UNIT-II

CHAPTER 9

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 source to destination, they can become corrupted in passage. In fact, it is more likely that some part of a message will be altered in transit than that the entire contents will arrive intact. Many factors, including line noise, 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.

Error detection and correction are implemented either at the data link layer or the transport layer of the OSI model.

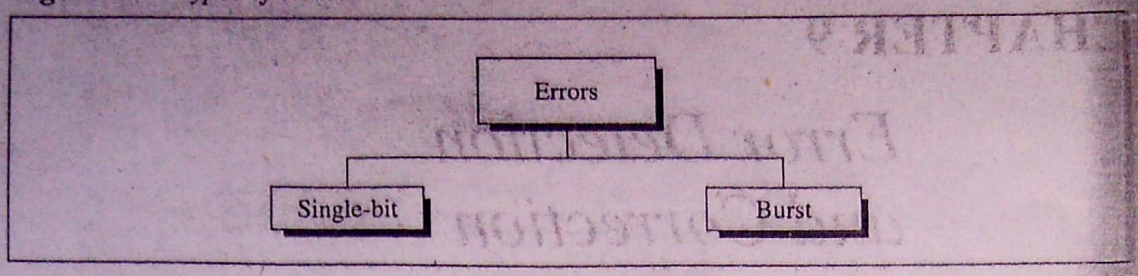
9.1 TYPES OF ERRORS

Whenever an electromagnetic signal flows from one point to another, it is subject to unpredictable interference from heat, magnetism, and other forms of electricity. This interference can change the shape or timing of the signal. If the signal is carrying encoded binary data, such changes can alter the meaning of the data. 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-second burst of impulse noise on a transmission with a data rate of 1200 bps might change all or some of 12 bits of information (see Figure 9.1).

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.

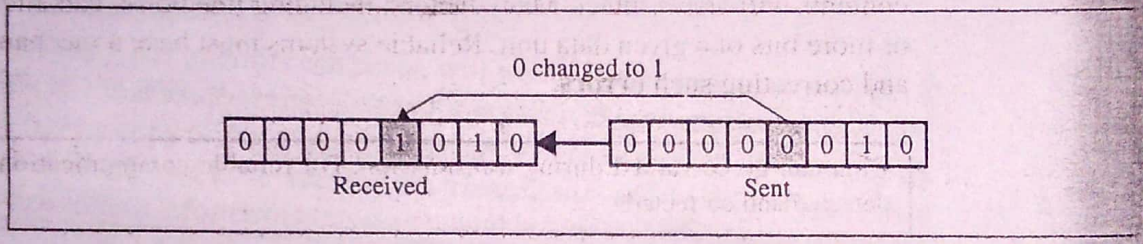
Figure 9.1 *Types of errors*



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

Figure 9.2 shows the effect of a single-bit error on a data unit. To understand the impact of the change, imagine that each group of eight 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 9.2 *Single-bit error*



Single-bit errors are the least likely type of error in serial data transmission. To see why, imagine a sender sends data at 1Mbps. This means that each bit lasts only 1/1,000,000 second, 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 of the eight bits of a 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.

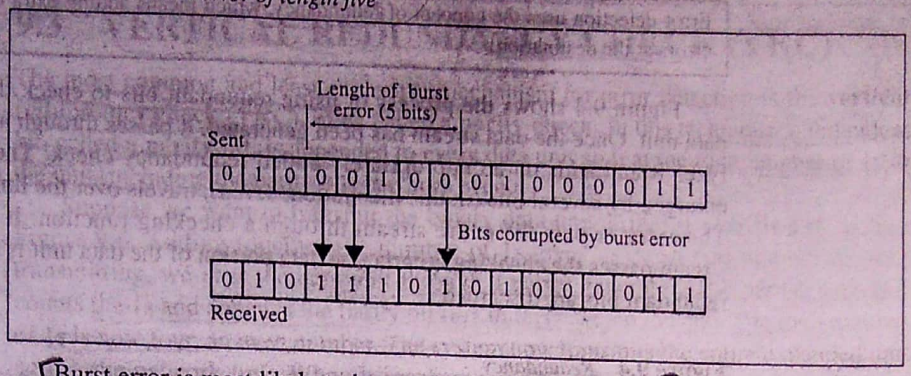
Burst Error

The term **burst error** means that two or more bits in the data unit have changed from 1 to 0 or from 0 to 1.

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

Figure 9.3 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 9.3 *Burst error of length five*



Burst error is most likely to happen in a serial transmission. The duration of noise is normally longer than the duration of a 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 seconds can affect 10 bits; if we are sending data at 1 Mbps, the same noise can affect 10,000 bits.

9.2 DETECTION

Even if we know what types of errors can occur, will we recognize one when we see it? If we have a copy of the intended transmission for comparison, of course we will. But what if we don't have a copy of the original? Then we will have no way of knowing we have received an error until we have decoded the transmission and failed to make sense of it. For a machine to check for errors this way would be slow, costly, and of questionable value. We don't need a system where computers decode whatever comes in, then sit around trying to decide if the sender really meant to use the word *glbrshnif* in the middle of an array of weather statistics. What we need is a mechanism that is simple and completely objective.

Redundancy

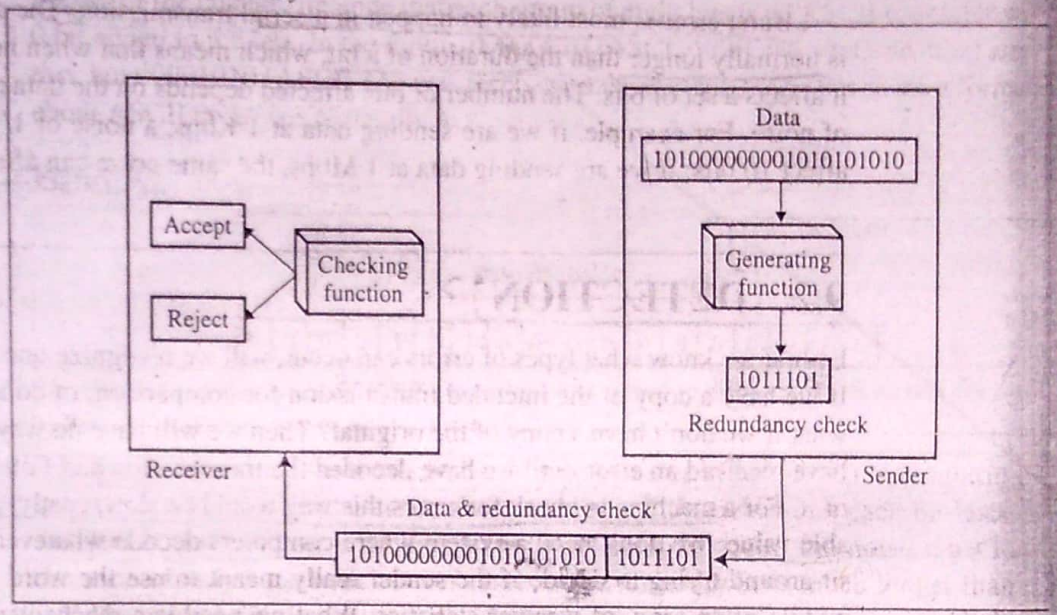
One **error detection** mechanism that would satisfy these requirements would be to send every data unit twice. The receiving device would then be able to do a bit-for-bit comparison between the two versions of the data. Any discrepancy would indicate an error, and an appropriate correction mechanism 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 the time it takes to compare every unit bit by bit must be added.

The concept of including extra information in the transmission solely for the purposes of comparison 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.

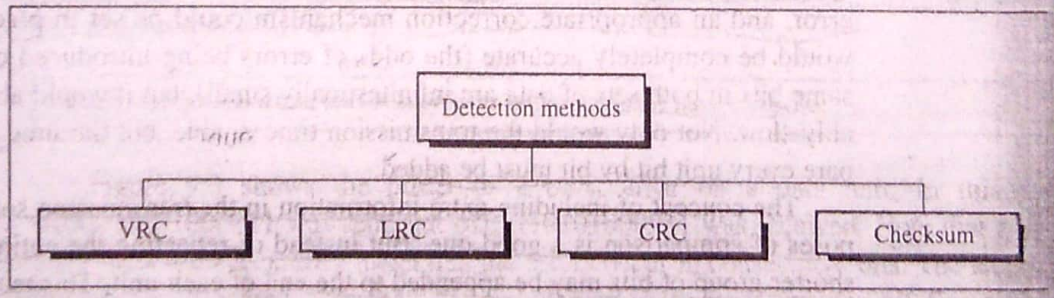
Figure 9.4 shows the process of using redundant bits to check the accuracy of a data unit. Once the data stream has been generated, it passes through a device that analyzes it and adds on an appropriately coded redundancy check. The data unit, now enlarged by several bits (in this illustration, seven), travels over the link to the receiver. The receiver puts 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.

Figure 9.4 *Redundancy*



Four types of redundancy checks are used in data communications: vertical redundancy check (VRC) (also called parity check), longitudinal redundancy check (LRC), cyclical redundancy check (CRC), and checksum. The first three, VRC, LRC, and CRC, are normally implemented in the physical layer for use in the data link layer. The fourth, checksum, is used primarily by upper layers (see Figure 9.5).

Figure 9.5 *Detection methods*



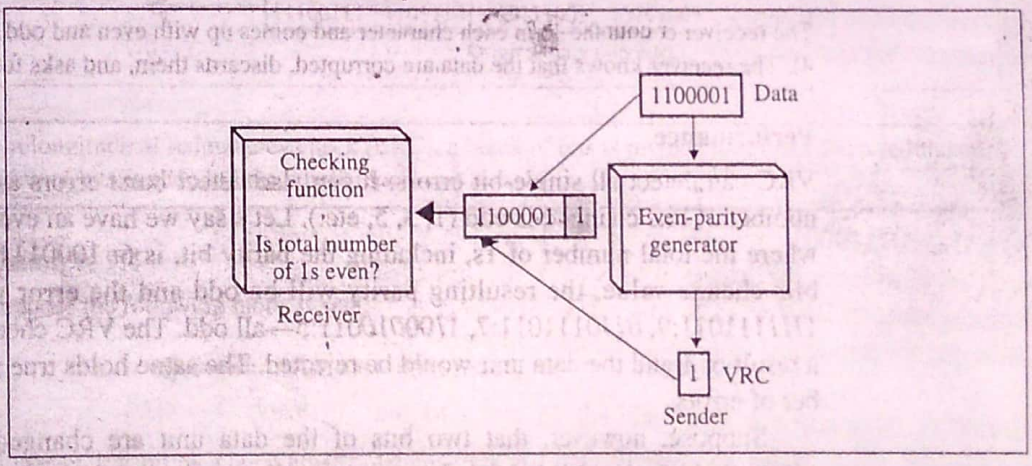
9.3 VERTICAL REDUNDANCY CHECK (VRC)

The most common and least expensive mechanism for error detection is the **vertical redundancy check (VRC)**, often called a **parity check**. In this technique, a redundant bit, called a **parity bit**, is appended to every data unit so that the total number of 1s in the unit (including the parity bit) becomes even.

Suppose we want to transmit the binary data unit 1100001 [ASCII *a* (97)]; see Figure 9.6. Adding together 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 four, an even number. The system now transmits the entire expanded unit across the network link. When it reaches its destination, the receiver puts all eight bits through an **even-parity** checking function. If the receiver sees 11100001, 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 11100001, the receiver sees 11100101? 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.

> In vertical redundancy check (VRC), a parity bit is added to every data unit so that the total number of 1s becomes even.

Figure 9.6 *Even parity VRC concept*



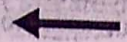
Note that for the sake of simplicity, we are discussing here even-parity checking, where the number of 1s should be an even number. Some systems may use **odd-parity** checking, where the number of 1s should be odd. The principle is the same; the calculation is different.

Example 9.1

Imagine 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 9.2

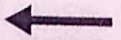
Now suppose the word "world," in the previous example, is received by the receiver without being corrupted in transmission.

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 would be accepted.

Example 9.3

Now suppose the word "world," in Example 9.1, is received by the receiver but corrupted during transmission.



11*1*1110 11011110 1110*1*100 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

VRC 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 three bits change value, the resulting parity will be odd and the error will be detected: 1*111*111011:9, *011*0111011:7, 1*100010*011:5—all odd. The VRC checker would return a result of 1 and the data unit would be rejected. The same holds true for any odd number of errors.

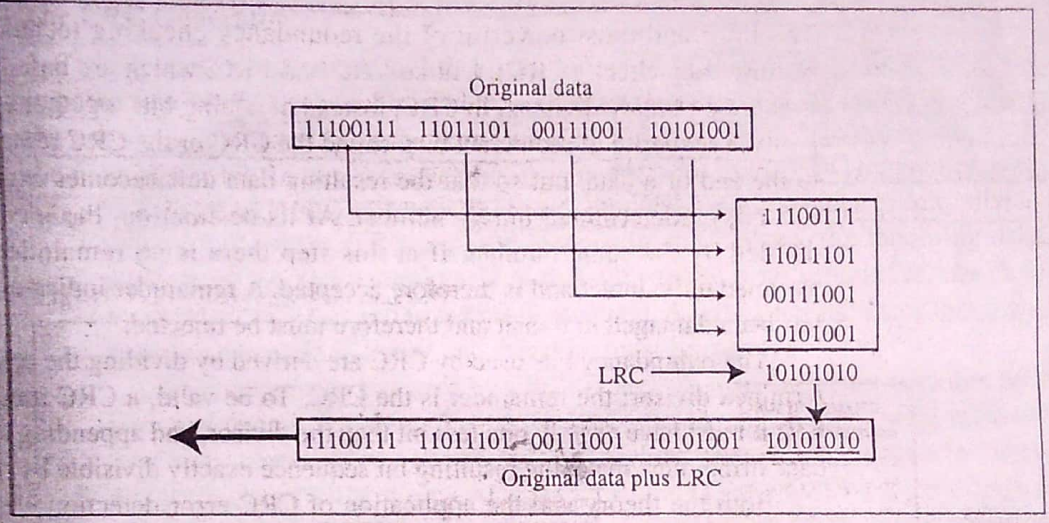
Suppose, however, that two bits of the data unit are changed: 1*11*0111011:8, 1*100*011011:6, 1000011010:4. In each case the number of 1s in the data unit is still even. The VRC checker will add them and return an even number although the data unit contains two errors. VRC cannot detect errors where 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.

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

9.4 LONGITUDINAL REDUNDANCY CHECK (LRC)

In longitudinal redundancy check (LRC), a block of bits is organized in a table (rows and columns). For example, instead of sending a block of 32 bits, we organize them in a table made of four rows and eight columns, as shown in Figure 9.7. We then calculate the parity bit for each column and create a new row of eight bits, which are the parity bits for the whole block. Note that the first parity bit in the fifth row is calculated based on all first bits. The second parity bit is calculated based on all second bits, and so on. We then attach the eight parity bits to the original data and send them to the receiver.

Figure 9.7 *LRC*



In longitudinal redundancy check (LRC), a block of bits is divided into rows and a redundant row of bits is added to the whole block.

Example 9.4

Suppose the following block is sent:

10101001 00111001 11011101 11100111 10101010
(LRC)

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

10100011 10001001 11011101 11100111 10101010
(LRC)

When the receiver checks the LRC, 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**
(LRC)

Performance

LRC increases the likelihood of detecting burst errors. As we showed in the previous example, an LRC of n bits can easily detect a burst error of n bits. A burst error of more than n bits is also detected by LRC with a very high probability. There is, however, one pattern of errors that remains elusive. If two bits in one data unit are damaged and two bits *in exactly the same positions* in another data unit are also damaged, the LRC 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 and *0*1000011, the errors cannot be detected by LRC.

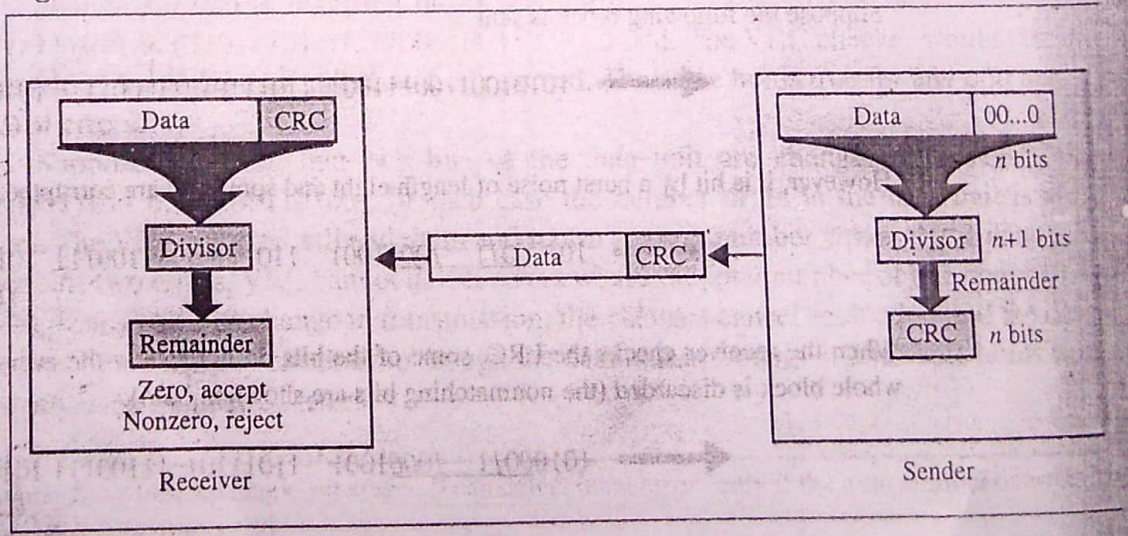
9.5 CYCLIC REDUNDANCY CHECK (CRC)

The third and most powerful of the redundancy checking techniques is the **cyclic redundancy check (CRC)**. Unlike VRC and LRC, which are based on addition, CRC is based on binary division. In CRC, instead of adding bits together 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. In order to clarify this process, we will start with an overview and add complexity as we go. Figure 9.8 provides an outline of the three basic steps.

Figure 9.8 *CRC generator and checker*



First, a string of n 0s is appended to the data unit. The number n is one less than the 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 process 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 end of the data unit. Note that the CRC may consist of all 0s.

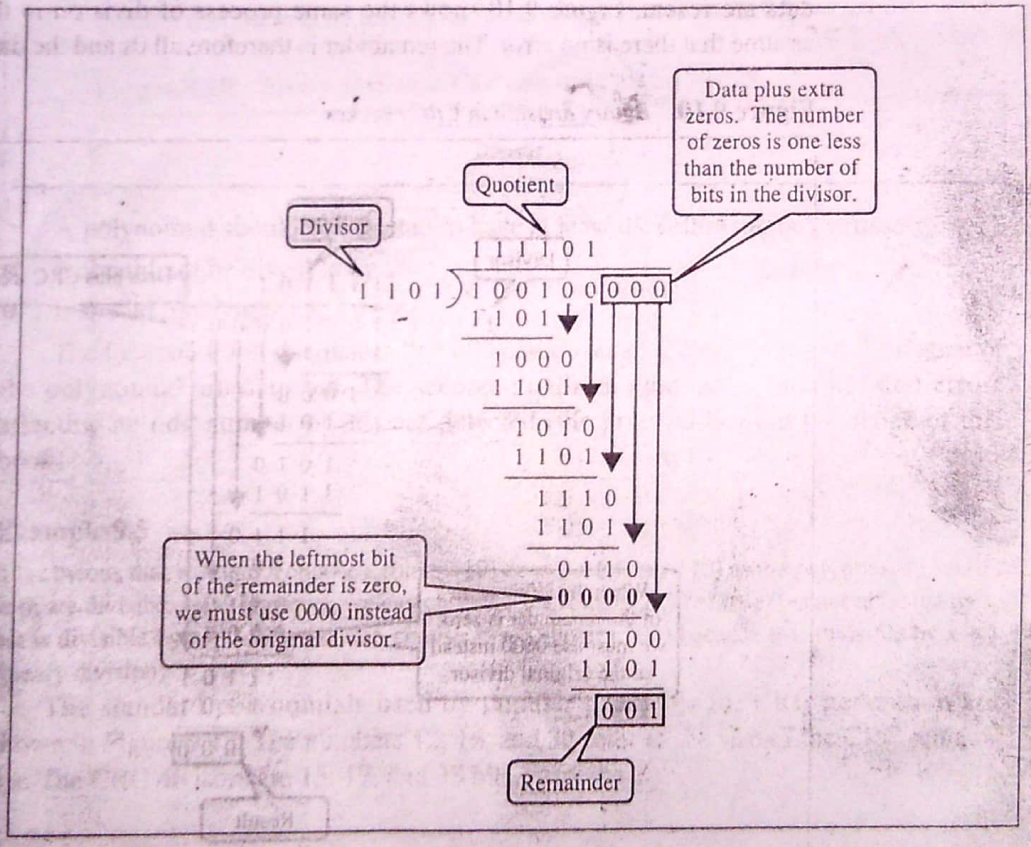
The data unit arrives at the receiver data first, followed by the CRC. The receiver treats the whole string as a unit and divides it by the same divisor that was used to find the CRC remainder.

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

The CRC Generator

A CRC generator uses modulo-2 division. Figure 9.9 shows this process. In the first step, the four-bit divisor is subtracted from the first four bits of the dividend. Each bit of the divisor is subtracted from the corresponding bit of the dividend without disturbing the next higher bit. In our example, the divisor, 1101, is subtracted from the first four bits of the dividend, 1001, yielding 100 (the leading 0 of the remainder is dropped off).

Figure 9.9 *Binary division in a CRC generator*



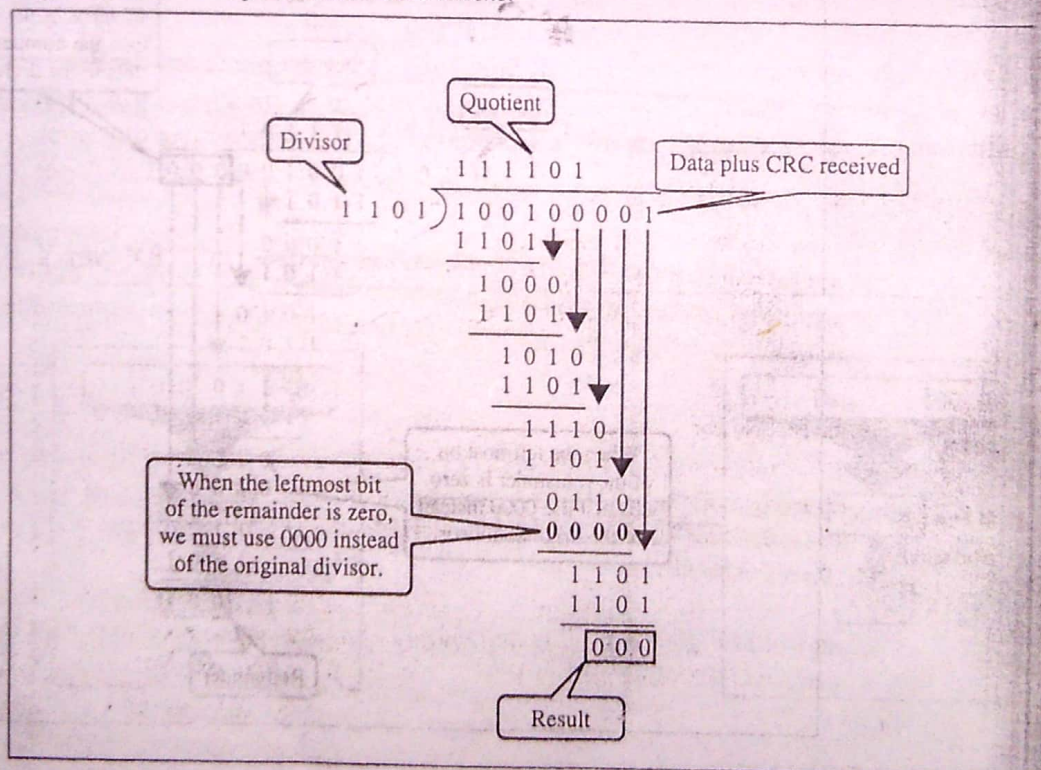
The next unused bit from the dividend is then pulled down to make the number of bits in the remainder equal to the number of bits in the divisor. The next step, therefore, is 1000 – 1101, which yields 101, and so on.

In this process, the divisor always begins with a 1; the divisor is subtracted from a portion of the previous dividend/remainder that is equal to it in length; the divisor can only be subtracted from a dividend/remainder whose leftmost bit is 1. Anytime the leftmost bit of the dividend/remainder is 0, a string of 0s, of the same length as the divisor, replaces the divisor in that step of the process. For example, if the divisor is four bits long, it is replaced by four 0s. (Remember, we are dealing with bit patterns, not with quantitative values; 0000 is not the same as 0.) This restriction means that, at any step, the leftmost 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 off, 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 like the generator. 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 accepted; otherwise, the received stream of bits is discarded and data are resent. Figure 9.10 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.

Figure 9.10 *Binary division in CRC checker*



Polynomials

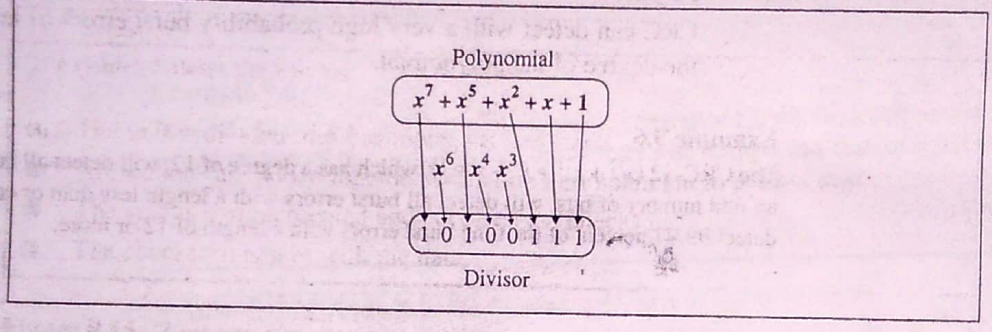
The CRC generator (the divisor) is most often represented not as a string of 1s and 0s, but as an algebraic polynomial (see Figure 9.11). The polynomial format is useful for two reasons: It is short, and it can be used to prove the concept mathematically (which is beyond the scope of this book).

Figure 9.11 *A polynomial*

$$x^7 + x^5 + x^2 + x + 1$$

The relationship of a polynomial to its corresponding binary representation is shown in Figure 9.12.

Figure 9.12 *A polynomial representing a divisor*



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

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

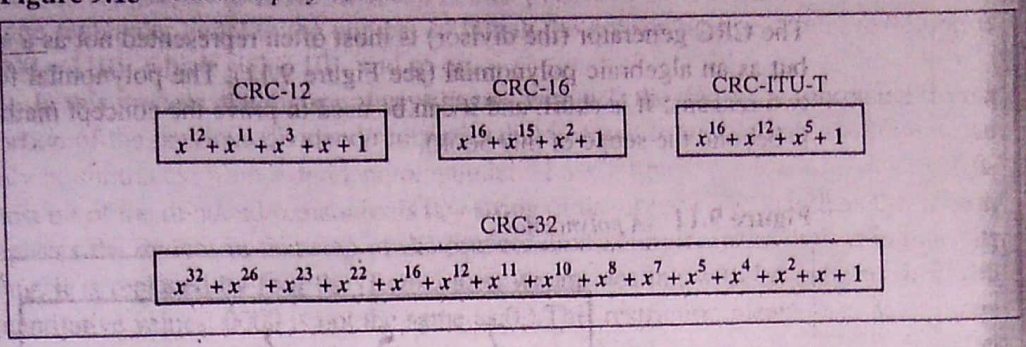
The first condition guarantees that all burst errors of a length equal to the degree of the polynomial are detected. The second condition guarantees that all burst errors affecting an odd number of bits are detected (the proof is beyond the scope of this book).

Example 9.5

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).

The standard polynomials used by popular protocols for CRC generation are shown in Figure 9.13. The numbers 12, 16, and 32 refer to the size of the CRC remainder. The CRC divisors are 13, 17, and 33 bits, respectively.

Figure 9.13 *Standard polynomials*



Performance

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

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

Example 9.6

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.

9.6 CHECKSUM

The error detection method used by the higher-layer protocols is called **checksum**. Like VRC, LRC, and CRC, checksum is based on the concept of redundancy.

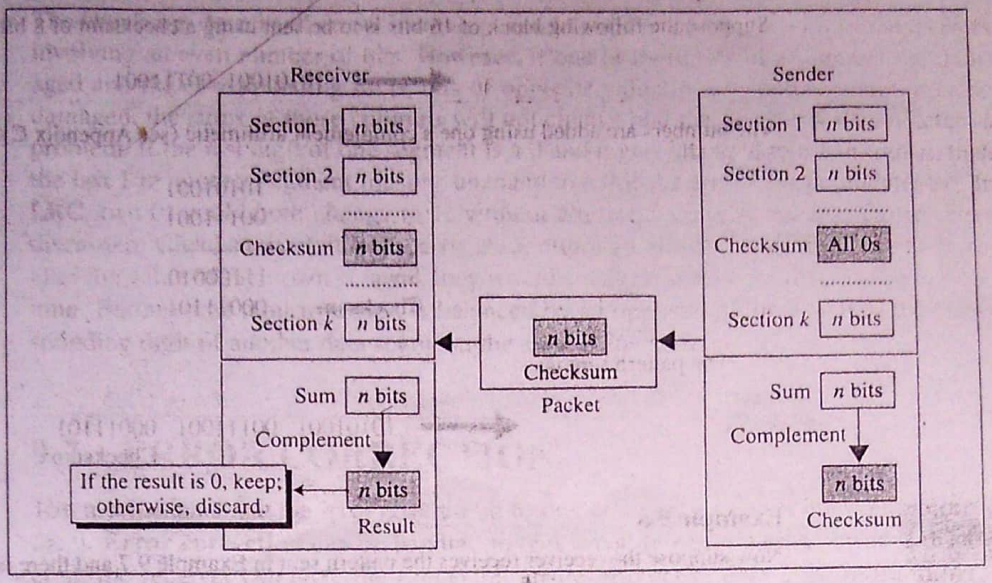
Checksum Generator

In the sender, the checksum generator subdivides the data unit into equal segments of *n* bits (usually 16). These segments are added together using **one's complement** arithmetic (see Appendix C) 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 Figures 9.14 and 9.15).

Checksum Checker

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

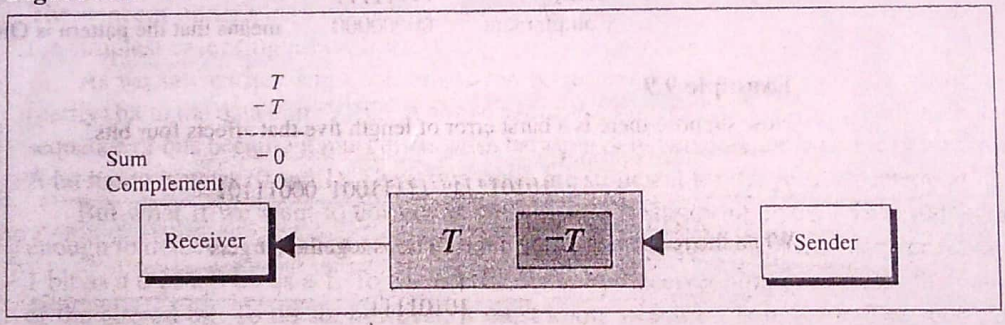
Figure 9.14 *Checksum*



The sender follows these steps:

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

Figure 9.15 *Data unit and checksum*

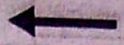


The receiver follows these steps:

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

Example 9.7

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



10101001 00111001

The numbers are added using one's complement arithmetic (see Appendix C).

```
                10101001
                00111001
              ----------
Sum             11100010
Checksum        00011101
```

The pattern sent is:

10101001 00111001 00011101
Checksum

Example 9.8

Now suppose the receiver receives the pattern sent in Example 9.7 and there is no error.

10101001 00111001 00011101

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

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

Example 9.9

Now suppose there is a burst error of length five that affects four bits.

10101*111* *11*111001 00011101

When the receiver adds the three sections together, 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 LRC, 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, they 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.

9.7 ERROR CORRECTION

The mechanisms that we have covered up to this point detect errors but do not correct them. **Error correction** can be handled in two ways. In one, when an error is discovered, the receiver can have the sender retransmit the entire data unit. In the other, a receiver can use an error-correcting code, which automatically corrects certain errors.

In theory, it is possible to correct any binary code errors automatically. Error-correcting codes, however, are more sophisticated than error-detection codes and require more redundancy bits. The number of bits required to correct a multiple-bit or burst error is so high that in most cases it is inefficient to do so. For this reason, most error correction is limited to one-, two-, or three-bit errors.

Single-Bit Error Correction

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 to the data unit (VRC). 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 seven 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.

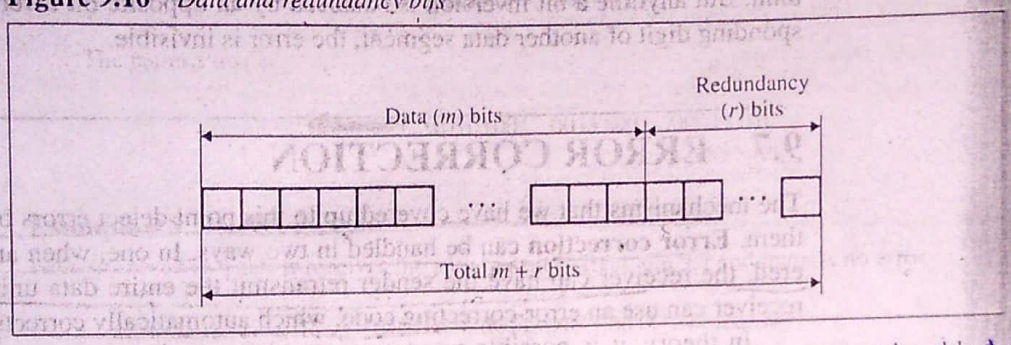
At first glance, it looks like a three-bit redundancy code should be adequate because three bits can show eight different states (000 to 111) and can therefore

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 three bits of redundancy equals 10 bits. Three bits, however, can identify only eight possibilities. Additional bits are necessary to cover all possible error locations.

Redundancy Bits

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. Figure 9.16 shows m bits of data with r bits of redundancy added to them. The length of the resulting code is $m + r$.

Figure 9.16 *Data and redundancy bits*



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 $m + r + 1$:

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

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 seven-bit ASCII code), the smallest r value that can satisfy this equation is 4:

$$2^4 \geq 7 + 4 + 1$$

Table 9.1 shows some possible m values and the corresponding r values.

Table 9.1 *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

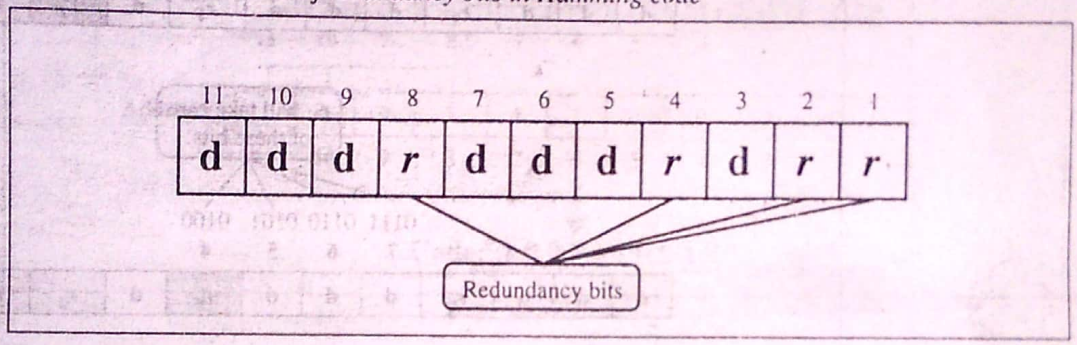
Hamming Code

So far, we have examined the number of bits required to cover all of the possible single-bit error states in a transmission. But how do we manipulate those bits to discover which state has occurred? A technique developed by R. W. Hamming provides a practical solution.

Positioning the Redundancy Bits

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 seven-bit ASCII code requires four redundancy bits that can be added to the end of the data unit or interspersed with the original data bits. In Figure 9.17, 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 9.17 *Positions of redundancy bits in Hamming code*



In the Hamming code, each r bit is the VRC bit for one combination of data bits: r_1 is the VRC bit for one combination of data bits, r_2 is the VRC bit for another combination of data bits, and so on. The combinations used to calculate each of the four r values for a seven-bit data sequence are as follows:

r_1: bits 1, 3, 5, 7, 9, 11

r_2: bits 2, 3, 6, 7, 10, 11

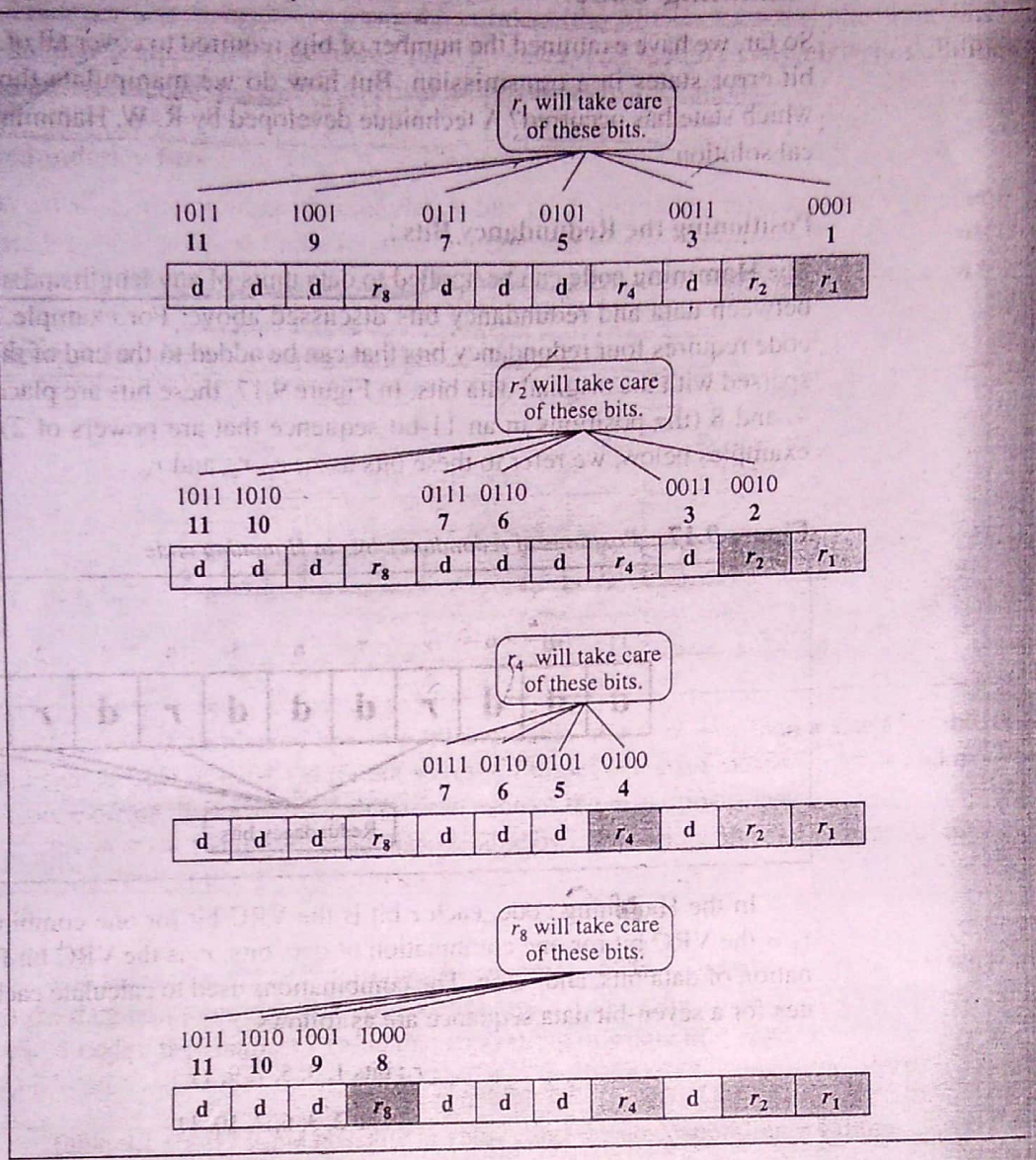
r_4: bits 4, 5, 6, 7

r_8: bits 8, 9, 10, 11

Each data bit may be included in more than one VRC 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.

To see the pattern behind this strategy, look at the binary representation of each bit position. The r_1 bit is calculated using all bit positions whose binary representation includes a 1 in the rightmost position. The r_2 bit is calculated using all bit positions with a 1 in the second position, and so on (see Figure 9.18).

Figure 9.18 *Redundancy bits calculation*



Calculating the *r* Values

Figure 9.19 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. For example, the value of r_1 is calculated to provide even parity for a combination of bits 3, 5, 7, 9, and 11. The value of r_2 is calculated to provide even parity with bits 3, 6, 7, 10, and 11, and so on. The final 11-bit code is sent through the transmission line.

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 (see Figure 9.20).

Figure 9.19 *Example of redundancy bit calculation*

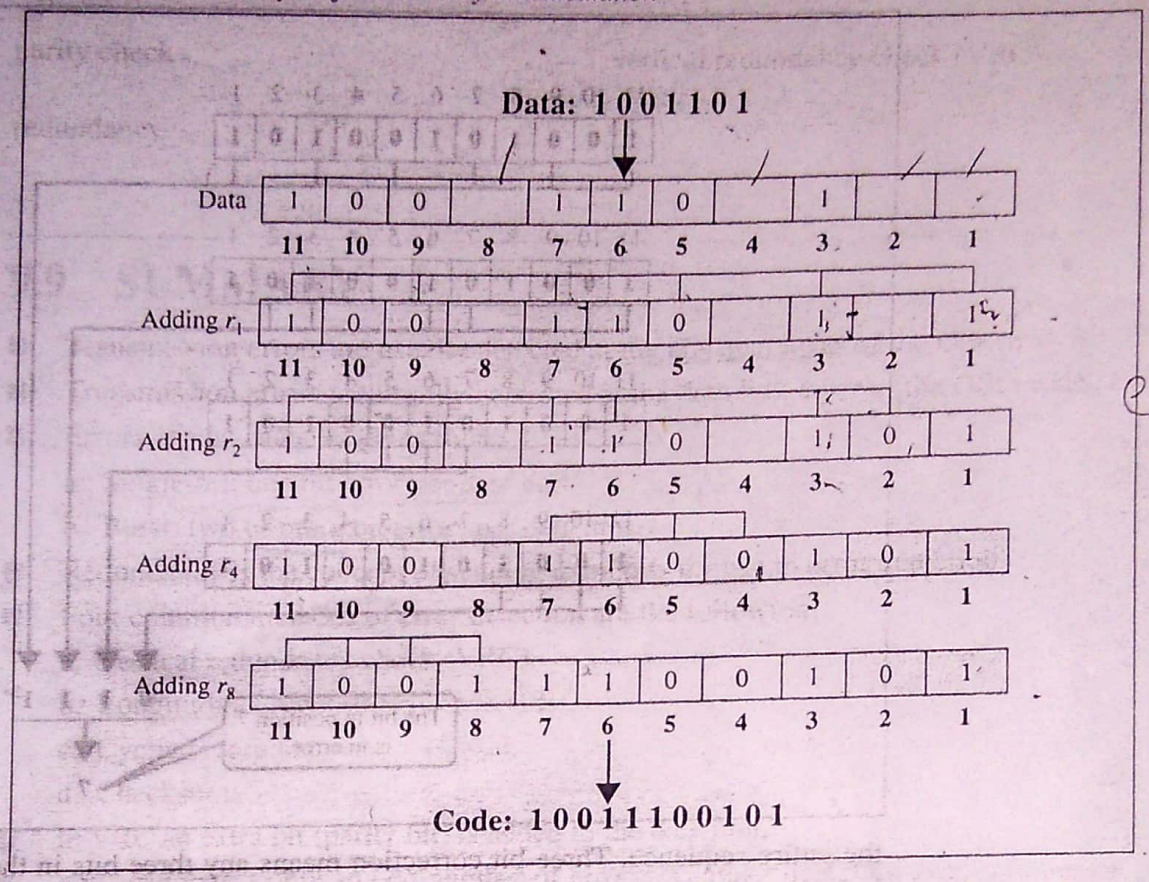
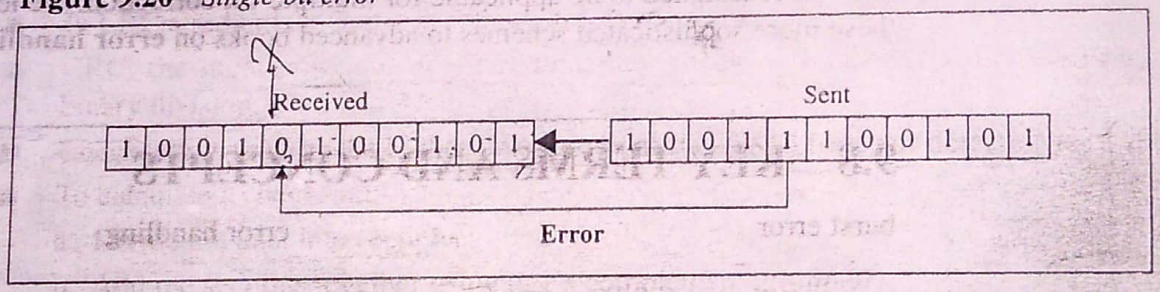


Figure 9.20 *Single-bit error*



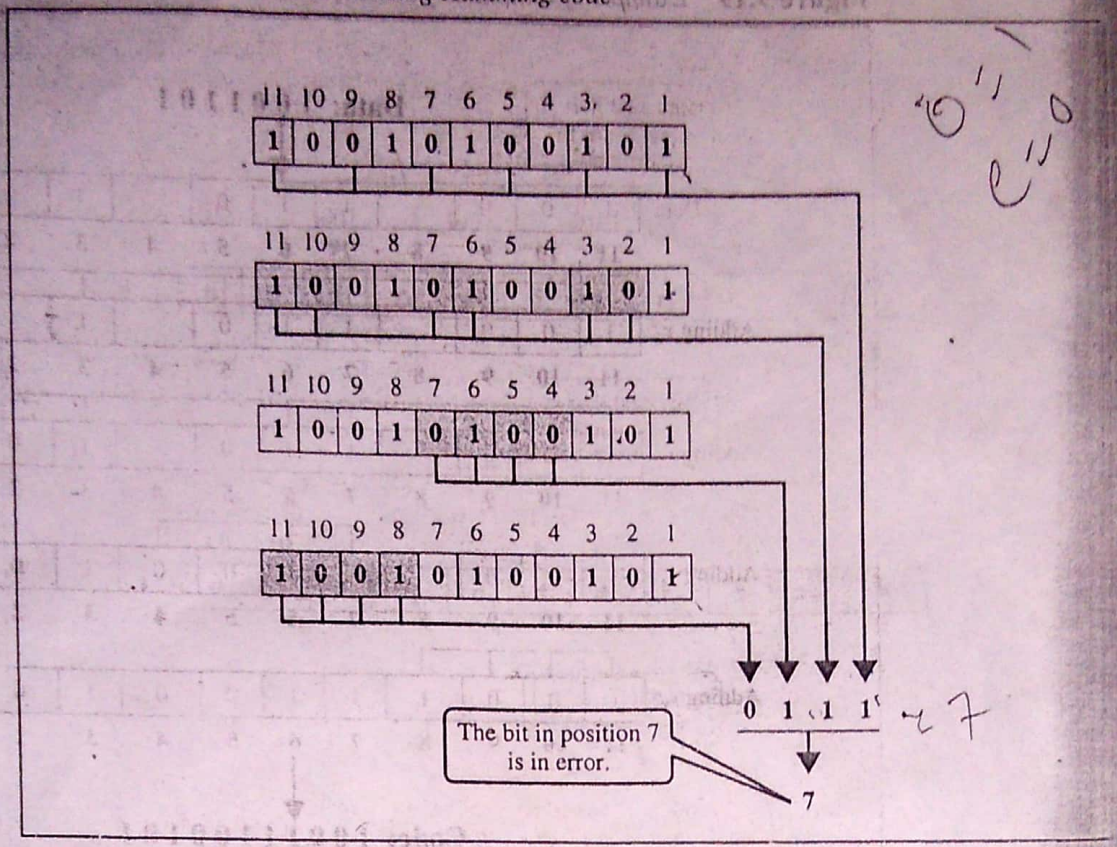
The receiver takes the transmission and recalculates four new VRCs using the same sets of bits used by the sender plus the relevant parity (r) bit for each set (see Figure 9.21). Then it assembles the new parity values into a binary number in order of r position (r_8, r_4, r_2, r_1). In our example, this step gives us the binary number 0111 (7 in decimal), which is the precise location of the bit in error.

Once the bit is identified, the receiver can reverse its value and correct the error.

Burst Error Correction

A Hamming code can be designed to correct burst errors of certain lengths. The number of redundancy bits required to make these corrections, however, is dramatically higher than that required for single-bit errors. To correct double-bit errors, for example, we must take into consideration that the two bits can be a combination of any two bits in

Figure 9.21 *Error detection using Hamming code*



the entire sequence. Three-bit correction means any three bits in the entire sequence and so on. So the simple strategy used by the Hamming code to correct single-bit errors must be redesigned to be applicable for multiple-bit correction. We leave the details of these more sophisticated schemes to advanced books on **error handling.**

9.8 KEY TERMS AND CONCEPTS

burst error

checksum

cyclic redundancy check (CRC)

error

error correction

error detection

error handling

even parity

Hamming code

longitudinal redundancy check (LRC)

odd parity

one's complement