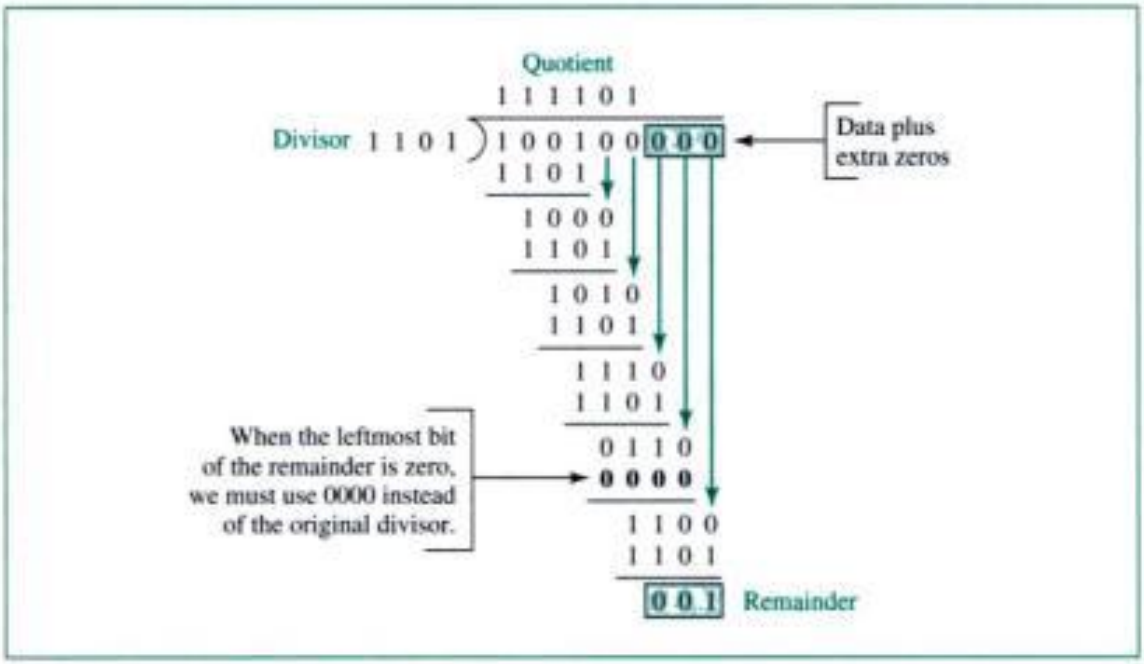
If the string arrives without error, the CRC checker yields a remainder of ze the data unit passes. If the string has been changed in transit, the division yields zero remainder and the data unit does not pass.

The CRC Generator

A **CRC generator** uses modulo-2 division. Figure 10.8 shows this process. In th step, the 4-bit divisor is subtracted from the first 4 bits of the dividend. Each bit divisor is subtracted from the corresponding bit of the dividend without disturbi next-higher bit. In our example, the divisor, 1101, is subtracted from the first 4 the dividend, 1001, yielding 100 (the leading 0 of the remainder is dropped). Th unused bit from the dividend is then pulled down to make the number of bits remainder equal to the number of bits in the divisor. The next step, therefore, is 1101, which yields 101, and so on.

In this process, the divisor always begins with a 1; the divisor is subtracted portion of the previous dividend/remainder that is equal to it in length; the divis only be subtracted from a dividend/remainder whose leftmost bit is 1. Anytime th most bit of the dividend/remainder is 0, a string of 0s, of the same length as the d replaces the divisor in that step of the process. For example, if the divisor is 4 bits is replaced by four 0s. (Remember, we are dealing with bit patterns, not with quan values; 0000 is not the same as 0.) This restriction means that, at any step, the le

Figure 10.8 *Binary division in a CRC generator*



subtraction will be either 0 − 0 or 1 − 1, both of which equal 0. So, after subtraction, the leftmost bit of the remainder will always be a leading zero, which is dropped, and the next unused bit of the dividend is pulled down to fill out the remainder. Note that only the first bit of the remainder is dropped—if the second bit is also 0, it is retained, and the dividend/remainder for the next step will begin with 0. This process repeats until the entire dividend has been used.

The CRC Checker

A **CRC checker** functions exactly as the generator does. After receiving the data appended with the CRC, it does the same modulo-2 division. If the remainder is all 0s, the CRC is dropped and the data are accepted; otherwise, the received stream of bits is discarded and data are resent. Figure 10.9 shows the same process of division in the receiver. We assume that there is no error. The remainder is therefore all 0s, and the data are accepted.

Polynomials

The divisor in the CRC generator is most often represented not as a string of 1s and 0s, but as an algebraic **polynomial** (see Fig. 10.10). The polynomial format is useful for two reasons: It is short, and it can be used to prove the concept mathematically (which is beyond the scope of this book).

The relationship of a polynomial to its corresponding binary representation is shown in Figure 10.11.

A polynomial should be selected to have at least the following properties:

- It should not be divisible by x.
- It should be divisible by $x + 1$.

Figure 10.9 *Binary division in CRC checker*

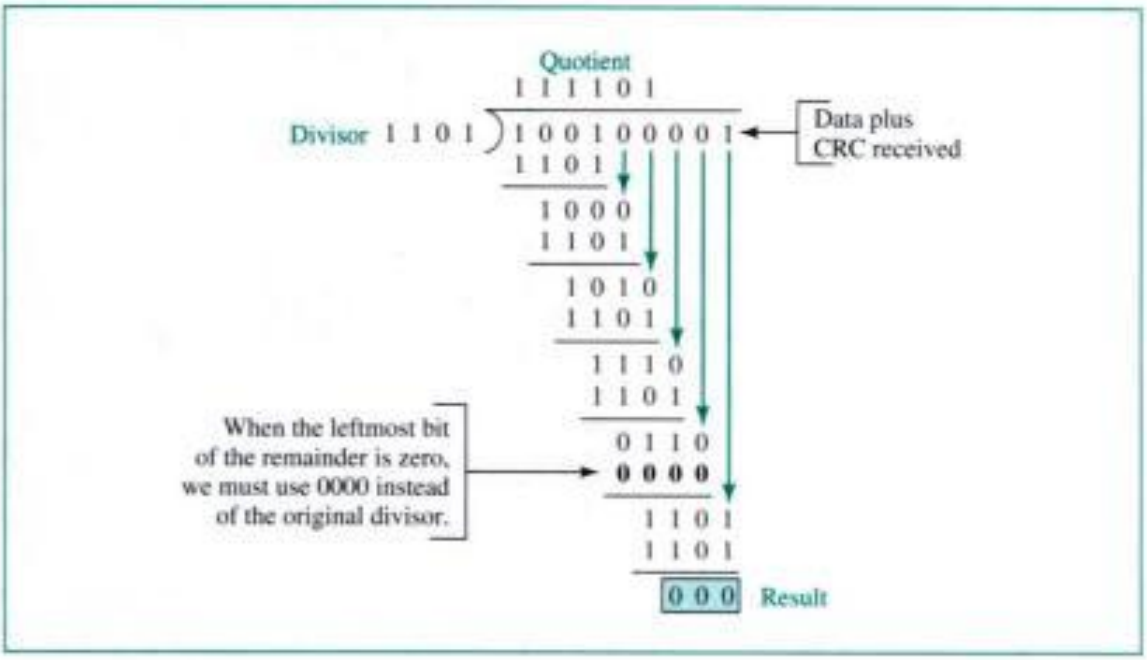


Figure 10.10 *A polynomial*

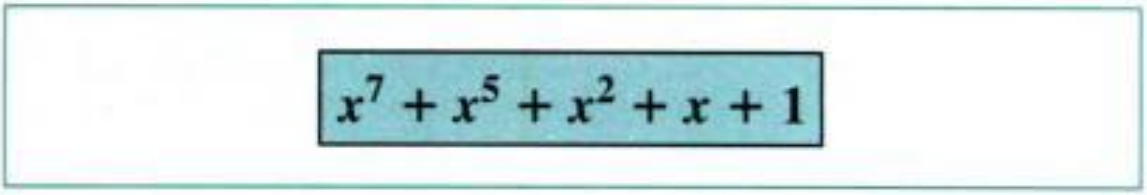
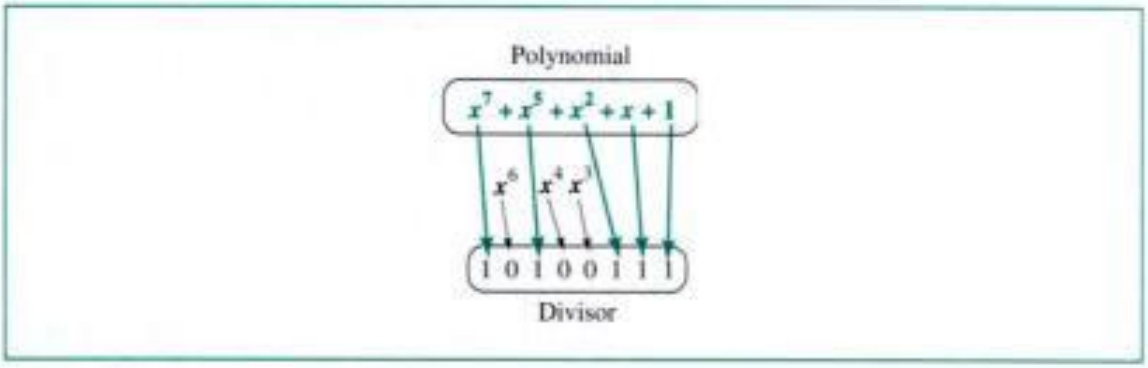


Figure 10.11 *A polynomial representing a divisor*



The first condition guarantees that all burst errors of a length equal to the degree of the polynomial are detected. The second condition guarantees that all burst errors affecting an odd number of bits are detected (the proof is beyond the scope of this book).

Example 5

It is obvious that we cannot choose x (binary 10) or $x^2 + x$ (binary 110) as the polynomial because both are divisible by x. However, we can choose $x + 1$ (binary 11) because it is not divisible by x,

but is divisible by $x + 1$. We can also choose $x^2 + 1$ (binary 101) because it is divisible by $x + 1$ (binary division).

Standard Polynomials

Some standard polynomials used by popular protocols for CRC generation are shown in Table 10.1.

Table 10.1 *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
ITU-16	$x^{16} + x^{12} + x^5 + 1$	HDLC
ITU-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

Performance

CRC is a very effective error detection method. If the divisor is chosen according to the previously mentioned rules,

1. CRC can detect all burst errors that affect an odd number of bits.
2. CRC can detect all burst errors of length less than or equal to the degree of the polynomial.
3. CRC can detect, with a very high probability, burst errors of length greater than the degree of the polynomial.

Example 6

The CRC-12 ($x^{12} + x^{11} + x^3 + x + 1$), which has a degree of 12, will detect all burst errors affecting an odd number of bits, will detect all burst errors with a length less than or equal to 12, and will detect, 99.97 percent of the time, burst errors with a length of 12 or more.

Checksum

The third error detection method we discuss here is called the **checksum.** Like the parity checks and CRC, the checksum is based on the concept of redundancy.

Checksum Generator

In the sender, the checksum generator subdivides the data unit into equal segments of n bits (usually 16). These segments are added using **ones complement** arithmetic (see Appendices B and E) in such a way that the total is also n bits long. That total (sum) is then complemented and appended to the end of the original data unit as redundancy bits, called the checksum field. The extended data unit is transmitted across the network. So if the sum of the data segment is T, the checksum will be $-T$ (see Figs. 10.12 and 10.13).

Figure 10.12 *Checksum*

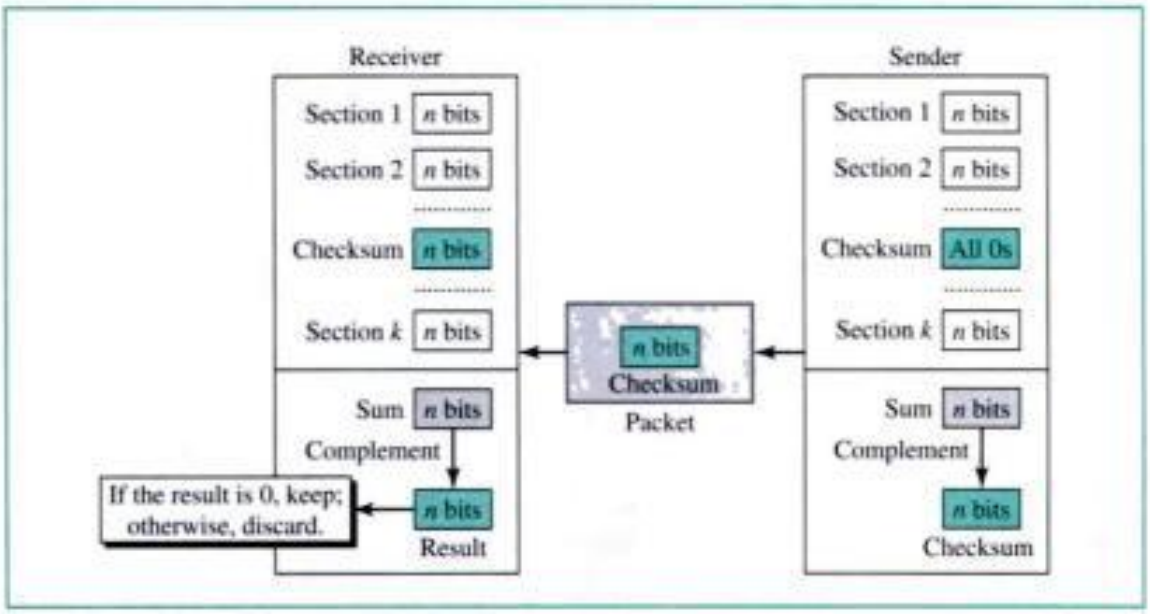
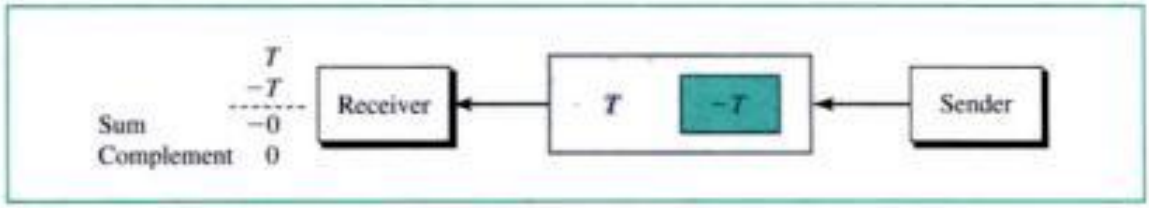


Figure 10.13 *Data unit and checksum*



The sender follows these steps:

- The unit is divided into k sections, each of n bits.
- All sections are added using ones complement to get the sum.
- The sum is complemented and becomes the checksum.
- The checksum is sent with the data.

Checksum Checker

The receiver subdivides the data unit as above and adds all segments and complements the result. If the extended data unit is intact, the total value found by adding the data

The receiver follows these steps:

- The unit is divided into k sections, each of n bits.
- All sections are added using ones complement to get the sum.
- The sum is complemented.
- If the result is zero, the data are accepted: otherwise, they are rejected.

segments and the checksum field should be zero. If the result is not zero, the packet contains an error and the receiver rejects it (see Appendix E).

Example 7

Suppose the following block of 16 bits is to be sent using a checksum of 8 bits.

```
10101001  00111001
```

The numbers are added using ones complement arithmetic (see Appendix E).

	10101001
	00111001
Sum	11100010
Checksum	00011101

The pattern sent is

```
10101001  00111001  00011101
                    Checksum
```

Example 8

Now suppose the receiver receives the pattern sent in Example 7 and there is no error.

```
10101001  00111001  00011101
```

When the receiver adds the three sections, it will get all 1s, which, after complementing, is all 0s and shows that there is no error.

	10101001	
	00111001	
	00011101	
Sum	11111111	
Complement	00000000	means that the pattern is OK.

Example 9

Now suppose there is a burst error of length 5 that affects 4 bits.

```
10101111  11111001  00011101
```

When the receiver adds the three sections, it gets

		10101111	
		11111001	
		00011101	
Result	1	11000101	
Carry		1	
Sum		11000110	
Complement		00111001	means that the pattern is corrupted.

Performance

The checksum detects all errors involving an odd number of bits as well as most errors involving an even number of bits. However, if one or more bits of a segment are

damaged and the corresponding bit or bits of opposite value in a second segment are also damaged, the sums of those columns will not change and the receiver will not detect a problem. If the last digit of one segment is a 0 and it gets changed to a 1 in transit, then the last 1 in another segment must be changed to a 0 if the error is to go undetected. In two-dimensional parity check, two 0s could both change to 1s without altering the parity because carries were discarded. Checksum retains all carries; so although two 0s becoming 1s would not alter the value of their own column, it would change the value of the next-higher column. But anytime a bit inversion is balanced by an opposite bit inversion in the corresponding digit of another data segment, the error is invisible.

10.3 ERROR CORRECTION

The mechanisms that we have discussed up to this point detect errors but do not correct them. **Error correction** can be handled in several ways. The two most common are error correction by retransmission and forward error correction.

Error Correction by Retransmission

In **error correction by retransmission,** when an error is discovered, the receiver can have the sender retransmit the entire data unit. This type of error correction is discussed along with flow and error control protocols in Chapter 11.

Forward Error Correction

In **forward error correction (FEC),** a receiver can use an error-correcting code, which automatically corrects certain errors. In theory, it is possible to correct any errors automatically. Error-correcting codes, however, are more sophisticated than error detection codes and require more redundancy bits.

The concept underlying error correction can be most easily understood by examining the simplest case: single-bit errors. As we saw earlier, single-bit errors can be detected by the addition of a redundant (parity) bit. A single additional bit can detect single-bit errors in any sequence of bits because it must distinguish between only two conditions: error or no error. A bit has two states (0 and 1). These two states are sufficient for this level of detection.

But what if we want to correct as well as detect single-bit errors? Two states are enough to detect an error but not to correct it. An error occurs when the receiver reads a 1 bit as a 0 or a 0 bit as a 1. To correct the error, the receiver simply reverses the value of the altered bit. To do so, however, it must know which bit is in error. The secret of error correction, therefore, is to locate the invalid bit or bits.

For example, to correct a single-bit error in an ASCII character, the error correction code must determine which of the 7 bits has changed. In this case, we have to distinguish between eight different states: no error, error in position 1, error in position 2,

and so on, up to error in position 7. To do so requires enough redundancy bits to show all eight states.

At first glance, it seems that a 3-bit redundancy code should be adequate because 3 bits can show eight different states (000 to 111) and can therefore indicate the locations of eight different possibilities. But what if an error occurs in the redundancy bits themselves? Seven bits of data (the ASCII character) plus 3 bits of redundancy equals 10 bits. Three bits, however, can identify only eight possibilities. Additional bits are necessary to cover all possible error locations.

To calculate the number of redundancy bits r required to correct a given number of data bits m, we must find a relationship between m and r. With m bits of data and r bits of redundancy added to them, the length of the resulting code is $m + r$.

If the total number of bits in a transmittable unit is $m + r$, then r must be able to indicate at least $m + r + 1$ different states. Of these, one state means no error, and $m + r$ states indicate the location of an error in each of the $m + r$ positions.

So $m + r + 1$ states must be discoverable by r bits; and r bits can indicate 2^r different states. Therefore, 2^r must be equal to or greater than $m + r + 1$:

$$2^r >= m + r + 1$$

The value of r can be determined by plugging in the value of m (the original length of the data unit to be transmitted). For example, if the value of m is 7 (as in a 7-bit ASCII code), the smallest r value that can satisfy this equation is 4:

$$2^4 >= 7 + 4 + 1$$

Table 10.2 shows some possible m values and the corresponding r values.

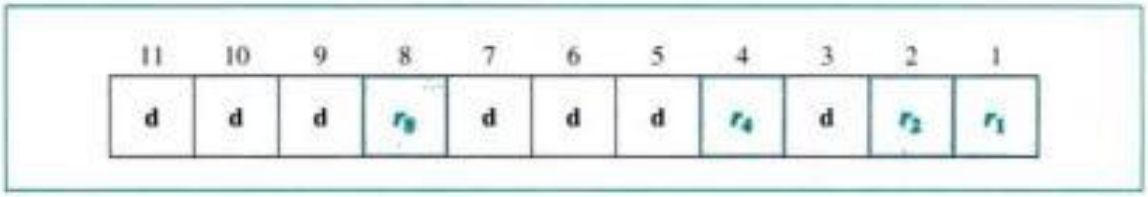
Table 10.2 *Relationship between data and redundancy bits*

Number of Data Bits m	*Number of Redundancy Bits* r	*Total Bits* $m+r$
1	2	3
2	3	5
3	3	6
4	3	7
5	4	9
6	4	10
7	4	11

Hamming Code

Hamming provides a practical solution. The **Hamming code** can be applied to data units of any length and uses the relationship between data and redundancy bits discussed above. For example, a 7-bit ASCII code requires 4 redundancy bits that can be added to the end of the data unit or interspersed with the original data bits. In Figure 10.14, these bits are placed in positions 1, 2, 4, and 8 (the positions in an 11-bit sequence that are powers of 2). For clarity in the examples below, we refer to these bits as r_1, r_2, r_4, and r_8.

Figure 10.14 *Positions of redundancy bits in Hamming code*

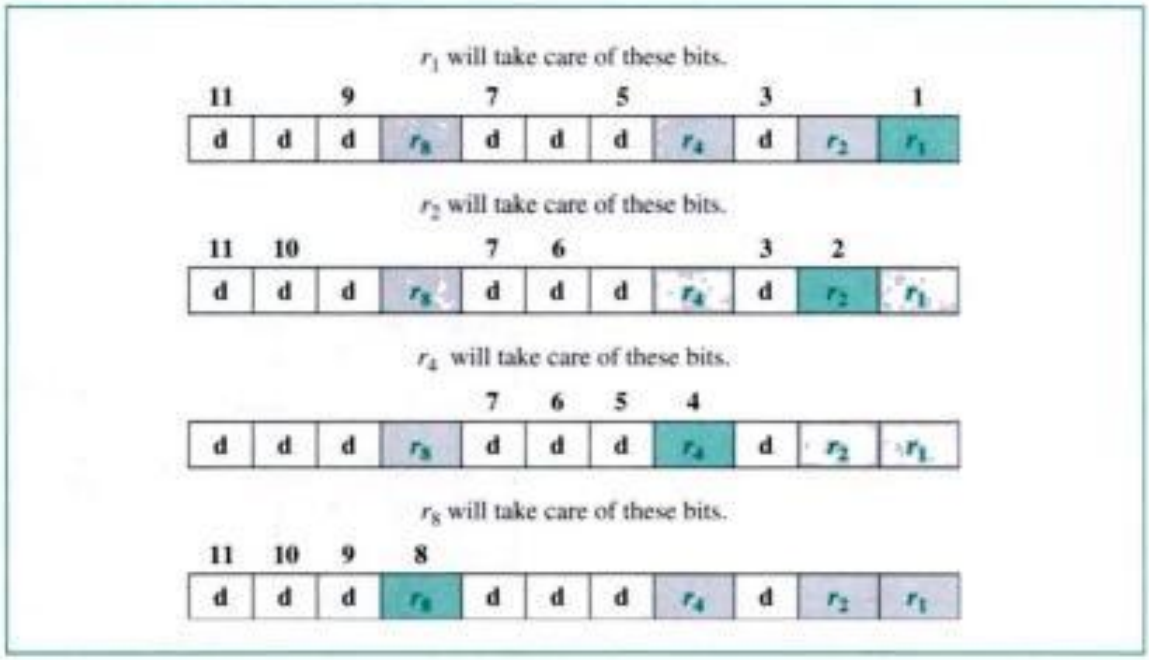


In the Hamming code, each r bit is the parity bit for one combination of data bits, as shown below:

r_1: bits 1, 3, 5, 7, 9, 11
r_2: bits 2, 3, 6, 7, 10, 11
r_4: bits 4, 5, 6, 7
r_8: bits 8, 9, 10, 11

Each data bit may be included in more than one calculation. In the sequences above, for example, each of the original data bits is included in at least two sets, while the r bits are included in only one (see Fig. 10.15).

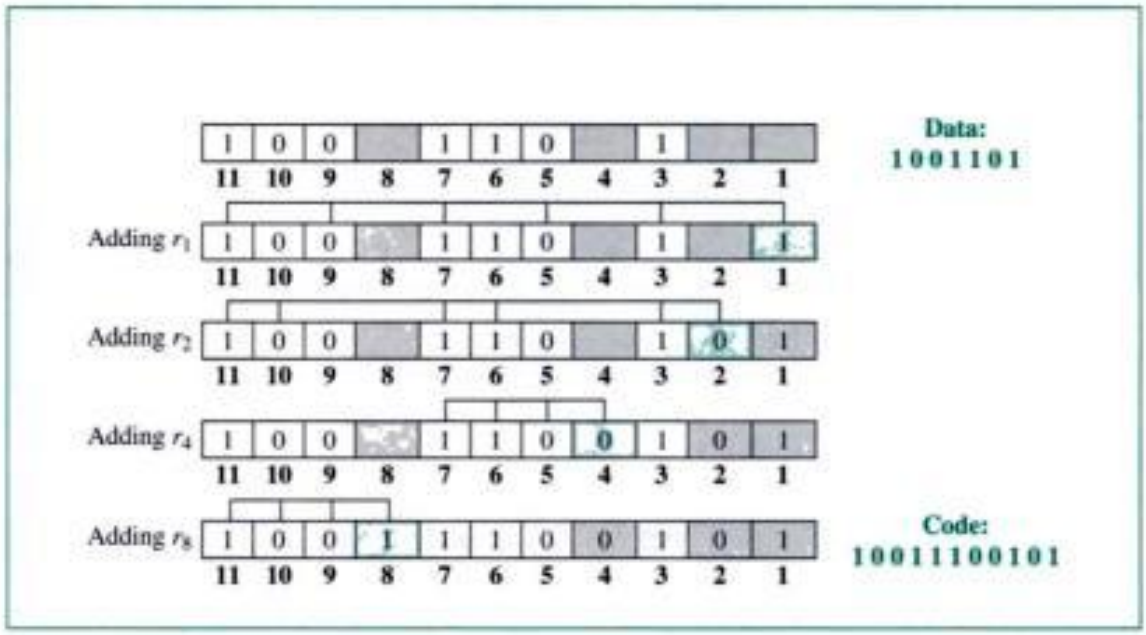
Figure 10.15 *Redundancy bits calculation*



Calculating the *r* Values Figure 10.16 shows a Hamming code implementation for an ASCII character. In the first step, we place each bit of the original character in its appropriate position in the 11-bit unit. In the subsequent steps, we calculate the even parities for the various bit combinations. The parity value for each combination is the value of the corresponding r bit.

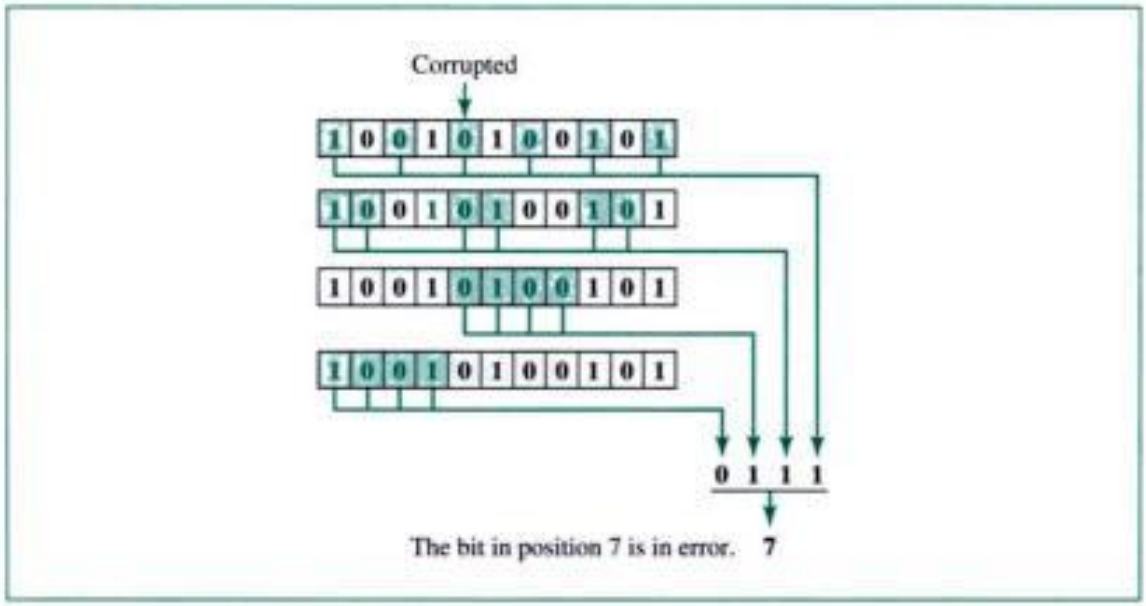
Error Detection and Correction Now imagine that by the time the above transmission is received, the number 7 bit has been changed from 1 to 0. The receiver takes the

Figure 10.16 *Example of redundancy bit calculation*



transmission and recalculates 4 new parity bits, using the same sets of bits used by the sender plus the relevant parity r bit for each set (see Fig. 10.17). Then it assembles the new parity values into a binary number in order of r position (r_8, r_4, r_2, r_1). In our example, this step gives us the binary number 0111 (7 in decimal), which is the precise location of the bit in error.

Figure 10.17 *Error detection using Hamming code*

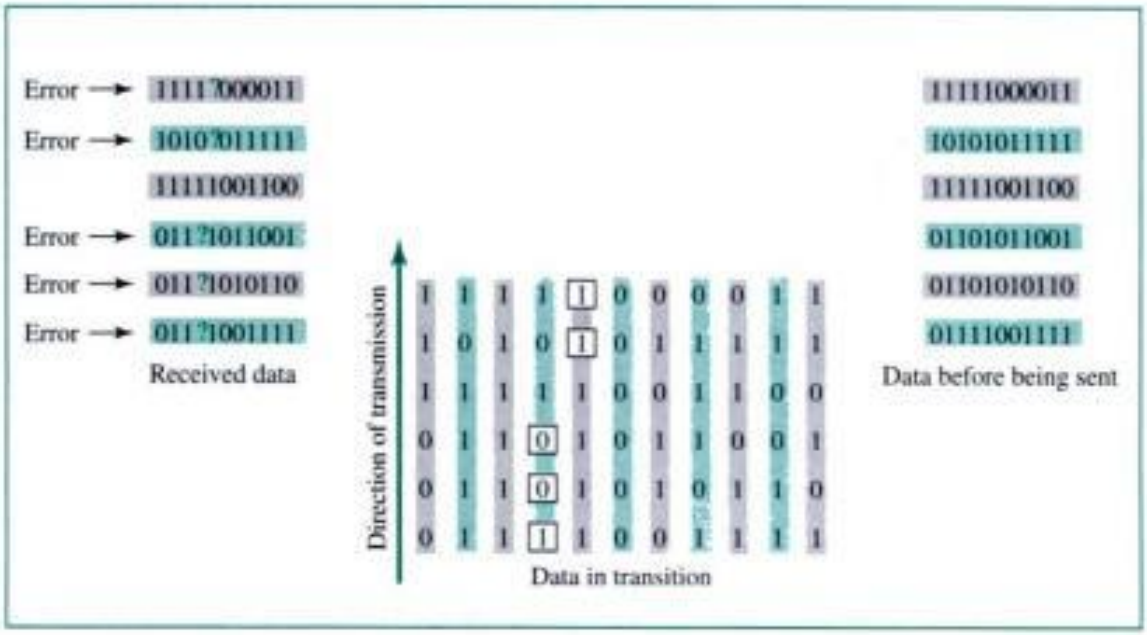


Once the bit is identified, the receiver can reverse its value and correct the error. The beauty of the technique is that it can easily be implemented in hardware and the code is corrected before the receiver knows about it.

Burst Error Correction

Although the Hamming code cannot correct a burst error directly, it is possible to rearrange the data and then apply the code. Instead of sending all the bits in a data unit together, we can organize N units in a column and then send the first bit of each, followed by the second bit of each, and so on. In this way, if a burst error of M bits occurs ($M < N$), then the error does not corrupt M bits of one single unit; it corrupts only 1 bit of a unit. With the Hamming scheme, we can then correct the corrupted bit in each unit. Figure 10.18 shows an example.

Figure 10.18 *Burst error correction example*



In Figure 10.18 we need to send six data units where each unit is a character with Hamming redundant bits. We organize the bits in columns and rows. We send the first column, then the second column, and so on. The bits that are corrupted by a burst error are shown in squares. Five consecutive bits are corrupted during the actual transmission. However, when these bits arrive at the destination and are reorganized into data units, each corrupted bit belongs to one unit and is automatically corrected. The trick here is to let the burst error corrupt only 1 bit of each unit.

10.4 KEY TERMS

burst error
checksum
CRC checker
CRC generator
cyclic redundancy check (CRC)
error
error correction
error correction by retransmission
error detection
even parity
forward error correction
Hamming code

odd parity
one's complement
parity bit
parity check
polynomial
redundancy
single-bit error
two-dimensional parity check

10.5 SUMMARY

- Errors can be categorized as a single-bit error or a burst error. A single-bit error has one bit error per data unit. A burst error has two or more bit errors per data unit.
- Redundancy is the concept of sending extra bits for use in error detection.
- Three common redundancy methods are parity check, cyclic redundancy check (CRC), and checksum.
- An extra bit (parity bit) is added to the data unit in the parity check.
- The parity check can detect only an odd number of errors; it cannot detect an even number of errors.
- In the two-dimensional parity check, a redundant data unit follows n data units.
- CRC, a powerful redundancy checking technique, appends a sequence of redundant bits derived from binary division to the data unit.
- The divisor in the CRC generator is often represented as an algebraic polynomial.
- Errors are corrected through retransmission and by forward error correction.
- The Hamming code is an error correction method using redundant bits. The number of bits is a function of the length of the data bits.
- In the Hamming code, for a data unit of m bits, use the formula $2^r >= m + r + 1$ to determine r, the number of redundant bits needed.
- By rearranging the order of bit transmission of the data units, the Hamming code can correct burst errors.

10.6 PRACTICE SET

Review Questions

1. How does a single-bit error differ from a burst error?
2. Discuss the concept of redundancy in error detection.
3. What are three types of redundancy checks used in data communications?
4. How can the parity bit detect a damaged data unit?
5. What is the difference between even parity and odd parity?
6. Discuss the parity check and the types of errors it can and cannot detect.
7. How is the simple parity check related to the two-dimensional parity check?
8. Discuss the two-dimensional parity check and the types of errors it can and cannot detect.

9. What does the CRC generator append to the data unit?
10. What is the relationship between the size of the CRC remainder and the divisor?
11. How does the CRC checker know that the received data unit is undamaged?
12. What are the conditions for the polynomial used by the CRC generator?
13. How is CRC superior to the two-dimensional parity check?
14. What is the error detection method used by upper-layer protocols?
15. What kind of arithmetic is used to add segments in the checksum generator and checksum checker?
16. List the steps involved in creating a checksum.
17. How does the checksum checker know that the received data unit is undamaged?
18. What kind of error is undetectable by the checksum?
19. What is the formula to calculate the number of redundancy bits required to correct a bit error in a given number of data bits?
20. What is the purpose of the Hamming code?
21. How can we use the Hamming code to correct a burst error?

Multiple-Choice Questions

22. Which error detection method consists of a parity bit for each data unit as well as an entire data unit of parity bits?
    a. Simple parity check
    b. Two-dimensional parity check
    c. CRC
    d. Checksum
23. Which error detection method uses ones complement arithmetic?
    a. Simple parity check
    b. Two-dimensional parity check
    c. CRC
    d. Checksum
24. Which error detection method consists of just one redundant bit per data unit?
    a. Simple parity check
    b. Two-dimensional parity check
    c. CRC
    d. Checksum
25. Which error detection method involves polynomials?
    a. Simple parity check
    b. Two-dimensional parity check
    c. CRC
    d. Checksum
26. Which of the following best describes a single-bit error?
    a. A single bit is inverted.
    b. A single bit is inverted per data unit.

c. A single bit is inverted per transmission.
d. Any of the above

27. If the ASCII character G is sent and the character D is received, what type of error is this?
    a. Single-bit
    b. Multiple-bit
    c. Burst
    d. Recoverable
28. If the ASCII character H is sent and the character I is received, what type of error is this?
    a. Single-bit
    b. Multiple-bit
    c. Burst
    d. Recoverable
29. In cyclic redundancy checking, what is the CRC?
    a. The divisor
    b. The quotient
    c. The dividend
    d. The remainder
30. In cyclic redundancy checking, the divisor is _____ the CRC.
    a. The same size as
    b. 1 bit less than
    c. 1 bit more than
    d. 2 bits more than
31. If the data unit is 111111, the divisor 1010, and the remainder 110, what is the dividend at the receiver?
    a. 111111011
    b. 111111110
    c. 1010110
    d. 110111111
32. If the data unit is 111111 and the divisor 1010, what is the dividend at the transmitter?
    a. 111111000
    b. 1111110000
    c. 111111
    d. 1111111010
33. If odd parity is used for ASCII error detection, the number of 0s per 8-bit symbol is _____.
    a. Even
    b. Odd
    c. Indeterminate
    d. 42

34. The sum of the checksum and data at the receiver is _____ if there are no errors.
    a. –0
    b. +0
    c. The complement of the checksum
    d. The complement of the data
35. The Hamming code is a method of _____.
    a. Error detection
    b. Error correction
    c. Error encapsulation
    d. (a) and (b)
36. In CRC there is no error if the remainder at the receiver is _____.
    a. Equal to the remainder at the sender
    b. Zero
    c. Nonzero
    d. The quotient at the sender
37. In CRC the quotient at the sender _____.
    a. Becomes the dividend at the receiver
    b. Becomes the divisor at the receiver
    c. Is discarded
    d. Is the remainder
38. Which error detection method involves the use of parity bits?
    a. Simple parity check
    b. Two-dimensional parity check
    c. CRC
    d. (a) and (b)
39. Which error detection method can detect a single-bit error?
    a. Simple parity check
    b. Two-dimensional parity check
    c. CRC
    d. All the above
40. Which error detection method can detect a burst error?
    a. The parity check
    b. Two-dimensional parity check
    c. CRC
    d. (b) and (c)
41. At the CRC generator, _____ added to the data unit before the division process.
    a. 0s are
    b. 1s are
    c. A polynomial is
    d. A CRC remainder is

42. At the CRC generator, _____ added to the data unit after the division process.
    a. 0s are
    b. 1s are
    c. The polynomial is
    d. The CRC remainder is
43. At the CRC checker, _____ means that the data unit is damaged.
    a. A string of 0s
    b. A string of 1s
    c. A string of alternating 1s and 0s
    d. A nonzero remainder

Exercises

44. What is the maximum effect of a 2-ms burst of noise on data transmitted at
    a. 1500 bps?
    b. 12,000 bps?
    c. 96,000 bps?
45. Assuming even parity, find the parity bit for each of the following data units.
    a. 1001011
    b. 0001100
    c. 1000000
    d. 1110111
46. A receiver receives the bit pattern 01101011. If the system is using even parity, is the pattern in error?
47. A system uses two-dimensional parity. Find the parity unit for the following two data units. Assume even parity.
    10011001 01101111
48. Given a 10-bit sequence 1010011110 and a divisor of 1011, find the CRC. Check your answer.
49. Given a remainder of 111, a data unit of 10110011, and a divisor of 1001, is there an error in the data unit?
50. Find the checksum for the following bit sequence. Assume a 16-bit segment size.
    1001001110010011
    1001100001001101
51. Find the complement of 1110010001110011.
52. Add 11100011 and 00011100 in ones complement. Interpret the result.
53. For each data unit of the following sizes, find the minimum number of redundancy bits needed to correct one single-bit error.
    a. 12
    b. 16
    c. 24
    d. 64

54. Construct the Hamming code for the bit sequence 10011101.
55. Find the parity bits for the following bit pattern, using simple parity. Do the same for two-dimensional parity. Assume even parity.

    ← 0011101 1100111 1111111 0000000
56. A sender sends 01110001; the receiver receives 01000001. If simple parity is used, can the receiver detect the error?
57. The following block is received by a system using two-dimensional even parity. Which bits are in error?

    ← 10010101 01001111 11010000 11011011
58. A system using two-dimensional even parity sends a block of 8 bytes. How many redundant bits are sent per block? What is the ratio of useful bits to total bits?
59. If a divisor is 101101, how many bits long is the CRC?
60. Find the binary equivalent of $x^8 + x^3 + x + 1$.
61. Find the polynomial equivalent of 100001110001.
62. A receiver receives the code 11001100111. When it uses the Hamming encoding algorithm, the result is 0101. Which bit is in error? What is the correct code?
63. In single-bit error correction, a code of 3 bits can be in one of four states: no error, first bit in error, second bit in error, and third bit in error. How many of these 3 bits should be redundant to correct this code? How many bits can be the actual data?
64. Using the logic in Exercise 63, find out how many redundant bits should be in a 10-bit code to detect an error.
65. The code 11110101101 was received. Using the Hamming encoding algorithm, what is the original code sent?

CHAPTER 11

Data Link Control and Protocols

Data communication requires at least two devices working together, one to send and one to receive. Even such a basic arrangement requires a great deal of coordination for an intelligible exchange to occur. The most important responsibilities of the data link layer are **flow control** and **error control.** Collectively, these functions are known as **data link control.**

In this chapter, we first informally define flow and error control. We then introduce three mechanisms that handle flow and error control. We finally discuss a popular data link protocol, HDLC.

11.1 FLOW AND ERROR CONTROL

Flow and error control are the main functions of the data link layer. Let us informally define each.

Flow Control

Flow control coordinates the amount of data that can be sent before receiving acknowledgment and is one of the most important duties of the data link layer. In most protocols, flow control is a set of procedures that tells the sender how much data it can transmit before it must wait for an acknowledgment from the receiver. The flow of data must not be allowed to overwhelm the receiver. Any receiving device has a limited speed at which it can process incoming data and a limited amount of memory in which to store incoming data. The receiving device must be able to inform the sending device before those limits are reached and to request that the transmitting device send fewer frames or stop temporarily. Incoming data must be checked and processed before they can be used. The rate of such processing is often slower than the rate of transmission. For this reason, each receiving device has a block of memory, called a *buffer,* reserved for storing incoming data until they are processed. If the buffer begins to fill up, the receiver must be able to tell the sender to halt transmission until it is once again able to receive.