MODULE 3: TABLE OF CONTENTS

Chapter 10

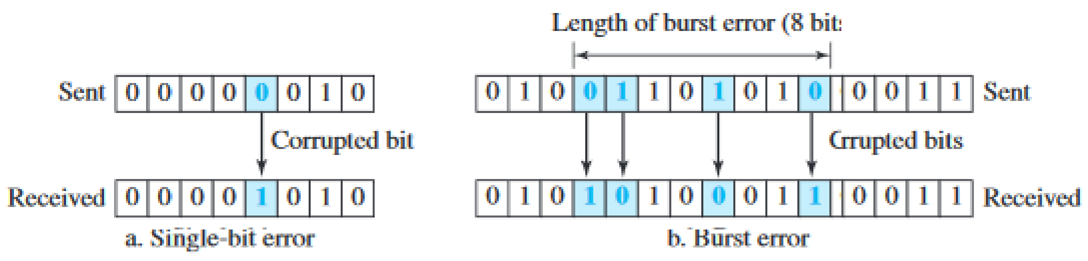
Error Detection and Correction

INTRODUCTION

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 11100 s burst of impulse noise on a transmission with a data rate of 1200 bps might change all or some of the12 bits of information.

Figure 10.1 *Single-bit and burst error*



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

Redundancy

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

Detection Versus Correction

The correction of errors is more difficult than the detection. In error detection, we are looking only to see if any error has occurred. The answer is a simple yes or no. We are not even interested in the number of errors. A single-bit error is the same for us as a burst error.

In error correction, we need to know the exact number of bits that are corrupted and more importantly, their location in the message. The number of the errors and the size of the message are important factors. If we need to correct one single error in an 8-bit data unit, we need to consider eight possible error locations; if we need to correct two errors in a data unit of the same size, we need to consider 28 possibilities. You can imagine the receiver's difficulty in finding 10 errors in a data unit of 1000 bits.

Forward Error Correction Versus Retransmission

There are two main methods of error correction. Forward error correction is the process in which the receiver tries to guess the message by using redundant bits. This is possible, as we see later, if the number of errors is small. Correction by retransmission is a technique in which the receiver detects the occurrence of an error and asks the sender to resend the message. Resending is repeated until a message arrives that the receiver believes is error-free.

Coding

Redundancy is achieved through various coding schemes. The sender adds redundant bits through a process that creates a relationship between the redundant bits and the actual data bits. The receiver checks the relationships between the two sets of bits to detect or correct the errors. The ratio of redundant bits to the data bits and the robustness of the process are important factors in any coding scheme. Figure 10.3 shows the general idea of coding.

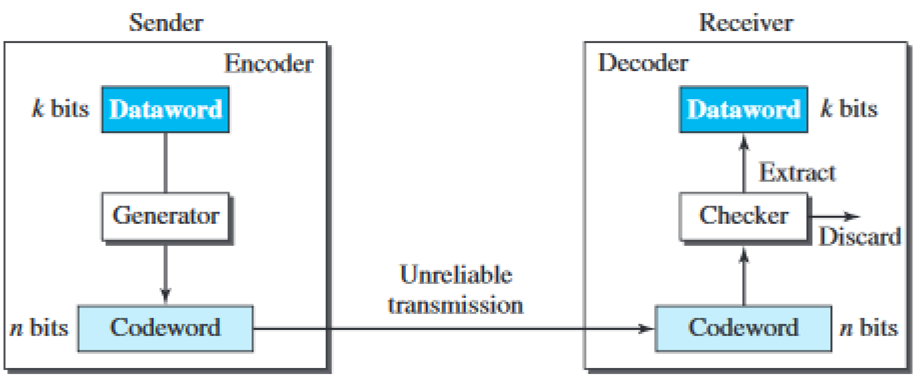
We can divide coding schemes into two broad.

Categories: block coding and convolution coding.

BLOCK CODING

Figure 10.2 shows the role of block coding in error detection. The sender creates code word out of data words by using a generator that applies the rules and procedures of encoding. Each codeword sent to the receiver may change during transmission. If the received codeword is the same as one of the valid code words, the
word is accepted; the corresponding data word is extracted for use. If the received code-word is not valid, it is discarded. However, if the codeword is corrupted during trans-mission but the received word still matches a valid codeword, the error remains undetected.

Figure 10.2 *Process of error detection in block coding*



In block coding, we divide our message into blocks, each of k bits, called data words. We add r redundant bits to each block to make the length $n = k + r$. The resulting *n-bit* blocks are called code words. How the extra r bits is chosen or calculated is something we will discuss later. For the moment, it is important to know that we have a set of data words, each of size k, and a set of code words, each of size of *n*. With *k* bits, we can create a combination of *2k* data words; with *n* bits, we can create a combination of *2n* code words. Since *n > k*, the number of possible code words is larger than the number of possible data words.

The block coding process is one-to-one; the same data word is always encoded as the same codeword. This means that we have *2n - 2k* code words that are not used. We call these code words invalid or illegal.

Example

Let us assume that k=2 and n=3. Table 10.1 shows the list of data words and code words. Later, we will see how to derive a codeword from a data word.

Table 10.1 *A code for error detection in Example 10.1*

Dataword	Codeword	Dataword	Codeword
00	000	10	101
01	011	11	110

Assume the sender encodes the data word 01 as 011 and sends it to the receiver. Consider the following cases:

1. The receiver receives 011. It is a valid codeword. The receiver extracts the dataword 01 from it.
2. The codeword is corrupted during transmission, and 111 is received (the leftmost bit is corrupted). This is not a valid codeword and is discarded.
3. The codeword is corrupted during transmission, and 000 is received (the right two bits are corrupted). This is a valid codeword. The receiver incorrectly extracts the data word 00. Two corrupted bits have made the error undetectable.

Hamming Distance

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

Minimum Hamming Distance

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

Before we continue with our discussion, we need to mention that any coding scheme needs to have at least three parameters: the codeword size *n*, the data word size *k*, and the minimum Hamming distance *dmin*. A coding scheme C is written as *C(n, k)* with a separate expression for *dmin*- For example, we can call our first coding scheme *C(3, 2)* with dmin =2 and our second coding scheme *C(5,* 2) with dmin ::= 3.

Hamming Distance and Error

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

Minimum Distance for Error Detection

Now let us find the minimum Hamming distance in a code if we want to be able to detect up to s errors. If s errors occur during transmission, the Hamming distance between the sent codeword and received codeword is s. If our code is to detect up to s errors, the minimum distance between the valid codes must be s + 1, so that the received codeword does not match a valid codeword. In other words, if the minimum distance between all valid codewords is s + 1, the received codeword cannot be erroneously mistaken for another codeword. The distances are not enough (s + 1) for the receiver to accept it as valid. The error will be detected. We need to clarify a point here: Although a code with *dmin* =s + 1

Minimum Distance for Error Correction

Error correction is more complex than error detection; a decision is involved. When a received codeword is not a valid codeword, the receiver needs to decide which valid codeword was actually sent. The decision is based on the concept of territory, an exclusive area surrounding the codeword. Each valid codeword has its own territory. We use a geometric approach to define each territory. We assume that each valid codeword has a circular territory with a radius of *t* and that the valid codeword is at the center. For example, suppose a codeword *x* is corrupted by *t* bits or less. Then this corrupted codeword is located either inside or on the perimeter of this circle. If the receiver receives a codeword that belongs to this territory, it decides that the original codeword is the one at the center. Note that we assume that only up to *t* errors have occurred; otherwise, the decision is wrong. Figure 10.9 shows this geometric interpretation. Some texts use a sphere to show the distance between all valid block codes.

LINEAR BLOCK CODES

Almost all block codes used today belong to a subset called linear block codes. The use of nonlinear block codes for error detection and correction is not as widespread because their

structure makes theoretical analysis and implementation difficult. We therefore concentrate on linear block codes. The formal definition of linear block codes requires the knowledge of abstract algebra (particularly Galois fields), which is beyond the scope of this book. We therefore give an informal definition. For our purposes, a linear block code is a code in which the exclusive OR (addition modulo-2) of two valid codewords creates another valid codeword.

Minimum Distance for Linear Block Codes

It is simple to find the minimum Hamming distance for a linear block code. The minimum Hamming distance is the number of Is in the nonzero valid codeword with the smallest number of 1s.

Some Linear Block Codes

Let us now show some linear block codes. These codes are trivial because we can easily find the encoding and decoding algorithms and check their performances.

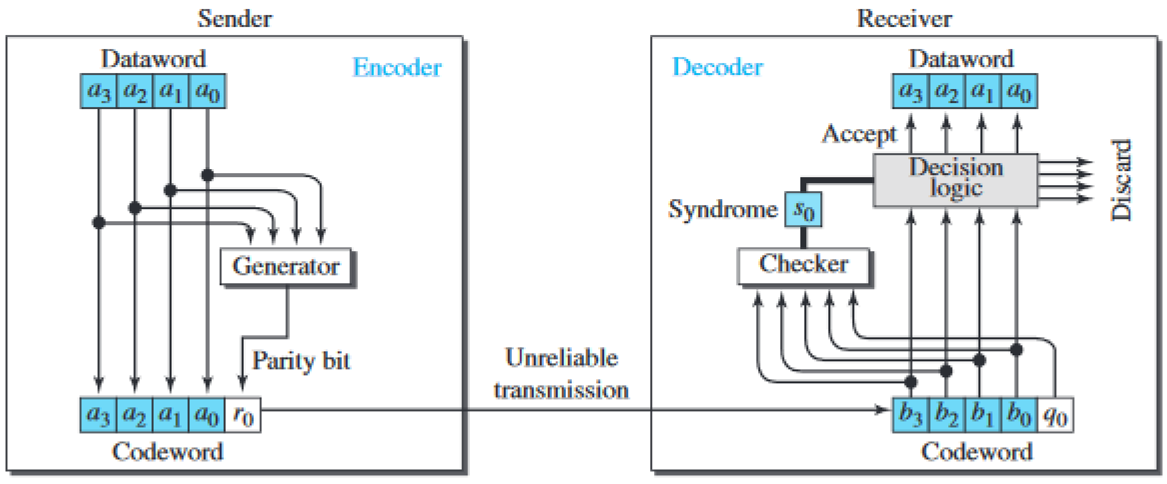
Simple Parity-Check Code

Perhaps the most familiar error-detecting code is the simple parity-check code. In this code, a *k-bit* data word is changed to an n-bit codeword where $n = k + 1$. The extra bit, called the parity bit, is selected to make the total number of Is in the codeword even. Although some implementations specify an odd number of Is, we discuss the even case. The minimum Hamming distance for this category is dmin =2, which means that the code is a single-bit error-detecting code; it cannot correct any error.

Table 10.2 *Simple parity-check code C(5, 4)*

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

Figure 10.4 *Encoder and decoder for simple parity-check code*



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

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

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

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

The syndrome is passed to the decision logic analyzer. If the syndrome is 0, there is no error in the received codeword; the data portion of the received codeword is accepted as the data word; if the syndrome is 1, the data portion of the received codeword is discarded. The data word is not created.

Example

Let us look at some transmission scenarios. Assume the sender sends the data word 1011. The codeword created from this dataword is 10111, which is sent to the receiver. We examine five cases:

1. No error occurs; the received codeword is 10111. The syndrome is 0. The data word 1011 is created.

2. One single-bit error changes a1, the received codeword is 10011. The syndrome is 1. No data word is created.

3. One single-bit error changes r0, the received codeword is 10110. The syndrome is 1. No data word is created. Note that although none of the data word bits are corrupted, no data word is created because the code is not sophisticated enough to show the position of the corrupted bit.

4. An error changes *ro* and a second error changes *a3*,the received codeword is 00110. The syndrome is 0. The data word 0011 is created at the receiver. Note that here the data word is wrongly created due to the syndrome value. The simple parity-check decoder cannot detect an even number of errors. The errors cancel each other out and give the syndrome a value of 0.

5. Three bits-a3, a*2*, and a1- are changed by errors. The received codeword is 01011. The syndrome is 1. The data word is not created. This shows that the simple parity check, guaranteed to detect one single error, can also find any odd number of errors.

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

CYCLIC CODES

Cyclic codes are special linear block codes with one extra property. In a cyclic code, if a codeword is cyclically shifted (rotated), the result is another codeword. For example, if 1011000 is a codeword and we cyclically left-shift, then 0110001 is also a codeword.

In this case, if we call the bits in the first word *ao* to *a6'* and the bits in the second word *B0* to *b6,* we can shift the bits by using the following:

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

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

Cyclic Redundancy Check

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

Table shows an example of a CRC code.

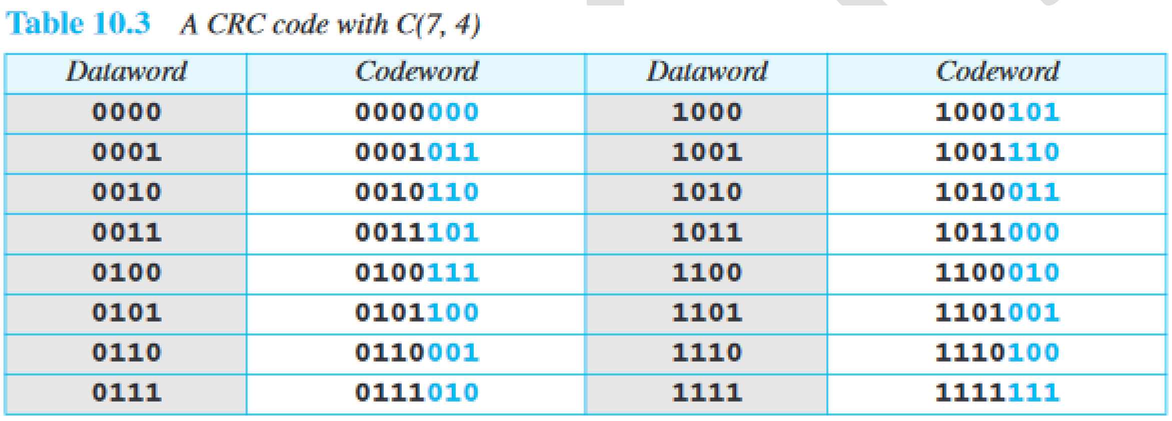
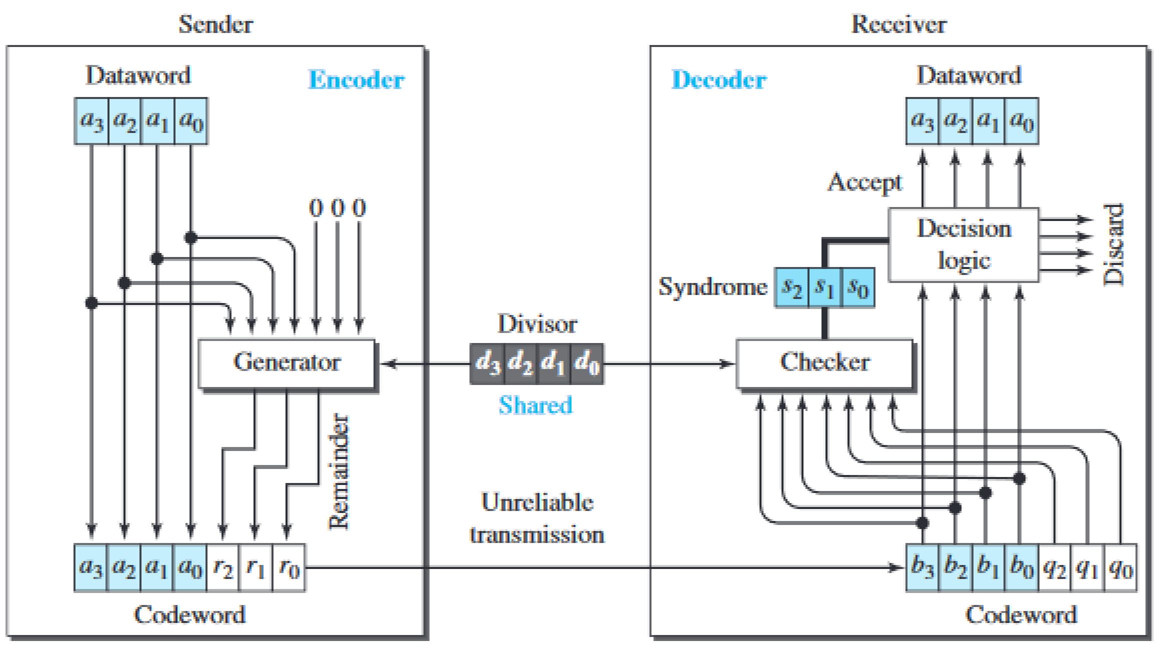


Table 10.3 *A CRC code with C(7, 4)*

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

Figure 10.5 shows one possible design for the encoder and decoder.

Figure 10.5 *CRC encoder and decoder*

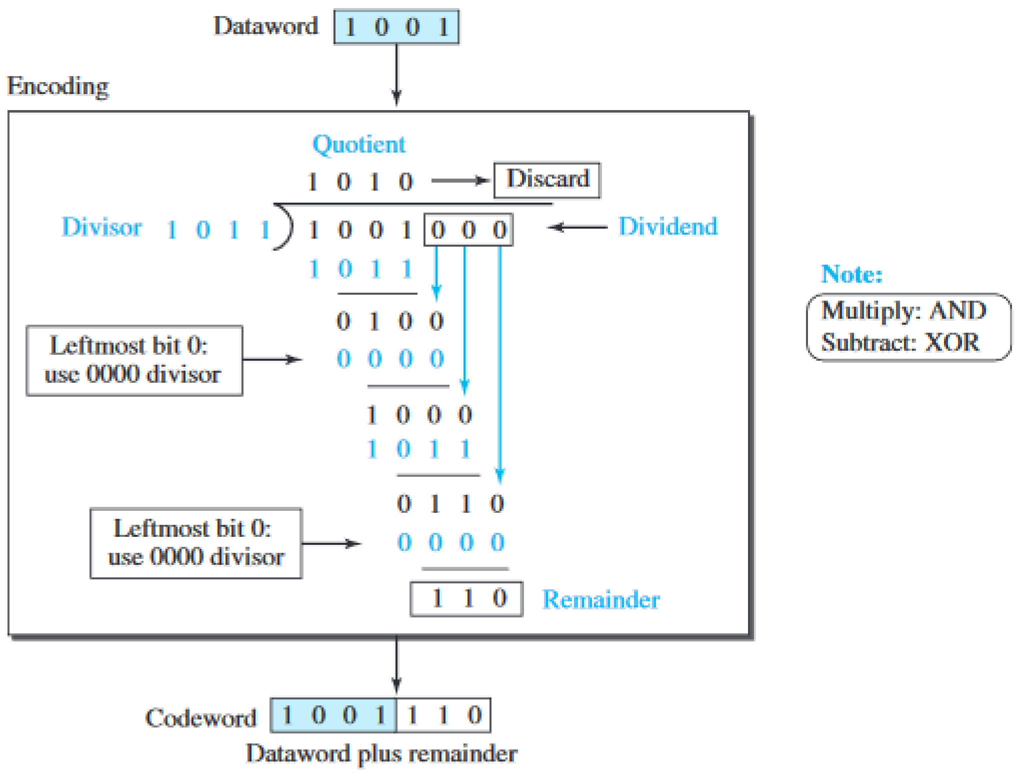


In the encoder, the data word has *k* bits (4 here); the codeword has *n* bits. The size of the data word is augmented by adding *n - k* (3 here) Os to the right-hand side of the word. The n-bit result is fed into the generator. The generator uses a divisor of size *n - k* + I (4 here), predefined and agreed upon. The generator divides the augmented data word by the divisor (modulo-2 division). The quotient of the division is discarded; the remainder *(r2 r1 r0)* is appended to the data word to create the codeword. The decoder receives the possibly corrupted codeword. A copy of all *n* bits is fed to the checker which is a replica of the generator. The remainder produced by the checker is a syndrome of *n - k* (3 here) bits, which is fed to the decision logic analyzer. The analyzer has a simple function. If the syndrome bits are all as, the 4 leftmost bits of the codeword are accepted as the data word (interpreted as no error); otherwise, the 4 bits are discarded (error).

Encoder

Let us take a closer look at the encoder. The encoder takes the data word and augments it with *n* - *k* number of as. It then divides the augmented data word by the divisor, as shown in Figure.

Figure 10.6 *Division in CRC encoder*

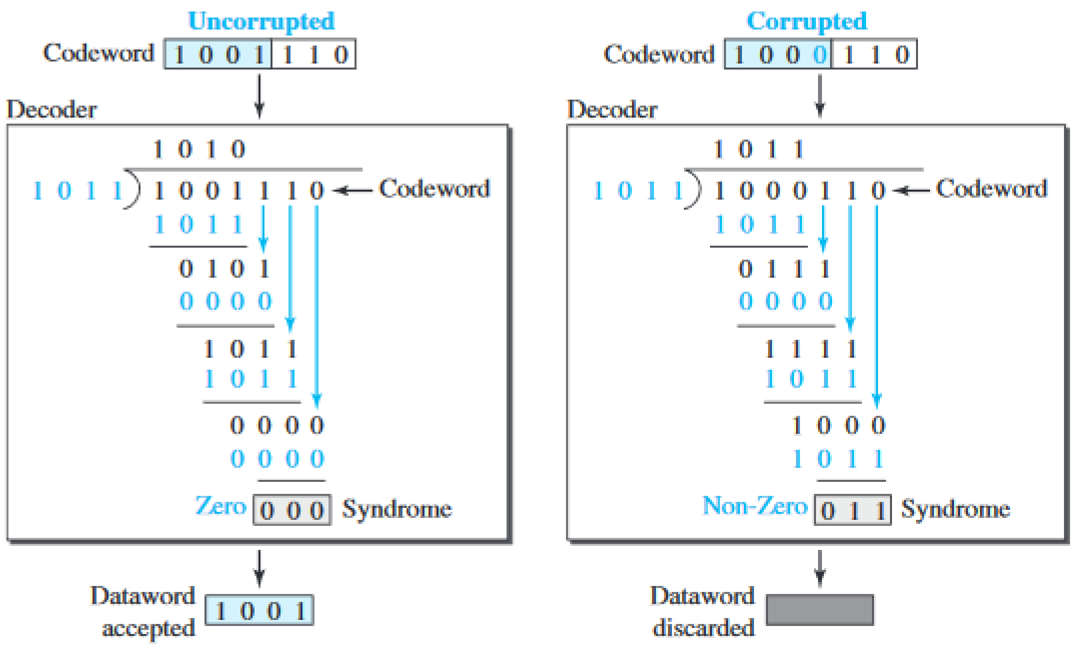


The process of modulo-2 binary division is the same as the familiar division process we use for decimal numbers. However, as mentioned at the beginning of the chapter, in this case addition and subtraction are the same. We use the XOR operation to do both. As in decimal division, the process is done step by step. In each step, a copy of the divisor is XORed with the 4 bits of the dividend. The result of the XOR operation (remainder) is 3 bits (in this case), which is used for the next step after 1 extra bit is pulled down to make it 4 bits long. There is one important point we need to remember in this type of division. If the leftmost bit of the dividend (or the part used in each step) is 0, the step cannot use the regular divisor; we need to use an all-0s divisor. When there are no bits left to pull down, we have a result. The 3-bit remainder forms the check bits *(r2 r1* and *r0).* They are appended to the data word to create the codeword.

Decoder

The codeword can change during transmission. The decoder does the same division process as the encoder. The remainder of the division is the syndrome. If the syndrome is all 0s, there is no error; the data word is separated from the received codeword and accepted. Otherwise, everything is discarded. Figure 10.16 shows two cases: The left hand figure shows the value of syndrome when no error has occurred; the syndrome is 000. The right-hand part of the figure shows the case in which there is one single error. The syndrome is not all 0s (it is 01l).

Figure 10.7 *Division in the CRC decoder for two cases*



Polynomials

A better way to understand cyclic codes and how they can be analyzed is to represent them as polynomials. Again, this section is optional. A pattern of Os and 1s can be represented as a **polynomial** with coefficients of 0 and 1. The power of each term shows the position of the bit; the coefficient shows the value of the bit. Figure shows a binary pattern and its polynomial representation. In Figure 10.21a we show how to translate a binary pattern to a polynomial; in Figure we show how the polynomial can be shortened by removing all terms with zero coefficients and replacing x^1 by x and x^0 by 1.

Figure 10.8 *A polynomial to represent a binary word*

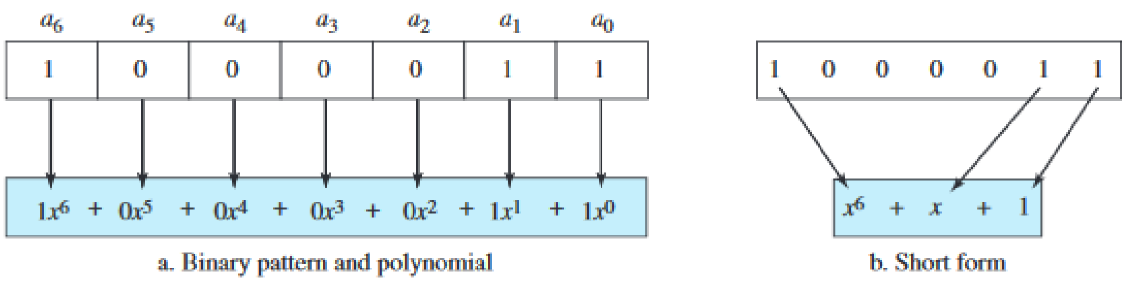

a_6 a_5 a_4 a_3 a_2 a_1 a_0
1 0 0 0 0 1 1
$1x^6 + 0x^5 + 0x^4 + 0x^3 + 0x^2 + 1x^1 + 1x^0$
a. Binary pattern and polynomial
1 0 0 0 0 1 1
$x^6 + x + 1$
b. Short form

Figure shows one immediate benefit; a 7-bit pattern can be replaced by three terms. The benefit is even more conspicuous when we have a polynomial such as $x23 + X3 + 1$. Here the bit pattern is 24 bits in length (three Is and twenty-one Os) while the polynomial is just three terms.

Degree of a Polynomial

The degree of a polynomial is the highest power in the polynomial. For example, the degree of the polynomial $x6 + x + 1$ is 6. Note that the degree of a polynomial is 1 less that the number of bits in the pattern. The bit pattern in this case has 7 bits.

Adding and Subtracting Polynomials

Adding and subtracting polynomials in mathematics are done by adding or subtracting the coefficients of terms with the same power. In our case, the coefficients are only 0 and 1, and adding is in modulo-2. This has two consequences. First, addition and subtraction are the same. Second, adding or subtracting is done by combining terms and deleting pairs of identical terms. For example, adding $x^5 + x^4 + x^2$ and $x^6 + x^4 + x^2$ gives just $x^6 + x^5$. The terms x^4 and x^2 are deleted. However, note that if we add, for example, three polynomials and we get x^2 three times, we delete a pair of them and keep the third.

Multiplying or Dividing Terms

In this arithmetic, multiplying a term by another term is very simple; we just add the powers. For example, $x^3 * x^4$ is x^7, For dividing, we just subtract the power of the second term from the power of the first. For example, x^5 / x^2 is x^3.

Multiplying Two Polynomials

Multiplying a polynomial by another is done term by term. Each term of the first polynomial must be multiplied by all terms of the second. The result, of course, is then simplified, and pairs of equal terms are deleted. The following is an example:

$$\begin{aligned}&(x^5 + x^3 + x^2 + x)(x^2 + x + 1)\\ &\quad = x^7 + x^6 + x^5 + x^5 + x^4 + x^3 + x^4 + x^3 + x^2 + x^3 + x^2 + x\\ &\quad = x^7 + x^6 + x^3 + x\end{aligned}$$

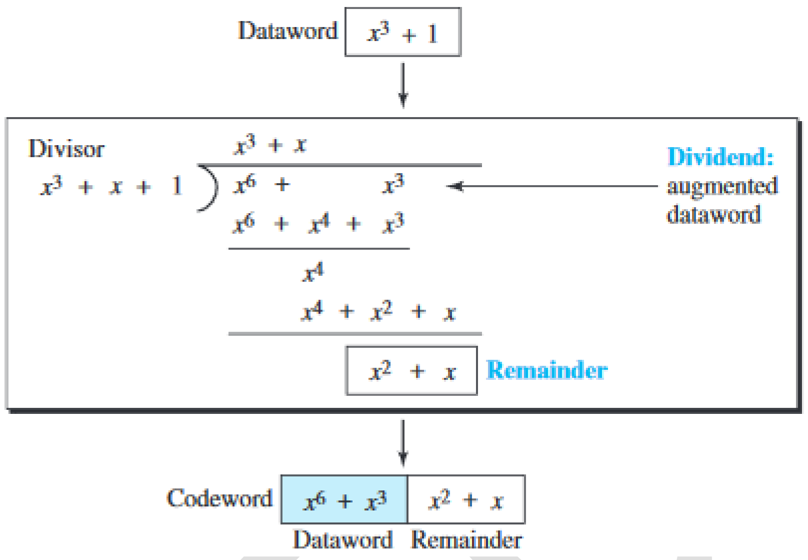
Dividing One Polynomial by Another

Division of polynomials is conceptually the same as the binary division we discussed for an encoder. We divide the first term of the dividend by the first term of the divisor to get the first term of the quotient. We multiply the term in the quotient by the divisor and subtract the result from the dividend. We repeat the process until the dividend degree is less than the divisor degree. We will show an example of division later in this chapter.

Cyclic Code Encoder Using Polynomials

Now that we have discussed operations on polynomials, we show the creation of a codeword from a data word. Figure 10.22 is the polynomial version of Figure 10.15. We can see that the process is shorter. The data word 1001 is represented as $x3 + 1$. The divisor 1011 is represented as $x^3 + x + 1$. To find the augmented data word, we have left-shifted the data word 3 bits (multiplying by x\ The result is $x^6 + x^3$. Division is straightforward. We divide the first term of the dividend, $x6$, by the first term of the divisor, x^3. The first term of the quotient is then x^6/x^3, or x^3. Then we multiply x^3 by the divisor and subtract (according to our previous definition of subtraction) the result from the dividend. The result is x^4, with a degree greater than the divisor's degree; we continue to divide until the degree of the remainder is less than the degree of the divisor.

Figure 10.9 *CRC division using polynomials*



It can be seen that the polynomial representation can easily simplify the operation of division in this case, because the two steps involving all-Os divisors are not needed here. (Of course, one could argue that the all-Os divisor step can also be eliminated in binary division.) In a polynomial representation, the divisor is normally referred to as the generator polynomial *t(x)*.

Cyclic Code Analysis

We can analyze a cyclic code to find its capabilities by using polynomials. We define the following, *where f(x)* is a polynomial with binary coefficients.

Dataword:
d(x)
Syndrome:
s(x)
Codeword:
c(x)

Error: *e(x)*

Generator: *g(x)*

If *s(x)* is not zero, then one or more bits is corrupted. However, if *s(x)* is zero, either no bit is corrupted or the decoder failed to detect any errors.

In a cyclic code,

1. If $s(x) \neq 0$, one or more bits is corrupted.
2. If $s(x) = 0$, either
   a. No bit is corrupted, or
   b. Some bits are corrupted, but the decoder failed to detect them.

In our analysis we want to find the criteria that must be imposed on the generator, *g(x)* to detect the type of error we especially want to be detected. Let us first find the relationship among the sent codeword, error, received codeword, and the generator. We can say

Received codeword =c(x) +e(x)

In other words, the received codeword is the sum of the sent codeword and the error. The receiver divides the received codeword by *g(x)* to get the syndrome. We can write this as

$$\frac{\text{Received codeword}}{g(x)} = \frac{c(x)}{g(x)} + \frac{e(x)}{g(x)}$$

The first term at the right-hand side of the equality does not have a remainder (according to the definition of codeword). So the syndrome is actually the remainder of the second term on the right-hand side. If this term does not have a remainder (syndrome =0), either *e(x)* is 0 or *e(x)* is divisible by *g(x).* We do not have to worry about the first case (there is no error); the second case is very important. Those errors that are divisible by *g(x)* are not caught. Let us show some specific errors and see how they can be caught by a well designed *g(x).*

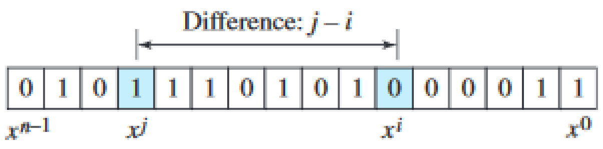
Single-Bit Error

What should be the structure of *g(x)* to guarantee the detection of a single-bit error? A single-bit error is *e(x)* =*xi,* where *i* is the position of the bit. If a single-bit error is caught, then *xi* is not divisible by *g(x).* (Note that when we say *not divisible*, we mean that there is a remainder.) If *g(x)* ha~ at least two terms (which is normally the case) and the coefficient of *x0* is not zero (the rightmost bit is 1), then *e(x)* cannot be divided by *g(x).*

Two Isolated Single-Bit Errors

Now imagine there are two single-bit isolated errors. Under what conditions can this type of error be caught? We can show this type of error as *e(x)* =: *xl* + *xi.* The values of *I* and *j* define the positions of the errors, and the difference *j* - *i* defines the distance between the two errors, as shown in Figure.

Figure 10.10 *Representation of two isolated single-bit errors using polynomials*



Odd Numbers of Errors

A generator with a factor of x + 1 can catch all odd numbers of errors. This means that we need to make x + 1 a factor of any generator. Note that we are not saying that the generator itself should be x + 1; we are saying that it should have a factor of x + 1. If it is only x + 1, it cannot catch the two adjacent isolated errors (see the previous section). For example, $x4 + x2$ + X + 1 can catch all odd-numbered errors since it can be written as a product of the two polynomials x + 1 and $x3 + x2$ + 1.

Burst Errors

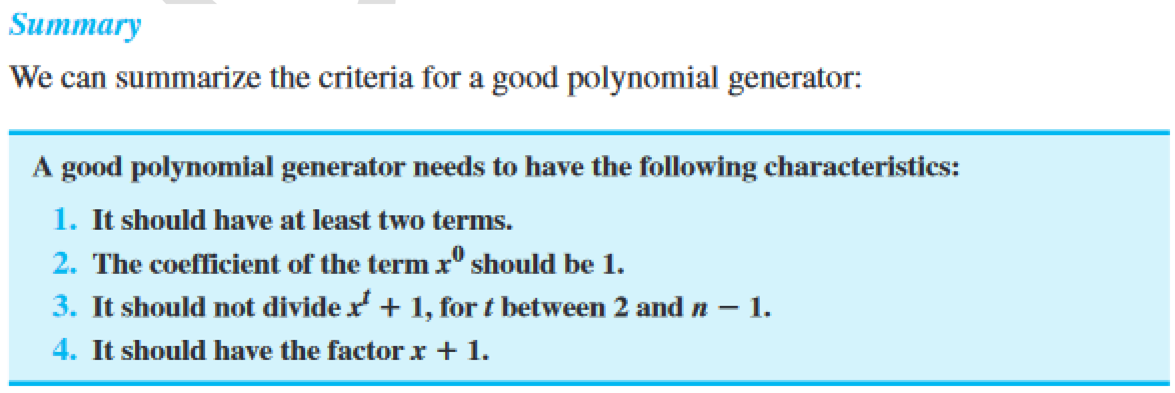
Now let us extend our analysis to the burst error, which is the most important of all. A burst error is of the form $e(x) =eJ + ... + xi)$. Note the difference between a burst error and two isolated single-bit errors. The first can have two terms or more; the second can only have two terms. We can factor out xi and write the error as $xi(xJ-i + ... +$ 1). If our generator can detect a single error (minimum condition for a generator), then it cannot divide xi. What we should worry about are those generators that divide $xJ-i + ... +$ 1. In other words, the remainder of $(xJ-i + ... + 1)/(xr +... +$ 1) must not be zero. Note that the denominator is the generator polynomial.

We can have three cases:

1. If $j - i < r$, the remainder can never be zero. We can write $j - i =L$ - 1, where L is the length of the error. So L - 1 < r or L < r + 1 or L :::;: r. This means all burst errors with length smaller than or equal to the number of check bits r will be detected.

2. In some rare cases, if $j - i = r$, or $L = r$ + 1, the syndrome is 0 and the error is undetected. It can be proved that in these cases, the probability of undetected burst error of length r + 1 is (Il2r-l. For example, if our generator is x $l4$ + ~ + 1, in which r =14, aburst error of length L =15 can slip by undetected with the probability of (1/2)14-1 or almost 1 in 10,000.

3. In some rare cases, if $j - i > r$, or $L > r$ + 1, the syndrome is 0 and the error is undetected. It can be proved that in these cases, the probability of undetected burst error of length greater than r + 1 is (112t For example, if our generator is $x14 + x3$ + 1, in which r =14, a burst error of length greater than 15 can slip by undetected with the probability of (112)14 or almost 1 in 16,000 cases.



Summary

We can summarize the criteria for a good polynomial generator:

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 $x + 1$.**

Standard Polynomials

Some standard polynomials used by popular protocols for Regeneration are shown in Table

Table 10.4 *Standard polynomials*

Name	*Polynomial*	*Used in*
CRC-8	$x^8 + x^2 + x + 1$ 100000111	ATM header
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x^2 + 1$ 11000110101	ATM AAL
CRC-16	$x^{16} + x^{12} + x^5 + 1$ 10001000000100001	HDLC
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ 100000100110000010001110110110111	LANs

Example 10.8

Which of the following $g(x)$ values guarantees that a single-bit error is caught? For each case, what is the error that cannot be caught?

a. $x + 1$
b. x^3
c. 1

Solution

a. No x^i can be divisible by $x + 1$. In other words, $x^i/(x + 1)$ always has a remainder. So the syndrome is nonzero. Any single-bit error can be caught.

b. If i is equal to or greater than 3, x^i is divisible by $g(x)$. The remainder of x^i/x^3 is zero, and the receiver is fooled into believing that there is no error, although there might be one. Note that in this case, the corrupted bit must be in position 4 or above. All single-bit errors in positions 1 to 3 are caught.

c. All values of i make x^i divisible by $g(x)$. No single-bit error can be caught. In addition, this $g(x)$ is useless because it means the codeword is just the dataword augmented with $n - k$ zeros.

Example 10.9

Find the status of the following generators related to two isolated, single-bit errors.

a. $x + 1$
b. $x^4 + 1$
c. $x^7 + x^6 + 1$
d. $x^{15} + x^{14} + 1$

Solution

a. This is a very poor choice for a generator. Any two errors next to each other cannot be detected.
b. This generator cannot detect two errors that are four positions apart. The two errors can be anywhere, but if their distance is 4, they remain undetected.
c. This is a good choice for this purpose.
d. This polynomial cannot divide any error of type $x^t + 1$ if t is less than 32,768. This means that a codeword with two isolated errors that are next to each other or up to 32,768 bits apart can be detected by this generator.

Example 10.10

Find the suitability of the following generators in relation to burst errors of different lengths.

a. $x^6 + 1$
b. $x^{18} + x^7 + x + 1$
c. $x^{32} + x^{23} + x^7 + 1$

Solution

a. This generator can detect all burst errors with a length less than or equal to 6 bits; 3 out of 100 burst errors with length 7 will slip by; 16 out of 1000 burst errors of length 8 or more will slip by.
b. This generator can detect all burst errors with a length less than or equal to 18 bits; 8 out of 1 million burst errors with length 19 will slip by; 4 out of 1 million burst errors of length 20 or more will slip by.
c. This generator can detect all burst errors with a length less than or equal to 32 bits; 5 out of 10 billion burst errors with length 33 will slip by; 3 out of 10 billion burst errors of length 34 or more will slip by.

Advantages of Cyclic Codes

We have seen that cyclic codes have a very good performance in detecting single-bit errors, double errors, an odd number of errors, and burst errors. They can easily be implemented in hardware and software. They are especially fast when implemented in hardware. This has made cyclic codes a good candidate for many networks.

Other Cyclic Codes

The cyclic codes we have discussed in this section are very simple. The check bits and syndromes can be calculated by simple algebra. There are, however, more powerful polynomials that are based on abstract algebra involving Galois fields. These are beyond the scope of this book. One of the most interesting of these codes is the Reed-Solomon code used today for both detection and correction.

CHECKSUM

The last error detection method we discuss here is called the checksum. The checksum is used in the Internet by several protocols although not at the data link layer. However,we briefly discuss it here to complete our discussion on error checking. Like linear and cyclic codes, the checksum is based on the concept of redundancy. Several protocols still use the checksum for error detection although the tendency is to replace it with a CRC. This means that the CRC is also used in layers other than the data link layer.

Idea

The concept of the checksum is not difficult. Let us illustrate it with a few examples. One's Complement The previous example has one major drawback. All of our data can be written as a 4-bit word (they are less than 15) except for the checksum. One solution is to use one's complement arithmetic. In this arithmetic, we can represent unsigned numbers between 0 and *2n* - 1 using only *n* bits. t If the number has more than *n* bits, the extra leftmost bits need to be added to the *n* rightmost bits (wrapping). In one's complement arithmetic, a negative number can be represented by inverting all bits (changing a 0 to a 1 and a 1 to a 0). This is the same as subtracting the number from *2n* - 1.

Internet Checksum

Traditionally, the Internet has been using a 16-bit checksum. The sender calculates the checksum by following these steps.

Sender site:

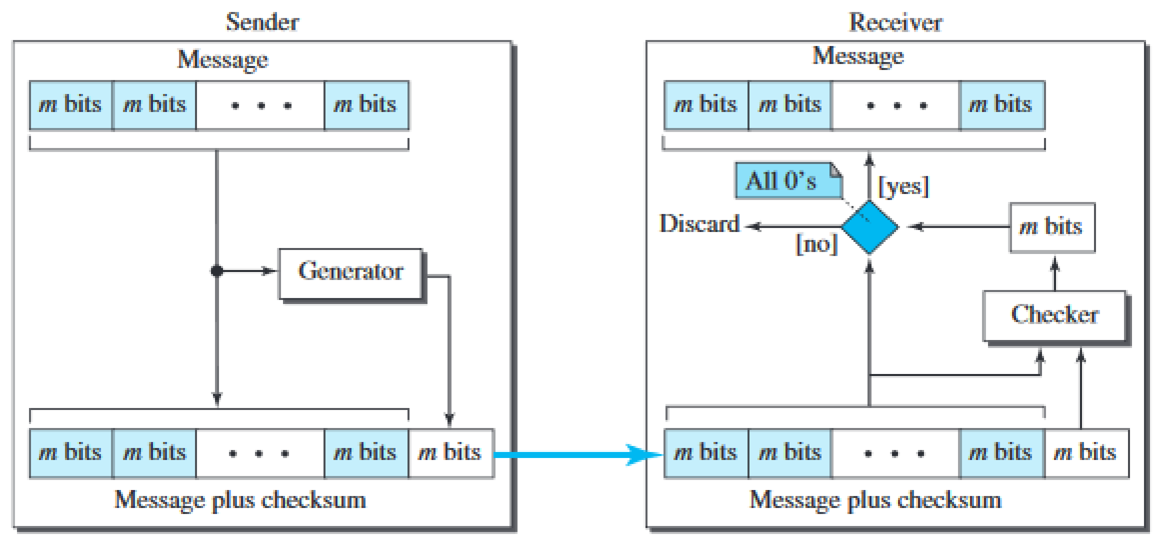
1. The message is divided into 16-bit words.
2. The value of the checksum word is set to O.
3. All words including the checksum are added ushtg one's complement addition.
4. The sum is complemented and becomes the checksum.
5. The checksum is sent with the data.

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.

At the source, the message is first divided into m-bit units. The generator then creates an extra m-bit unit called the checksum, which is sent with the message. At the destination, the checker creates a new checksum from the combination of the message and sent checksum. If the new checksum is all 0s, the message is accepted; otherwise, the message is discarded (Figure 10.15). Note that in the real implementation, the checksum unit is not necessarily added at the end of the message; it can be inserted in
the middle of the message.

Figure 10.15 *Checksum*



Concept

The idea of the traditional checksum is simple. We show this using a simple example. Example 10.11 Suppose the message is a list of five 4-bit numbers that we want to send to a destination. In addition to sending these numbers , we send the sum of the numbers. For example, if the set of numbers is (7, 11, 12, 0, 6), we send (7, 11, 12, 0, 6, 36), where 36 is the sum of the original numbers. The receiver adds the five numbers and compares the result with the sum. If the two are the same, the receiver assumes no error, accepts the five numbers, and discards the sum. Otherwise, there is an error somewhere and the message is not accepted. One's Complement Addition The previous example has one major drawback. Each number can be written as a 4-bit word (each is less than 15) except for the sum. One solution is to use one's complement arithmetic. In this arithmetic, we can represent unsigned numbers between 0 and 2m−1 using only m bits. If the number has more than m bits, the extra leftmost bits need to be added to the m rightmost bits (wrapping).

In the previous example, the decimal number 36 in binary is (100100)2. To change it to a 4-bit number we add the extra leftmost bit to the right four bits as shown below. Instead of sending 36 as the sum, we can send 6 as the sum (7, 11, 12, 0, 6, 6). The receiver can add the first five numbers in one's complement arithmetic. If the result is 6, the numbers are accepted; otherwise, they are rejected.

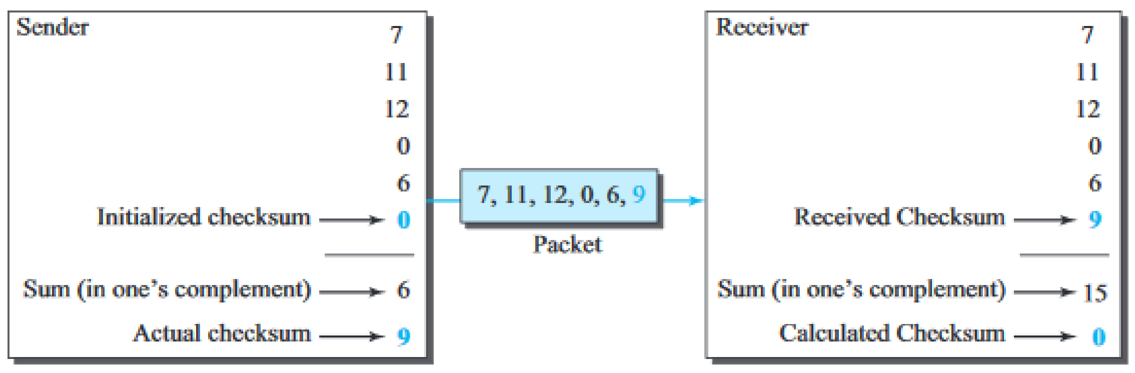
Checksum

We can make the job of the receiver easier if we send the complement of the sum, the checksum. In one's complement arithmetic, the complement of a number is found by completing all bits (changing all 1s to 0s and all 0s to 1s). This is the same as subtracting the number from 2m−1. In one's complement arithmetic, we have two 0s: one positive and one negative, which are complements of each other. The positive zero has all m bits set to 0; the negative zero has all bits set to 1 (it is 2m− 1). If we add a number with its complement, we get a negative zero (a number with all bits set to 1). When the receiver adds all five numbers (including the checksum), it gets a negative zero. The receiver can complement the result again to get a positive zero.

Example

Let us use the idea of the checksum in Example. The sender adds all five numbers in one's complement to get the sum = 6. The sender then complements the result to get the checksum =9, which is 15 −6. Note that 6 =(0110)2 and 9 =(1001)2; they are complements of each other. The sender sends the five data numbers and the checksum (7, 11, 12, 0, 6, 9). If there is no corruption in transmission, the receiver receives (7, 11, 12, 0, 6, 9) and adds them in one's complement to get 15.The sender complements 15 to get 0. This shows that data have not been corrupted. Figure shows the process.

Figure 10.16 *Example 10.13*



Internet Checksum

Traditionally, the Internet has used a 16-bit checksum. The sender and the receiver follow the steps depicted in Table 10.5. The sender or the receiver uses five steps.

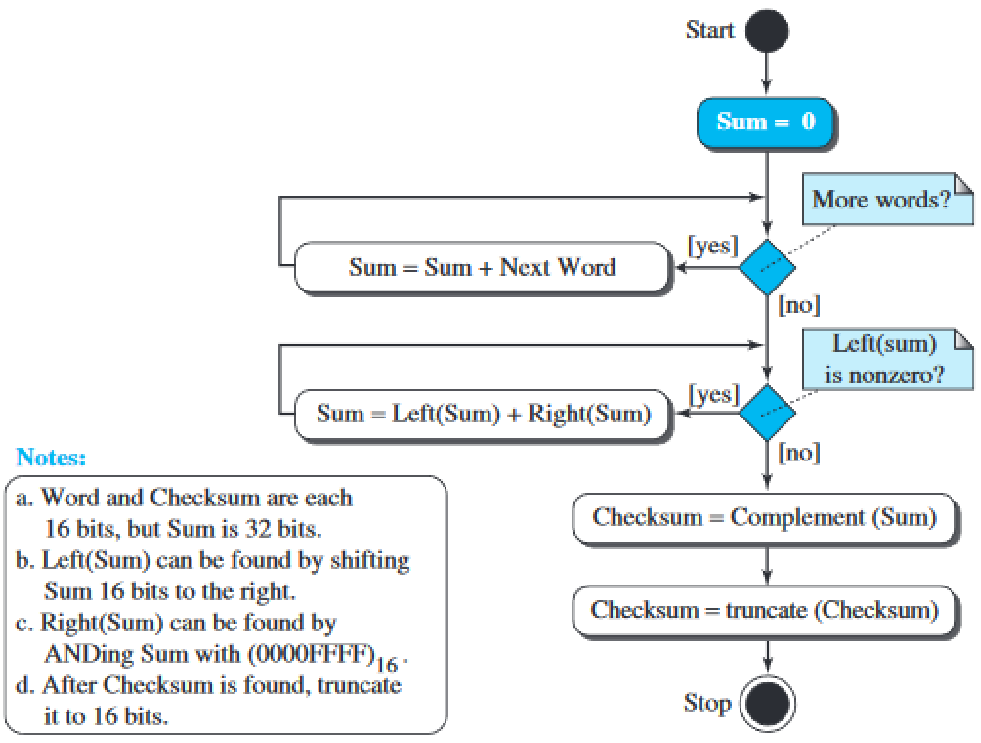
Table 10.5 *Procedure to calculate the traditional checksum*

Sender	*Receiver*
1. The message is divided into 16-bit words.	1. The message and the checksum are received.
2. The value of the checksum word is initially set to zero.	2. The message is divided into 16-bit words.
3. All words including the checksum are added using one's complement addition.	3. All words are added using one's complement addition.
4. The sum is complemented and becomes the checksum.	4. The sum is complemented and becomes the new checksum.
5. The checksum is sent with the data.	5. If the value of the checksum is 0, the message is accepted; otherwise, it is rejected.

Algorithm

We can use the flow diagram of Figure 10.17 to show the algorithm for calculation of the checksum. A program in any language can easily be written based on the algorithm. Note that the first loop just calculates the sum of the data units in two's complement; the second loop wraps the extra bits created from the two's complement calculation to simulate the calculations in one's complement. This is needed because almost all computers today do calculation in two's complement.

Figure 10.17 *Algorithm to calculate a traditional checksum*


Start
Sum = 0
More words?
[yes]
Sum = Sum + Next Word
[no]
Left(sum) is nonzero?
[yes]
Sum = Left(Sum) + Right(Sum)
[no]
Checksum = Complement (Sum)
Checksum = truncate (Checksum)
Stop
Notes:
a. Word and Checksum are each 16 bits, but Sum is 32 bits.
b. Left(Sum) can be found by shifting Sum 16 bits to the right.
c. Right(Sum) can be found by ANDing Sum with (0000FFFF)16 .
d. After Checksum is found, truncate it to 16 bits.

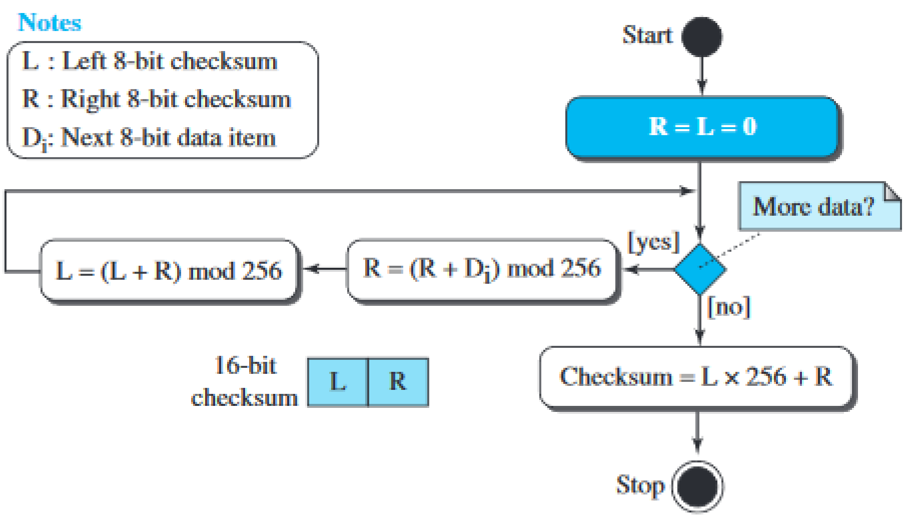
Other Approaches to the Checksum

As mentioned before, there is one major problem with the traditional checksum calculation. If two 16-bit items are transposed in transmission, the checksum cannot catch this error. The reason is that the traditional checksum is not weighted: it treats each data item equally. In other words, the order of data items is immaterial to the calculation. Several approaches have been used to prevent this problem. We mention two of them here: Fletcher and Adler.

Fletcher Checksum

The Fletcher checksum was devised to weight each data item according to its position. Fletcher has proposed two algorithms: 8-bit and 16-bit. The first, 8-bit Fletcher, calculates on 8-bit data items and creates a 16-bit checksum. The second, 16-bit Fletcher, calculates on 16-bit data items and creates a 32-bit checksum. The 8-bit Fletcher is calculated over data octets (bytes) and creates a 16-bit check-sum. The calculation is done modulo 256 (2^8), which means the intermediate results are divided by 256 and the remainder is kept. The algorithm uses two accumulators, L and R. The first simply adds data items together; the second adds a weight to the calculation. There are many variations of the 8-bit Fletcher algorithm; we show a simple one in Figure 10.18. The 16-bit Fletcher checksum is similar to the 8-bit Fletcher checksum, but it is calculated over 16-bit data items and creates a 32-bit checksum. The calculation is done modulo 65,536.

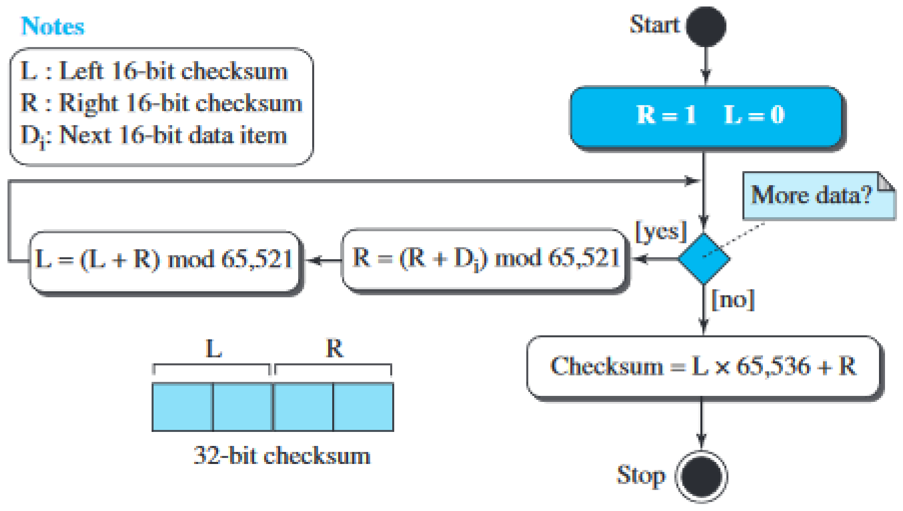
Figure 10.18 *Algorithm to calculate an 8-bit Fletcher checksum*



Adler Checksum

The Adler checksum is a 32-bit checksum. Figure 10.19 shows a simple algorithm in flowchart form. It is similar to the 16-bit Fletcher with three differences. First, calculation is done on single bytes instead of 2 bytes at a time. Second, the modulus is a prime number (65,521) instead of 65,536. Third, L is initialized to 1 instead of 0. It has been proved that a prime modulo has a better detecting capability in some combinations of data.

Figure 10.19 *Algorithm to calculate an Adler checksum*



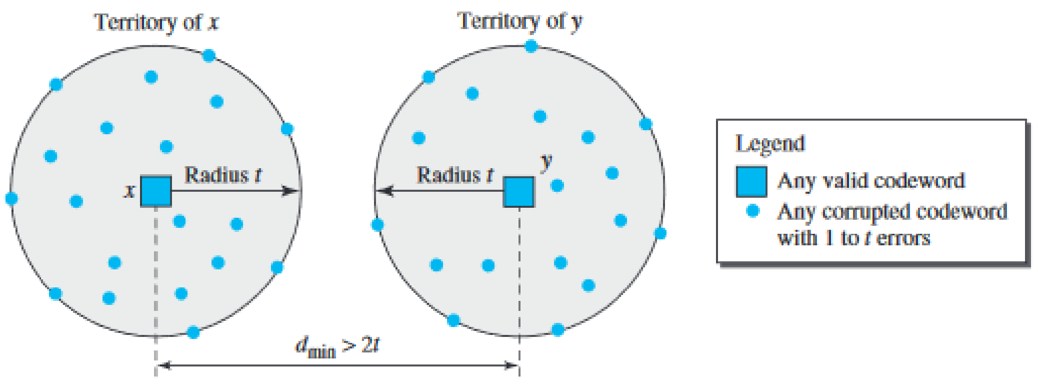
FORWARD ERROR CORRECTION

We discussed error detection and retransmission in the previous sections. However, retransmission of corrupted and lost packets is not useful for real-time multimedia transmission because it creates an unacceptable delay in reproducing: we need to wait until the lost or corrupted packet is resent. We need to correct the error or reproduce the packet immediately. Several schemes have been designed and used in this case that are collectively referred to as forward error correction (FEC) techniques. We briefly discuss some of the common techniques here.

Using Hamming Distance

We earlier discussed the Hamming distance for error detection. We said that to detect s errors, the minimum Hamming distance should be $d_{min}=s+1$. For error detection, we definitely need more distance. It can be shown that to detect t errors, we need to have $d_{min}=2t+1$. In other words, if we want to correct 10 bits in a packet, we need to make the minimum hamming distance 21 bits, which means a lot of redundant bits need to be sent with the data. To give an example, consider the famous BCH code. In this code, if data is 99 bits, we need to send 255 bits (extra 156 bits) to correct just 23 possible bit errors. Most of the time we cannot afford such a redundancy. We give some examples of how to calculate the required bits in the practice set. Figure 10.20 shows the geometrical representation of this concept.

Figure 10.20 *Hamming distance for error correction*



Using XOR

Another recommendation is to use the property of the exclusive OR operation as shown below.

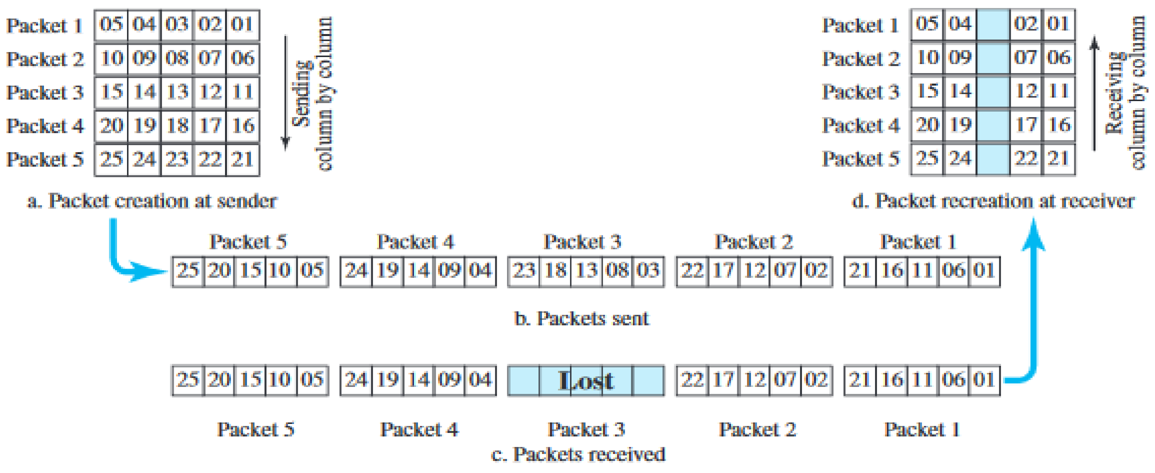
$$R = P_1 \oplus P_2 \oplus \ldots \oplus P_i \oplus \ldots \oplus P_N \rightarrow P_i = P_1 \oplus P_2 \oplus \ldots \oplus R \oplus \ldots \oplus P_N$$

In other words, if we apply the exclusive OR operation on N data items (P1to PN), we can recreate any of the data items by exclusive-ORing all of the items, replacing the one to be created by the result of the previous operation (R). This means that we can divide a packet into N chunks, create the exclusive OR of all the chunks and send N+1 chunks. If any chunk is lost or corrupted, it can be created at the receiver site. Now the question is what should the value of N be. If N= 4, it means that we need to send 25 percent extra data and be able to correct the data if only one out of four chunks is lost.

Chunk Interleaving

Another way to achieve FEC in multimedia is to allow some small chunks to be missing at the receiver. We cannot afford to let all the chunks belonging to the same packet be missing; however, we can afford to let one chunk be missing in each packet. Figure 10.21 shows that we can divide each packet into 5 chunks (normally the number is much larger). We can then create data chunk by chunk (horizontally), but combine the chunks into packets vertically. In this case, each packet sent carries a chunk from several original packets. If the packet is lost, we miss only one chunk in each packet, which is normally acceptable in multimedia communication.

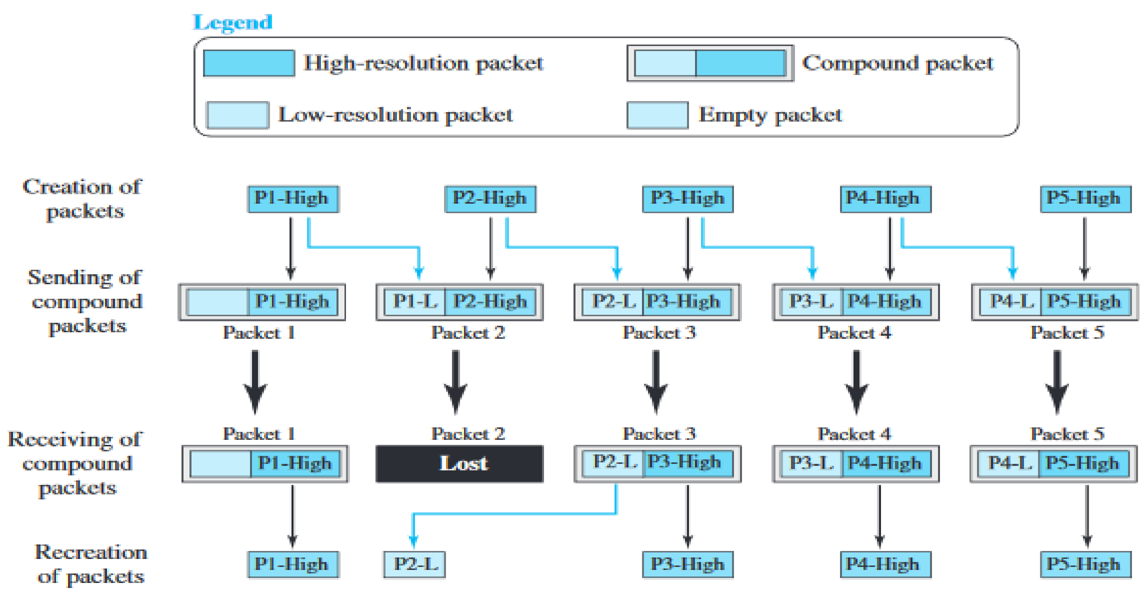
Figure 10.21 *Interleaving*



Compounding High- and Low-Resolution Packets

Still another solution is to create a duplicate of each packet with a low-resolution redundancy and combine the redundant version with the next packet. For example, we can create four low-resolution packets out of five high-resolution packets and send them as shown in Figure 10.22. If a packet is lost, we can use the low-resolution version from the next packet. Note that the low-resolution section in the first packet is empty. In this method, if the last packet is lost, it cannot be recovered, but we use the low-resolution version of a packet if the lost packet is not the last one. The audio and video reproduction does not have the same quality, but the lack of quality is not recognized most of the time.

Figure 10.22 *Compounding high- and low-resolution packets*



CHAPTER 11

DATA LINK CONTROL

DLC Services

- DLC deals with procedures for communication between two adjacent nodes.(dedicated or broadcast).
- DLC functions include framing and flow and error control.

Frames

- The data-link layer needs to pack bits into frames, so that each frame is distinguishable from another.

- Framing in the data-link layer separates a message from one source to a destination by adding a sender address and a destination address. The destination address defines where the packet is to go; the sender address helps the recipient acknowledge the receipt.

Frame Size

- Frames can be of fixed or variable size.
- In fixed-size framing, there is no need for defining the boundaries of the frames; the size itself can be used as a delimiter.
- An example of this type of framing is the ATM WAN, which uses frames of fixed size called cells.

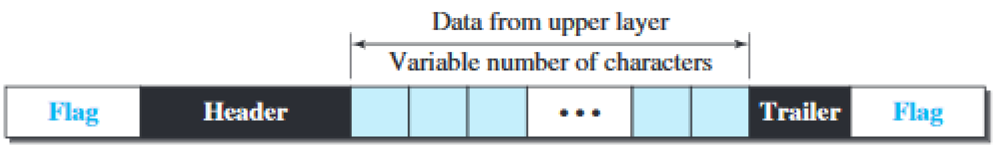
.

- Variable size framing, prevalent in local-area networks. In variable-size framing, we need a way to define the end of one frame and the beginning of the next.
- Historically, two approaches were used for this purpose: a character-oriented approach and a bit oriented approach.

Character-Oriented Framing

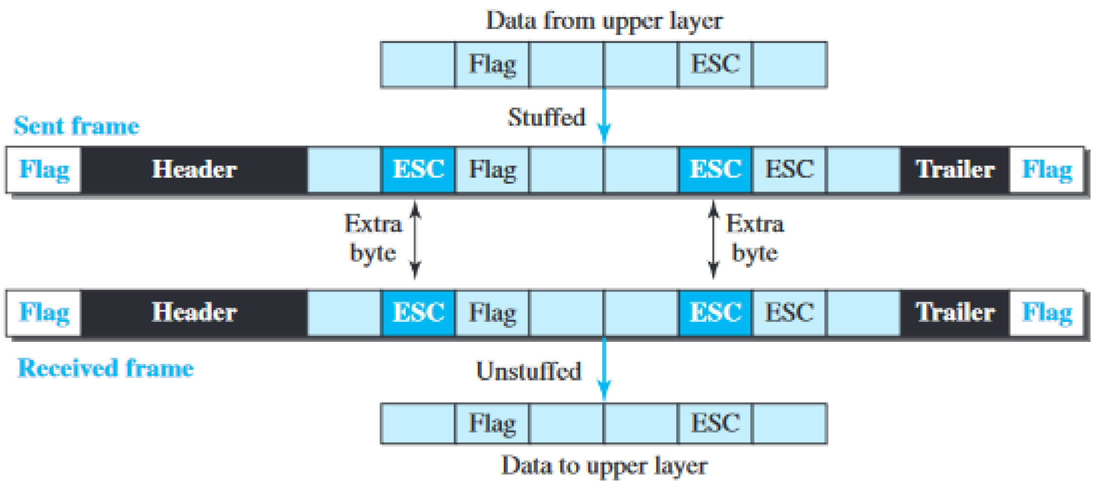
- In character-oriented (or byte-oriented) framing, data to be carried are 8-bit characters from a coding system such as ASCII.
- The header, which normally carries the source and destination addresses and other control information, and the trailer, which carries error detection redundant bits, are also multiples of 8 bits.
- To separate one frame from the next, an 8-bit (1-byte) flag is added at the beginning and the end of a frame. The flag, composed of protocol-dependent special characters, signals the start or end of a frame.

Figure 11.1 *A frame in a character-oriented protocol*



- Any character used for the flag could also be part of the information.
- If this happens, the receiver, when it encounters this pattern in the middle of the data, thinks it has reached the end of the frame.
- To fix this problem, a byte-stuffing strategy was added to character oriented framing.
- In byte stuffing (or character stuffing), a special byte is added to the data section of the frame when there is a character with the same pattern as the flag.
- The data section is stuffed with an extra byte. This byte is usually called the escape character (ESC) and has a predefined bit pattern.
- Whenever the receiver encounters the ESC character, it removes it from the data section and treats the next character as data, not as a delimiting flag.

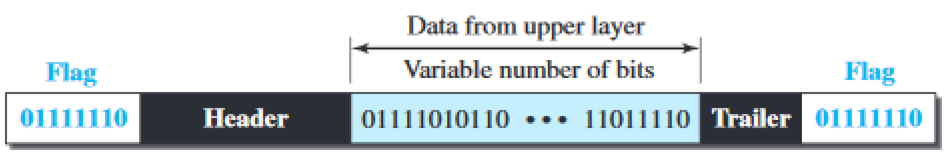
Figure 11.2 *Byte stuffing and unstuffing*



Bit-Oriented Framing

- In bit-oriented framing, the data section of a frame is a sequence of bits to be interpreted by the upper layer as text, graphic, audio, video, and so on.
- In addition to headers (and possible trailers), we still need a delimiter to separate one frame from the other.
- Most protocols use a special 8-bit pattern flag, 01111110, as the delimiter to define the beginning and the end of the frame.

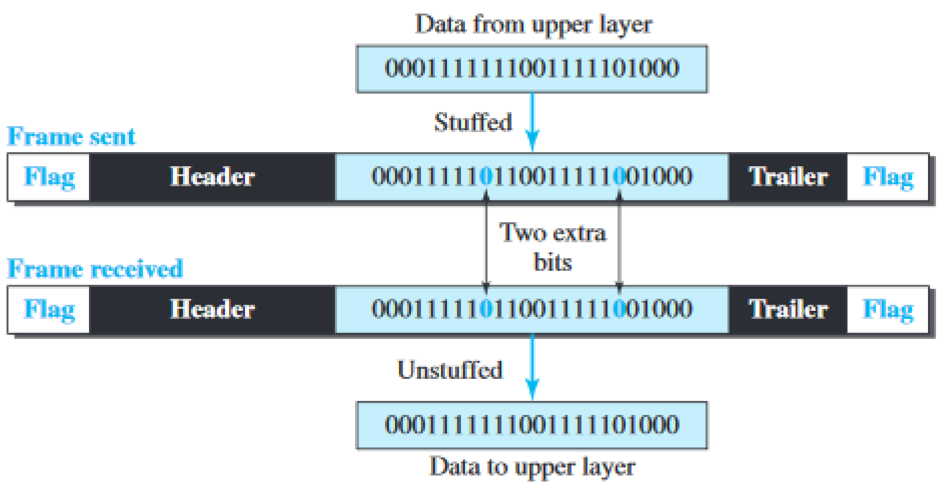
Figure 11.3 *A frame in a bit-oriented protocol*



- If the flag pattern appears in the data, we need to somehow inform the receiver that this is not the end of the frame.
- We do this by stuffing 1 single bit (instead of 1 byte) to prevent the pattern from looking like a flag.
- The strategy is called bit stuffing.
- In bit stuffing, if a 0 and five consecutive 1 bits are encountered, an extra 0 is added.
- This extra stuffed bit is eventually removed from the data by the receiver.
- Note that the extra bit is added after one 0 followed by five 1s regardless of the value of the next bit.
- This guarantees that the flag field sequence does not inadvertently appear in the frame.

Bit stuffing is the process of adding one extra 0 whenever five consecutive 1s follow a 0 in the data, so that the receiver does not mistake the pattern 0111110 for a flag.

Figure 11.4 *Bit stuffing and unstuffing*

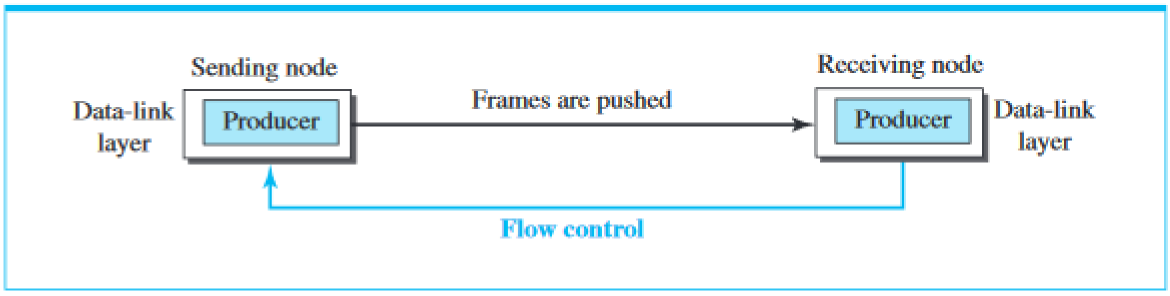


Flow and Error Control

Flow Control

- Whenever an entity produces items and another entity consumes them, there should be a balance between production and consumption rates.
- If the items are produced faster than they can be consumed, the consumer can be overwhelmed and may need to discard some items.
- If the items are produced more slowly than they can be consumed, the consumer must wait, and the system becomes less efficient.
- Flow control is related to the first issue.
- We need to prevent losing the data items at the consumer site.

Figure 11.5 *Flow control at the data-link layer*



The figure shows that the data-link layer at the sending node tries to push frames toward the data-link layer at the receiving node. If the receiving node cannot process and deliver the packet to its network at the same rate that the frames arrive, it becomes overwhelmed with frames. Flow control in this case can be feedback from the receiving node to the sending node to stop or slow down pushing frames.

Buffers

Although flow control can be implemented in several ways, one of the solutions is normally to use two buffers; one at the sending data-link layer and the other at the receiving data-link layer. A buffer is a set of memory locations that can hold packets at the sender and receiver. The flow control communication can occur by sending signals from the consumer to the producer. When the buffer of the receiving data-link layer is full, it informs the sending data-link layer to stop pushing frames.

Connectionless and Connection-Oriented

Connectionless Protocol

- In a connectionless protocol, frames are sent from one node to the next without any relationship between the frames; each frame is independent.
- Connectionless means that there is no connection between frames.
- The frames are not numbered and there is no sense of ordering.
- Most of the data-link protocols for LANs are connectionless protocols.

Connection oriented Protocol

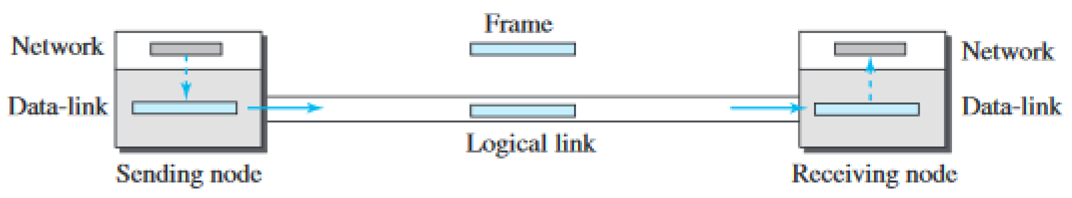
- In a connection-oriented protocol, a logical connection should first be established between the two nodes (setup phase).
- After all frames that are somehow related to each other are transmitted (transfer phase)
- The logical connection is terminated (teardown phase).
- Connection-oriented protocols are rare in wired LANs, but we can see them in some point-to-point protocols, some wireless LANs, and some WANs.

Data Link Layer Protocols

Simple Protocol

- First protocol is a simple protocol with neither flow nor error control.

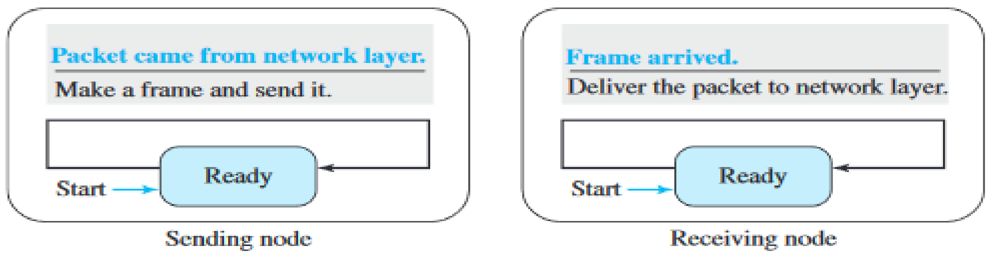
Figure 11.7 *Simple protocol*



- The data-link layer at the sender gets a packet from its network layer, makes a frame out of it, and sends the frame.
- The data-link layer at the receiver receives a frame from the link, extracts the packet from the frame, and delivers the packet to its network layer.
- The data-link layers of the sender and receiver provide transmission services for their network layers.

Finite State Machine (FSM)

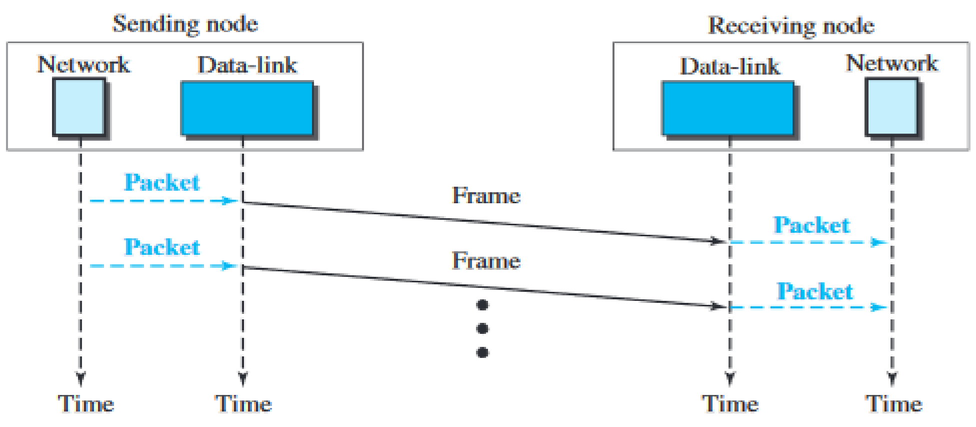
Figure 11.8 *FSMs for the simple protocol*


Packet came from network layer.
Make a frame and send it.
Start
Ready
Sending node
Frame arrived.
Deliver the packet to network layer.
Start
Ready
Receiving node

- The sender site should not send a frame until its network layer has a message to send. The receiver site cannot deliver a message to its network layer until a frame arrives. We can show these requirements using two FSMs.
- Each FSM has only one state, the ready state.
- The sending machine remains in the ready state until a request comes from the process in the network layer.
- When this event occurs, the sending machine encapsulates the message in a frame and sends it to the receiving machine.
- The receiving machine remains in the ready state until a frame arrives from the sending machine.
- When this event occurs, the receiving machine decapsulates the message out of the frame and delivers it to the process at the network layer.

Figure shows an example of communication using this protocol. It is very simple. The sender sends frames one after another without even thinking about the receiver.

Figure 11.9 *Flow diagram for Example 11.2*

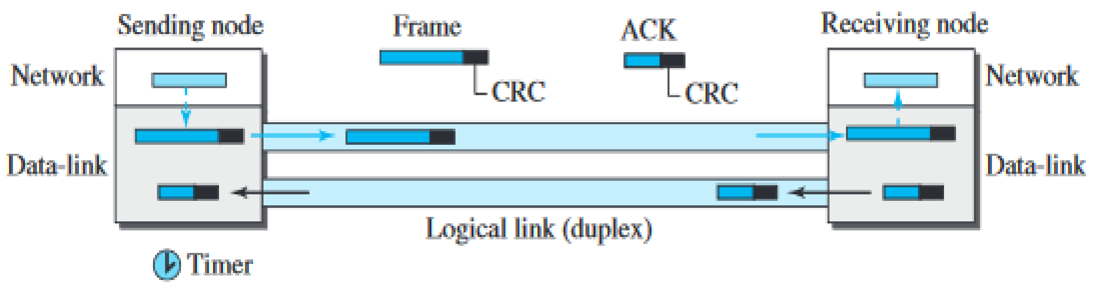


Stop-and-Wait Protocol

- Second protocol is called the Stop-and-Wait protocol, which uses both flow and error control.
- In this protocol, the sender sends one frame at a time and waits for an acknowledgment before sending the next one.
- To detect corrupted frames, we need to add a CRC to each data frame.
- When a frame arrives at the receiver site, it is checked.
- If its CRC is incorrect, the frame is corrupted and silently discarded.
- The silence of the receiver is a signal for the sender that a frame was either corrupted or lost.
- Every time the sender sends a frame, it starts a timer.
- If an acknowledgment arrives before the timer expires, the timer is stopped and the sender sends the next frame (if it has one to send).
- If the timer expires, the sender resends the previous frame, assuming that the frame was either lost or corrupted.

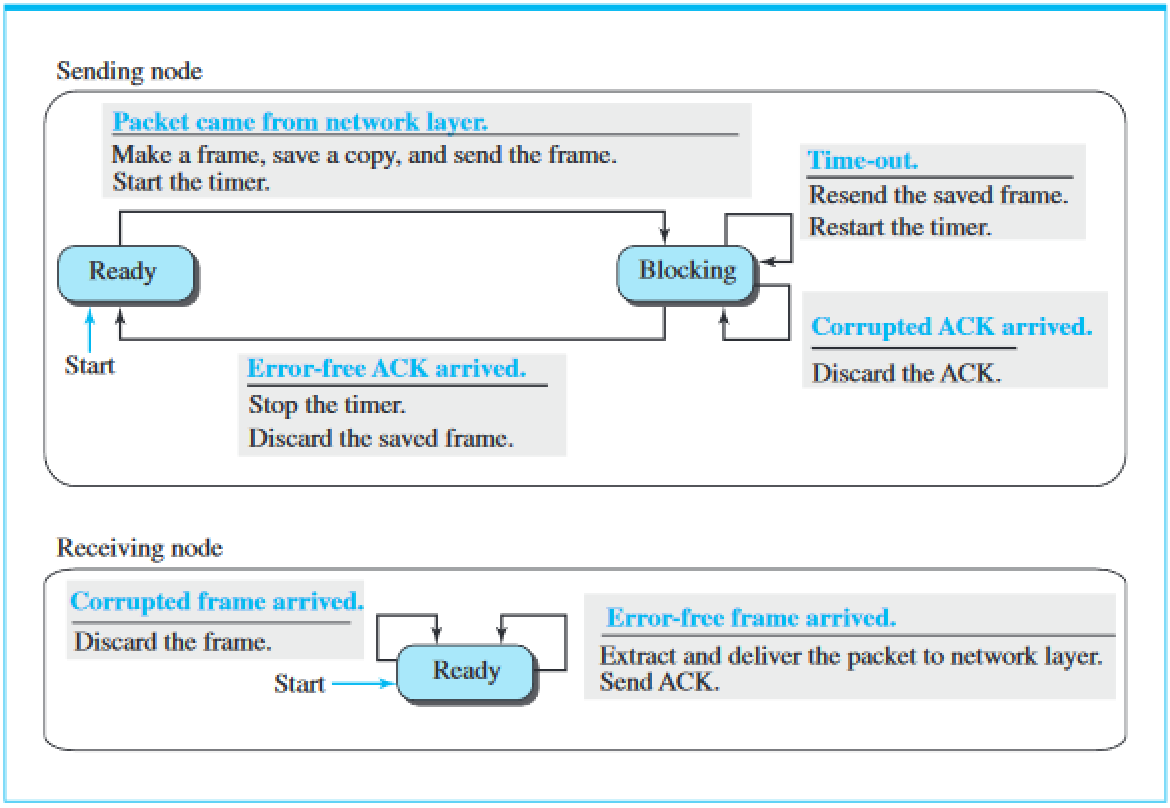
- This means that the sender needs to keep a copy of the frame until its acknowledgment arrives.
- When the corresponding acknowledgment arrives, the sender discards the copy and sends the next frame if it is ready.
- Figure shows the outline for the Stop-and-Wait protocol.
- Note that only one frame and one acknowledgment can be in the channels at any time.

Figure 11.10 *Stop-and-Wait protocol*



FSM

Figure 11.11 *FSM for the Stop-and-Wait protocol*



We describe the sender and receiver states below.

Sender States

The sender is initially in the ready state, but it can move between the ready and blocking state.

Ready State

- When the sender is in this state, it is only waiting for a packet from the network layer.
- If a packet comes from the network layer, the sender creates a frame, saves a copy of the frame, starts the only timer and sends the frame.
- The sender then moves to the blocking state.

Blocking State: When the sender is in this state, three events can occur

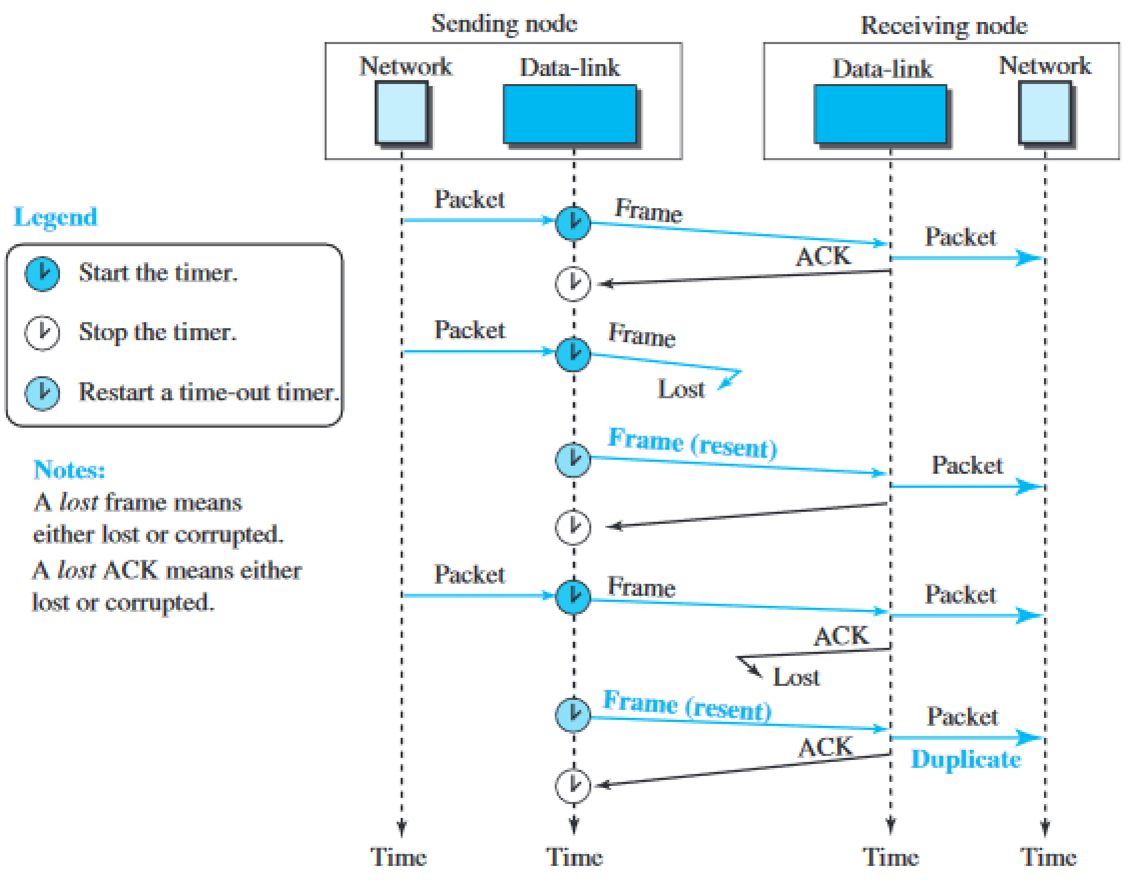
- If a time-out occurs, the sender resends the saved copy of the frame and restarts the timer.
- If a corrupted ACK arrives, it is discarded.
- If an error-free ACK arrives, the sender stops the timer and discards the saved copy of the frame. It then moves to the ready state.

Receiver

The receiver is always in the ready state. Two events may occur:
a. If an error-free frame arrives, the message in the frame is delivered to the network layer and an ACK is sent.
b. If a corrupted frame arrives, the frame is discarded.

Figure shows an example. The first frame is sent and acknowledged. The second frame is sent, but lost. After time-out, it is resent. The third frame is sent and acknowledged, but the acknowledgment is lost. The frame is resent. However, there is a problem with this scheme. The network layer at the receiver site receives two copies of the third packet, which is not right.

Figure 11.12 *Flow diagram for Example 11.3*

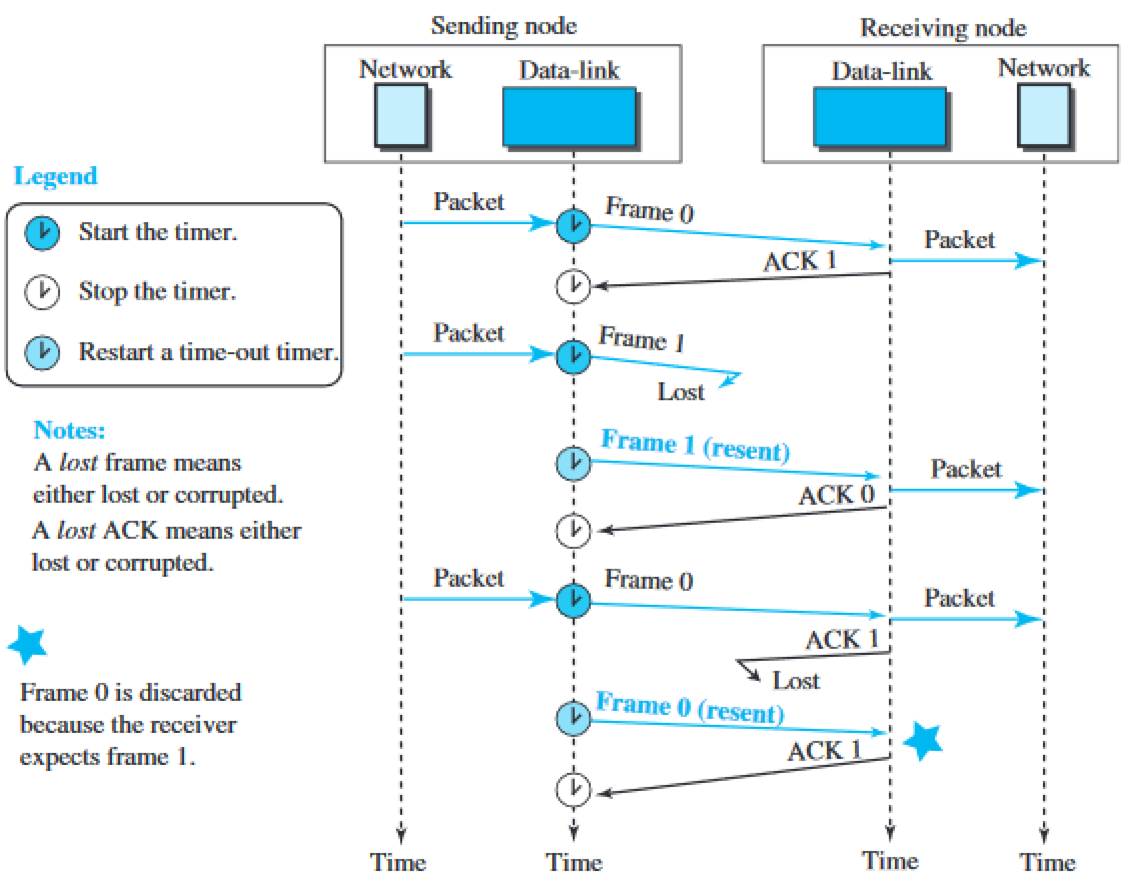


Sequence and Acknowledgment Numbers

- We need to add sequence numbers to the data frames and acknowledgment numbers to the ACK frames.
- Sequence numbers are 0, 1, 0, 1, 0, 1, . . . ; the acknowledgment numbers can also be 1, 0, 1, 0, 1, 0, ...
- In other words, the sequence numbers start with 0, the acknowledgment numbers start with 1. An acknowledgment number always defines the sequence number of the next frame to receive.

Figure shows how adding sequence numbers and acknowledgment numbers can prevent duplicates. The first frame is sent and acknowledged. The second frame is sent, but lost. After time-out, it is resent. The third frame is sent and acknowledged, but the acknowledgment is lost. The frame is resent.

Figure 11.13 *Flow diagram for Example 11.4*



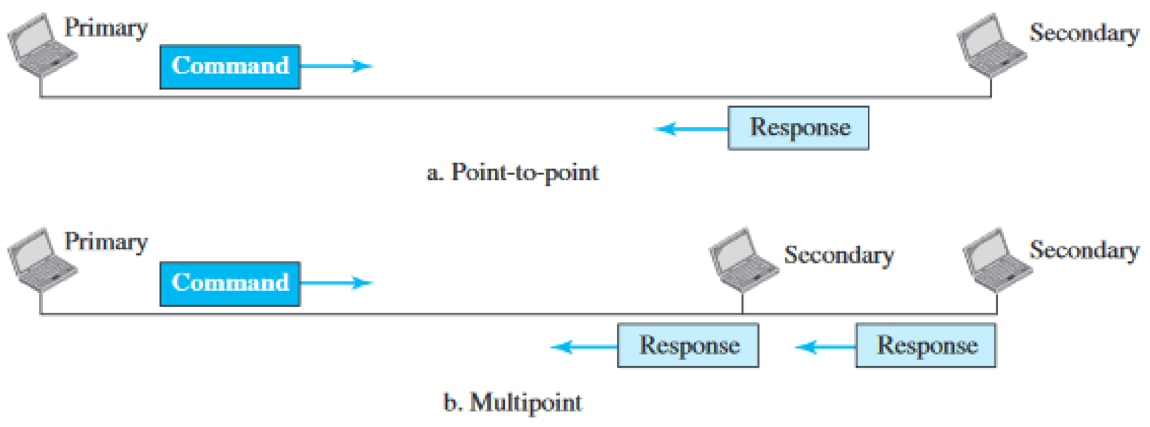
HDLC

High-level Data Link Control (HDLC) is a bit-oriented protocol for communication over point-to-point and multipoint links.

Configurations and Transfer Modes

HDLC provides two common transfer modes that can be used in different configurations: normal response mode (NRM) and asynchronous balanced mode (ABM) .In normal response mode (NRM) , the station configuration is unbalanced. We have one primary station and multiple secondary stations. A primary station can send commands; a secondary station can only respond. The NRM is used for both point-to point and multi point links, as shown in Figure. In ABM, the configuration is balanced. The link is point-to-point, and each station can function as a primary and a secondary

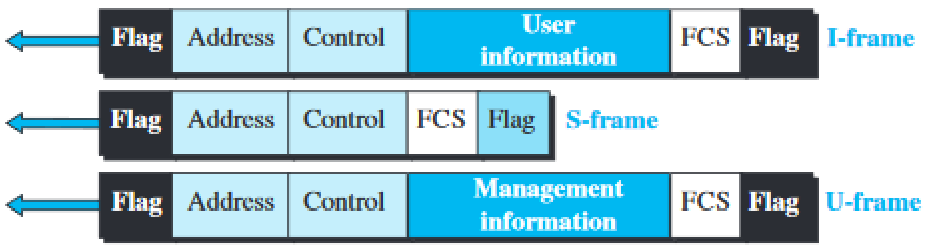
Figure 11.14 *Normal response mode*



Framing

- To provide the flexibility necessary to support all the options possible in the modes and configurations just described, HDLC defines three types of frames:
- Information frames (I-frames) , supervisory frames (S-frames), and unnumbered frames (U frames).
- Each type of frame serves as an envelope for the transmission of a different type of message.
- I-frames are used to data-link user data and control information relating to user data (piggy-backing).
- S-frames are used only to transport control information.
- U frames are reserved for system management.
- Information carried by U-frames is intended for managing the link itself.
- Each frame in HDLC may contain up to six fields, as shown in Figure a beginning flag field, an address field, a control field, an information field, a frame check sequence (FCS) field, and an ending flag field.
- In multiple-frame transmissions, the ending flag of one frame can serve as the beginning flag of the next frame.

Figure 11.16 *HDLC frames*



Flag field

This field contains synchronization pattern 01111110, which identifies both the beginning and the end of a frame.

Address field

This field contains the address of the secondary station. If a primary station created the frame, it contains a to address. If a secondary station creates the frame, it contains a from address. The address field can be one byte or several bytes long, depending on the needs of the network.

Control field

The control field is one or two bytes used for flow and error control.

Information field

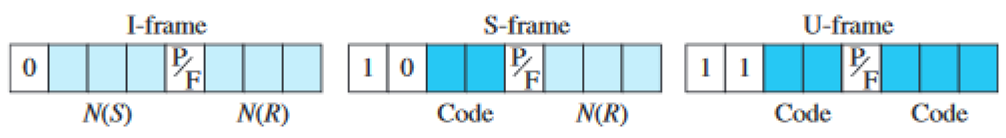
The information field contains the user's data from the network layer or management information. Its length can vary from one network to another.

FCS field

The frame check sequence (FCS) is the HDLC error detection field. It can contain either a 2- or 4-byte CRC.

The control field determines the type of frame and defines its functionality

Figure 11.17 *Control field format for the different frame types*



Control Field for I-Frames

- I-frames are designed to carry user data from the network layer.
- In addition, they can include flow- and error-control information (piggybacking).
- The subfields in the control field are used to define these functions.
- The first bit defines the type.
- If the first bit of the control field is 0, this means the frame is an I-frame.
- The next 3 bits, called N(S), define the sequence number of the frame.
- Note that with 3 bits, we can define a sequence number between 0 and 7.

- The last 3 bits, called N(R), correspond to the acknowledgment number when piggybacking is used.
- The single bit between N(S) and N(R) is called the P/F bit.
- The P/F field is a single bit with a dual purpose.
- It has meaning only when it is set (bit =1) and can mean poll or final.
- It means poll when the frame is sent by a primary station to a secondary (when the address field contains the address of the receiver).
- It means final when the frame is sent by a secondary to a primary (when the address field contains the address of the sender).

Control Field for S-Frames

- Supervisory frames are used for flow and error control whenever piggybacking is either impossible or inappropriate.
- S-frames do not have information fields.
- If the first 2 bits of the control field are 10, this means the frame is an S-frame.
- The last 3 bits, called N(R), correspond to the acknowledgment number (ACK) or negative acknowledgment number (NAK), depending on the type of S-frame.
- The 2 bits called code are used to define the type of S-frame itself.
- With 2 bits, we can have four types of S-frames, as described below:

Receive ready (RR)

- If the value of the code subfield is 00, it is an RR S-frame.
- This kind of frame acknowledges the receipt of a safe and sound frame or group of frames.
- In this case, the value of the N(R) field defines the acknowledgment number.

Receive not ready (RNR)

- If the value of the code subfield is 10, it is an RNR S-frame.
- This kind of frame is an RR frame with additional functions.
- It acknowledges the receipt of a frame or group of frames, and it announces that the receiver is busy and cannot receive more frames. It acts as a kind of congestion control mechanism by asking the sender to slow down. The value of N(R) is the acknowledgment number.

Reject (REJ)

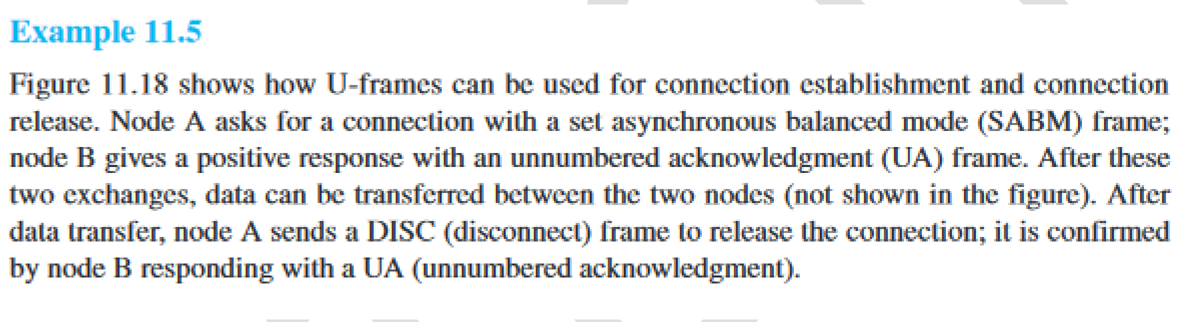
- If the value of the code subfield is 01, it is an REJ S-frame.
- This is a NAK frame, but not like the one used for Selective Repeat ARQ.
- It is a NAK that can be used in Go-Back-N ARQ to improve the efficiency of the process by informing the sender, before the sender timer expires, that the last frame is lost or damaged.
- The value of N(R) is the negative acknowledgment number.

Selective reject (SREJ)

- If the value of the code subfield is 11, it is an SREJ S-frame.
- This is a NAK frame used in Selective Repeat ARQ.
- Note that the HDLC Protocol uses the term selective reject instead of selective repeat.
- The value of N(R) is the negative acknowledgment number.

Control Field for U-Frames

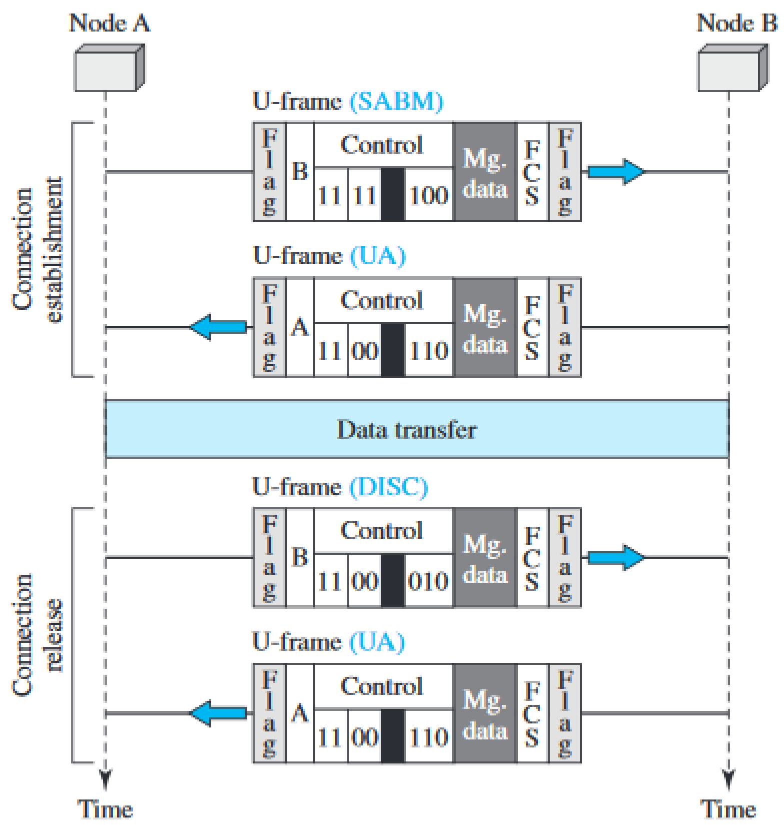
- Unnumbered frames are used to exchange session management and control information between connected devices.
- U-frames contain an information field, but one used for system management information, not user data.
- U-frame codes are divided in to two sections: a 2-bit prefix before the P/F bit and a 3 bit suffix after the P/F bit. Together, these two segments (5 bits) can be used to create up to 32 different types of U-frames.



Example 11.5

Figure 11.18 shows how U-frames can be used for connection establishment and connection release. Node A asks for a connection with a set asynchronous balanced mode (SABM) frame; node B gives a positive response with an unnumbered acknowledgment (UA) frame. After these two exchanges, data can be transferred between the two nodes (not shown in the figure). After data transfer, node A sends a DISC (disconnect) frame to release the connection; it is confirmed by node B responding with a UA (unnumbered acknowledgment).

Figure 11.18 *Example of connection and disconnection*

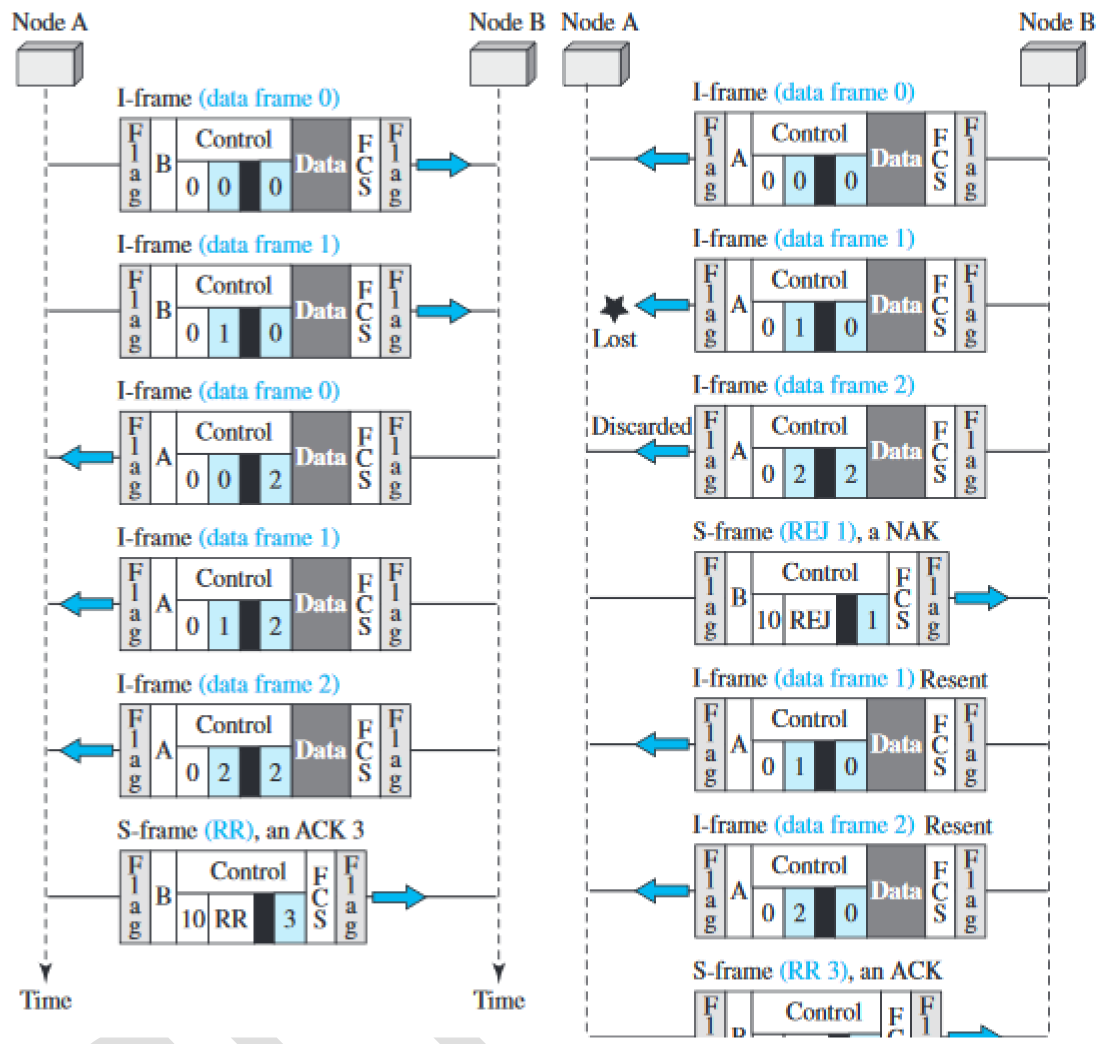


Example

Figure shows two exchanges using piggybacking.

The first is the case where no error has occurred; the second is the case where an error has occurred and some frames are discarded.

Figure 11.19 *Example of piggybacking with and without error*



POINT-TO-POINT PROTOCOL (PPP)

One of the most common protocols for point-to-point access is the Point-to-Point Protocol (PPP).

Services

Services Provided by PPP

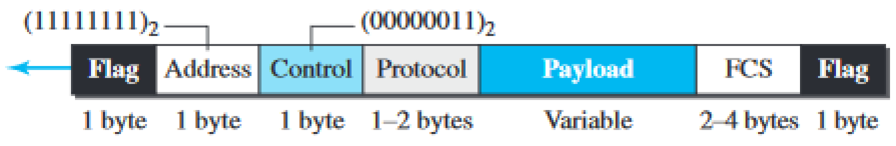
- PPP does not provide flow control.
- A sender can send several frames one after another with no concern about overwhelming the receiver.
- PPP has a very simple mechanism for error control.
- A CRC field is used to detect errors. If the frame is corrupted, it is silently discarded; the upper-layer protocol needs to take care of the problem.
- Lack of error control and sequence numbering may cause a packet to be received out of order.

- PPP does not provide a sophisticated addressing mechanism to handle frames in a multipoint configuration.

Framing

PPP uses a character-oriented (or byte-oriented) frame. Figure shows the format of a PPP frame. The description of each field follows:

Figure 11.20 *PPP frame format*



Address
The address field in this protocol is a constant value and set to 11111111 (broadcast address).

Control
This field is set to the constant value 00000011 (imitating unnumbered frames in HDLC). As we will discuss later, PPP does not provide any flow control. Error control is also limited to error detection.

Protocol
The protocol field defines what is being carried in the data field: either user data or other information. This field is by default 2 bytes long, but the two parties can agree to use only 1 byte.

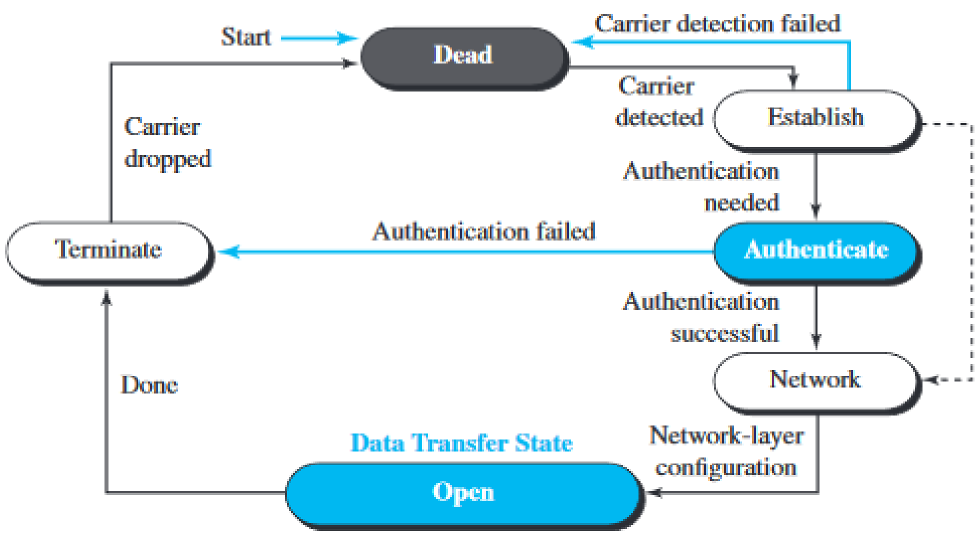
Payload field
This field carries either the user data or other information that we will discuss shortly. The data field is a sequence of bytes with the default of a maximum of 1500 bytes; but this can be changed during negotiation. The data field is byte-stuffed if the flag byte pattern appears in this field. Because there is no field defining the size of the data field, padding is needed if the size is less than the maximum default value or the maximum negotiated value.

FCS
The frame check sequence (FCS) is simply a 2-byte or 4-byte standard CRC.

Transition Phases

- A PPP connection goes through phases which can be shown in a transition phase diagram.
- The transition diagram, which is an FSM, starts with the **dead state**.
- In **dead state**, there is no active carrier (at the physical layer) and the line is quiet. When one of the two nodes starts the communication, the connection goes into the **establish state**.
- In **establish state**, options are negotiated between the two parties.
- If the two parties agree that they need authentication (for example, if they do not know each other), then the system needs to do authentication (an extra step); otherwise, the parties can simply start communication.
- Data transfer takes place in the **open state**.
- When a connection reaches **open state**, the exchange of data packets can be started.
- The connection remains in **open state** until one of the end points wants to terminate the connection.
- In this case, the system goes to the **terminate state**. The system remains in this state until the carrier (physical-layer signal) is dropped, which moves the system to the **dead state** again.

Figure 11.21 *Transition phases*



MODULE 3: ERROR-DETECTION AND CORRECTION

1.Explain two types of errors (4*)

2.Compare error detection vs. error correction (2)
3.Explain error detection using block coding technique. (10*)

4.Explain hamming distance for error detection (6*)

5.Explain parity-check code with block diagram. (6*)
6.Explain CRC with block diagram & an example. (10*)

7.Write short notes on polynomial codes. (5*)

8.Explain internet checksum algorithm along with an example. (6*)
9.Explain the following:
Fletcher checksum and ii) Adler checksum (8)

10.Explain various FEC techniques. (6)

MODULE 3: DATA LINK CONTROL

1.Explain two types of frames. (2)
2.Explain character oriented protocol. (6*)
3.Explain the concept of byte stuffing and unstuffing with example. (6*)

4.Explain bit oriented protocol. (6*)

5.Differentiate between character oriented and bit oriented format for Framing. (6*)
6.Compare flow control and error control. (4)
7.With a neat diagram, explain the design of the simplest protocol with no flow control. (6)

8.Write algorithm for sender site and receiver site for the simplest protocol. (6)

9.Explain Stop-and-Wait protocol (8*)
10.Explain the concept of Piggybacking (2*)

11.Explain in detail HDLC frame format. (8*)

12.Explain 3 type of frame used in HDLC (8*)
13.With a neat schematic, explain the frame structure of PPP protocol. (8*)
14.Explain framing and transition phases in Point-to-Point Protocol. (8*)