Chapter 10

Error Detection and Correction
Data can be corrupted during transmission.

Some applications require that errors be detected and corrected.
Let us first discuss some issues related, directly or indirectly, to error detection and correction.

Topics discussed in this section:
Types of Errors
Redundancy
Detection Versus Correction
Forward Error Correction Versus Retransmission
Coding
Modular Arithmetic
In a single-bit error, only 1 bit in the data unit has changed.
Figure 10.1 *Single-bit error*
A burst error means that 2 or more bits in the data unit have changed.
**Figure 10.2** Burst error of length 8
To detect or correct errors, we need to send extra (redundant) bits with data.
Figure 10.3 The structure of encoder and decoder
In this book, we concentrate on block codes; we leave convolution codes to advanced texts.
In modulo-N arithmetic, we use only the integers in the range 0 to \( N - 1 \), inclusive.
**Figure 10.4** XORing of two single bits or two words

\[
\begin{align*}
0 \oplus 0 &= 0 \\
1 \oplus 1 &= 0
\end{align*}
\]

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

\[
\begin{align*}
0 \oplus 1 &= 1 \\
1 \oplus 0 &= 1
\end{align*}
\]

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

c. Result of XORing two patterns

\[
\begin{align*}
1 & 0 1 1 1 0  \\
\oplus & 1 1 1 0 0 \\
\hline
0 & 1 0 1 0 0
\end{align*}
\]
In block coding, we divide our message into blocks, each of $k$ bits, called **datawords**. We add $r$ redundant bits to each block to make the length $n = k + r$. The resulting $n$-bit blocks are called **codewords**.

**Topics discussed in this section:**
- Error Detection
- Error Correction
- Hamming Distance
- Minimum Hamming Distance
Figure 10.5  *Datawords and codewords in block coding*

- **Datawords**: $2^k$ datawords, each of $k$ bits.
- **Codewords**: $2^n$ codewords, each of $n$ bits (only $2^k$ of them are valid).
The 4B/5B block coding discussed in Chapter 4 is a good example of this type of coding. In this coding scheme, \( k = 4 \) and \( n = 5 \). As we saw, we have \( 2^k = 16 \) datawords and \( 2^n = 32 \) codewords. We saw that 16 out of 32 codewords are used for message transfer and the rest are either used for other purposes or unused.
Figure 10.6  Process of error detection in block coding
Let us assume that \( k = 2 \) and \( n = 3 \). Table 10.1 shows the list of datawords and codewords. Later, we will see how to derive a codeword from a dataword.

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

1. The receiver receives 011. It is a valid codeword. The receiver extracts the dataword 01 from it.
2. The codeword is corrupted during transmission, and 111 is received. This is not a valid codeword and is discarded.

3. The codeword is corrupted during transmission, and 000 is received. This is a valid codeword. The receiver incorrectly extracts the dataword 00. Two corrupted bits have made the error undetectable.
### Table 10.1  A code for error detection (Example 10.2)

<table>
<thead>
<tr>
<th>Datawords</th>
<th>Codewords</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
</tr>
<tr>
<td>10</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
</tr>
</tbody>
</table>
An error-detecting code can detect only the types of errors for which it is designed; other types of errors may remain undetected.
Figure 10.7 Structure of encoder and decoder in error correction
Let us add more redundant bits to Example 10.2 to see if the receiver can correct an error without knowing what was actually sent. We add 3 redundant bits to the 2-bit dataword to make 5-bit codewords. Table 10.2 shows the datawords and codewords. Assume the dataword is 01. The sender creates the codeword 01011. The codeword is corrupted during transmission, and 01001 is received. First, the receiver finds that the received codeword is not in the table. This means an error has occurred. The receiver, assuming that there is only 1 bit corrupted, uses the following strategy to guess the correct dataword.
1. Comparing the received codeword with the first codeword in the table (01001 versus 00000), the receiver decides that the first codeword is not the one that was sent because there are two different bits.

2. By the same reasoning, the original codeword cannot be the third or fourth one in the table.

3. The original codeword must be the second one in the table because this is the only one that differs from the received codeword by 1 bit. The receiver replaces 01001 with 01011 and consults the table to find the dataword 01.
### Table 10.2  A code for error correction (Example 10.3)

<table>
<thead>
<tr>
<th>Dataword</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000000</td>
</tr>
<tr>
<td>01</td>
<td>01011</td>
</tr>
<tr>
<td>10</td>
<td>10101</td>
</tr>
<tr>
<td>11</td>
<td>11110</td>
</tr>
</tbody>
</table>
The Hamming distance between two words is the number of differences between corresponding bits.

Note
Let us find the Hamming distance between two pairs of words.

1. The Hamming distance $d(000, 011)$ is 2 because $000 \oplus 011$ is 011 (two 1s).

2. The Hamming distance $d(10101, 11110)$ is 3 because $10101 \oplus 11110$ is 01011 (three 1s).
The minimum Hamming distance is the smallest Hamming distance between all possible pairs in a set of words.

Note
**Example 10.5**

Find the minimum Hamming distance of the coding scheme in Table 10.1.

**Solution**

We first find all Hamming distances.

\[
\begin{align*}
    d(000, 011) &= 2 & d(000, 101) &= 2 & d(000, 110) &= 2 & d(011, 101) &= 2 \\
    d(011, 110) &= 2 & d(101, 110) &= 2
\end{align*}
\]

*The $d_{\text{min}}$ in this case is 2.*
Example 10.6

Find the minimum Hamming distance of the coding scheme in Table 10.2.

Solution
We first find all the Hamming distances.

\[
\begin{align*}
    d(00000, 01011) &= 3 &
    d(00000, 10101) &= 3 &
    d(00000, 11110) &= 4 \\
    d(01011, 10101) &= 4 &
    d(01011, 11110) &= 3 &
    d(10101, 11110) &= 3
\end{align*}
\]

The \(d_{\text{min}}\) in this case is 3.
To guarantee the detection of up to $s$ errors in all cases, the minimum Hamming distance in a block code must be $d_{\text{min}} = s + 1$. 

**Note**
The minimum Hamming distance for our first code scheme (Table 10.1) is 2. This code guarantees detection of only a single error. For example, if the third codeword (101) is sent and one error occurs, the received codeword does not match any valid codeword. If two errors occur, however, the received codeword may match a valid codeword and the errors are not detected.
Our second block code scheme (Table 10.2) has $d_{\text{min}} = 3$. This code can detect up to two errors. Again, we see that when any of the valid codewords is sent, two errors create a codeword which is not in the table of valid codewords. The receiver cannot be fooled.

However, some combinations of three errors change a valid codeword to another valid codeword. The receiver accepts the received codeword and the errors are undetected.
Figure 10.8 Geometric concept for finding $d_{\text{min}}$ in error detection

Legend
- Any valid codeword
- Any corrupted codeword with 0 to $s$ errors

$d_{\text{min}} > s$
**Figure 10.9** Geometric concept for finding $d_{\text{min}}$ in error correction

- **Legend**
  - Any valid codeword
  - Any corrupted codeword with 1 to $t$ errors

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

*Note*
A code scheme has a Hamming distance $d_{\text{min}} = 4$. What is the error detection and correction capability of this scheme?

**Solution**

This code guarantees the detection of up to **three** errors ($s = 3$), but it can correct up to **one** error. In other words, if this code is used for error correction, part of its capability is wasted. Error correction codes need to have an odd minimum distance (3, 5, 7, . . . ).
Almost all block codes used today belong to a subset called **linear block codes**. A linear block code is a code in which the exclusive OR (addition modulo-2) of two valid codewords creates another valid codeword.

**Topics discussed in this section:**
Minimum Distance for Linear Block Codes
Some Linear Block Codes
In a linear block code, the exclusive OR (XOR) of any two valid codewords creates another valid codeword.
Let us see if the two codes we defined in Table 10.1 and Table 10.2 belong to the class of linear block codes.

1. The scheme in Table 10.1 is a linear block code because the result of XORing any codeword with any other codeword is a valid codeword. For example, the XORing of the second and third codewords creates the fourth one.

2. The scheme in Table 10.2 is also a linear block code. We can create all four codewords by XORing two other codewords.
In our first code (Table 10.1), the numbers of 1s in the nonzero codewords are 2, 2, and 2. So the minimum Hamming distance is $d_{\text{min}} = 2$. In our second code (Table 10.2), the numbers of 1s in the nonzero codewords are 3, 3, and 4. So in this code we have $d_{\text{min}} = 3$. 

Example 10.11
A simple parity-check code is a single-bit error-detecting code in which \( n = k + 1 \) with \( d_{\text{min}} = 2 \).
Table 10.3  *Simple parity-check code C(5, 4)*

<table>
<thead>
<tr>
<th>Datawords</th>
<th>Codewords</th>
<th>Datawords</th>
<th>Codewords</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00000</td>
<td>1000</td>
<td>10001</td>
</tr>
<tr>
<td>0001</td>
<td>00011</td>
<td>1001</td>
<td>10010</td>
</tr>
<tr>
<td>0010</td>
<td>00101</td>
<td>1010</td>
<td>10100</td>
</tr>
<tr>
<td>0011</td>
<td>00110</td>
<td>1011</td>
<td>10111</td>
</tr>
<tr>
<td>0100</td>
<td>01001</td>
<td>1100</td>
<td>11000</td>
</tr>
<tr>
<td>0101</td>
<td>01010</td>
<td>1101</td>
<td>11011</td>
</tr>
<tr>
<td>0110</td>
<td>01100</td>
<td>1110</td>
<td>11101</td>
</tr>
<tr>
<td>0111</td>
<td>01111</td>
<td>1111</td>
<td>11110</td>
</tr>
</tbody>
</table>
**Figure 10.10** Encoder and decoder for simple parity-check code
Let us look at some transmission scenarios. Assume the sender sends the dataword 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 dataword 1011 is created.

2. One single-bit error changes $a_1$. The received codeword is 10011. The syndrome is 1. No dataword is created.

3. One single-bit error changes $r_0$. The received codeword is 10110. The syndrome is 1. No dataword is created.
4. An error changes $r_0$ and a second error changes $a_3$. The received codeword is 00110. The syndrome is 0. The dataword 0011 is created at the receiver. Note that here the dataword is wrongly created due to the syndrome value.

5. Three bits—$a_3$, $a_2$, and $a_1$—are changed by errors. The received codeword is 01011. The syndrome is 1. The dataword is not created. This shows that the simple parity check, guaranteed to detect one single error, can also find any odd number of errors.
A simple parity-check code can detect an odd number of errors.
All Hamming codes discussed in this book have $d_{\text{min}} = 3$.

The relationship between $m$ and $n$ in these codes is $n = 2m - 1$. 

*Note*
Figure 10.11  Two-dimensional parity-check code

a. Design of row and column parities
Figure 10.11  Two-dimensional parity-check code

b. One error affects two parities

c. Two errors affect two parities
Figure 10.11  *Two-dimensional parity-check code*

d. Three errors affect four parities

e. Four errors cannot be detected
<table>
<thead>
<tr>
<th>Datawords</th>
<th>Codewords</th>
<th>Datawords</th>
<th>Codewords</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000000</td>
<td>1000</td>
<td>1000110</td>
</tr>
<tr>
<td>0001</td>
<td>0001101</td>
<td>1001</td>
<td>1001011</td>
</tr>
<tr>
<td>0010</td>
<td>0010111</td>
<td>1010</td>
<td>1010001</td>
</tr>
<tr>
<td>0011</td>
<td>0011010</td>
<td>1011</td>
<td>1011100</td>
</tr>
<tr>
<td>0100</td>
<td>0100011</td>
<td>1100</td>
<td>1100101</td>
</tr>
<tr>
<td>0101</td>
<td>0101110</td>
<td>1101</td>
<td>1101000</td>
</tr>
<tr>
<td>0110</td>
<td>0110100</td>
<td>1110</td>
<td>1110010</td>
</tr>
<tr>
<td>0111</td>
<td>0111001</td>
<td>1111</td>
<td>1111111</td>
</tr>
</tbody>
</table>
Figure 10.12 *The structure of the encoder and decoder for a Hamming code*
Table 10.5  Logical decision made by the correction logic analyzer

<table>
<thead>
<tr>
<th>Syndrome</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>None</td>
<td>$q_0$</td>
<td>$q_1$</td>
<td>$b_2$</td>
<td>$q_2$</td>
<td>$b_0$</td>
<td>$b_3$</td>
<td>$b_1$</td>
</tr>
</tbody>
</table>
Example 10.13

Let us trace the path of three datawords from the sender to the destination:

1. The dataword 0100 becomes the codeword 0100011. The codeword 0100011 is received. The syndrome is 000, the final dataword is 0100.

2. The dataword 0111 becomes the codeword 0111001. The syndrome is 011. After flipping $b_2$ (changing the 1 to 0), the final dataword is 0111.

3. The dataword 1101 becomes the codeword 1101000. The syndrome is 101. After flipping $b_0$, we get 0000, the wrong dataword. This shows that our code cannot correct two errors.
We need a dataword of at least 7 bits. Calculate values of $k$ and $n$ that satisfy this requirement.

**Solution**

We need to make $k = n - m$ greater than or equal to 7, or $2m - 1 - m \geq 7$.

1. If we set $m = 3$, the result is $n = 23 - 1$ and $k = 7 - 3$, or 4, which is not acceptable.

2. If we set $m = 4$, then $n = 24 - 1 = 15$ and $k = 15 - 4 = 11$, which satisfies the condition. So the code is $C(15, 11)$.
Figure 10.13  *Burst error correction using Hamming code*
**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.

**Topics discussed in this section:**
- Cyclic Redundancy Check
- Hardware Implementation
- Polynomials
- Cyclic Code Analysis
- Advantages of Cyclic Codes
- Other Cyclic Codes
Table 10.6  A CRC code with $C(7, 4)$

<table>
<thead>
<tr>
<th>Dataword</th>
<th>Codeword</th>
<th>Dataword</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00000000</td>
<td>1000</td>
<td>1000101</td>
</tr>
<tr>
<td>0001</td>
<td>00010111</td>
<td>1001</td>
<td>10011110</td>
</tr>
<tr>
<td>0010</td>
<td>00101110</td>
<td>1010</td>
<td>10100111</td>
</tr>
<tr>
<td>0011</td>
<td>00111011</td>
<td>1011</td>
<td>10110000</td>
</tr>
<tr>
<td>0100</td>
<td>01001111</td>
<td>1100</td>
<td>11000100</td>
</tr>
<tr>
<td>0101</td>
<td>01011100</td>
<td>1101</td>
<td>11010001</td>
</tr>
<tr>
<td>0110</td>
<td>01100001</td>
<td>1110</td>
<td>11101000</td>
</tr>
<tr>
<td>0111</td>
<td>01110100</td>
<td>1111</td>
<td>11111111</td>
</tr>
</tbody>
</table>
Figure 10.14 *CRC encoder and decoder*
Figure 10.15  Division in CRC encoder

Dataword: 1001

Division

Quotient

Divisor: 1011

Dividend: augmented dataword

Leftmost bit 0: use 0000 divisor

Leftmost bit 0: use 0000 divisor

Remainder

Codeword: 1001110

Dataword: 110
Figure 10.16  Division in the CRC decoder for two cases
Figure 10.17  *Hardwired design of the divisor in CRC*

- Leftmost bit of the part of dividend involved in XOR operation
- Broken line: this bit is always 0
Figure 10.18  Simulation of division in CRC encoder

Time: 1

Time: 2

Time: 3

Time: 4

Time: 5

Time: 6

Time: 7

Augmented dataword

Final remainder
Figure 10.19  The CRC encoder design using shift registers
**Figure 10.20** General design of encoder and decoder of a CRC code

**Note:**

The divisor line and XOR are missing if the corresponding bit in the divisor is 0.

---

**a. Encoder**

- $r_{n-k-1}$
- $d_{n-k-1}$
- $d_1$
- $d_0$
- Dataword

---

**b. Decoder**

- $s_{n-k-1}$
- $d_{n-k-1}$
- $d_1$
- $d_0$
- Received codeword
Figure 10.21  A polynomial to represent a binary word

a. Binary pattern and polynomial

\[1 \times^6 + 0 \times^5 + 0 \times^4 + 0 \times^3 + 0 \times^2 + 1 \times^1 + 1 \times^0\]

b. Short form

\[1 0 0 0 0 0 1 1\]
Figure 10.22 CRC division using polynomials
The divisor in a cyclic code is normally called the generator polynomial or simply the generator.
In a cyclic code,
If $s(x) \neq 0$, one or more bits is corrupted.
If $s(x) = 0$, either

a. No bit is corrupted. or
b. Some bits are corrupted, but the decoder failed to detect them.
In a cyclic code, those $e(x)$ errors that are divisible by $g(x)$ are not caught.
If the generator has more than one term and the coefficient of $x^0$ is 1, all single errors can be caught.
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?

\begin{itemize}
\item[a.] \( x + 1 \)
\item[b.] \( x^3 \)
\item[c.] 1
\end{itemize}

Solution

\begin{itemize}
\item[a.] No \( x^i \) can be divisible by \( x + 1 \). Any single-bit error can be caught.
\item[b.] If \( i \) is equal to or greater than 3, \( x^i \) is divisible by \( g(x) \). All single-bit errors in positions 1 to 3 are caught.
\item[c.] All values of \( i \) make \( x^i \) divisible by \( g(x) \). No single-bit error can be caught. This \( g(x) \) is useless.
\end{itemize}
Figure 10.23  Representation of two isolated single-bit errors using polynomials

\[ x^{n-1} \quad x^j \quad x^i \quad x^0 \]

Difference: \( j - i \)
If a generator cannot divide $x^t + 1$ (t between 0 and n – 1), then all isolated double errors can be detected.
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.

c. This is a good choice for this purpose.

d. This polynomial cannot divide $x^t + 1$ if $t$ is less than 32,768. A codeword with two isolated errors up to 32,768 bits apart can be detected by this generator.
A generator that contains a factor of $x + 1$ can detect all odd-numbered errors.
All burst errors with $L \leq r$ will be detected.

All burst errors with $L = r + 1$ will be detected with probability $1 - (1/2)^{r-1}$.

All burst errors with $L > r + 1$ will be detected with probability $1 - (1/2)^r$. 

Note
Example 10.17

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.
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 $x + 1$. 

Note
Table 10.7 *Standard polynomials*

<table>
<thead>
<tr>
<th>Name</th>
<th>Polynomial</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC-8</td>
<td>$x^8 + x^2 + x + 1$</td>
<td>ATM header</td>
</tr>
<tr>
<td>CRC-10</td>
<td>$x^{10} + x^9 + x^5 + x^4 + x^2 + 1$</td>
<td>ATM AAL</td>
</tr>
<tr>
<td>CRC-16</td>
<td>$x^{16} + x^{12} + x^5 + 1$</td>
<td>HDLC</td>
</tr>
<tr>
<td>CRC-32</td>
<td>$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$</td>
<td>LANs</td>
</tr>
</tbody>
</table>
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.

Topics discussed in this section:

- Idea
- One’s Complement
- Internet Checksum
Suppose our data 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 data are not accepted.
We can make the job of the receiver easier if we send the negative (complement) of the sum, called the checksum. In this case, we send (7, 11, 12, 0, 6, −36). The receiver can add all the numbers received (including the checksum). If the result is 0, it assumes no error; otherwise, there is an error.
Example 10.20

How can we represent the number 21 in one’s complement arithmetic using only four bits?

Solution

The number 21 in binary is 10101 (it needs five bits). We can wrap the leftmost bit and add it to the four rightmost bits. We have \((0101 + 1) = 0110\) or 6.
How can we represent the number $-6$ in one’s complement arithmetic using only four bits?

Solution

In one’s complement arithmetic, the negative or complement of a number is found by inverting all bits. Positive 6 is 0110; negative 6 is 1001. If we consider only unsigned numbers, this is 9. In other words, the complement of 6 is 9. Another way to find the complement of a number in one’s complement arithmetic is to subtract the number from $2^n - 1$ (16 – 1 in this case).
Let us redo Exercise 10.19 using one’s complement arithmetic. Figure 10.24 shows the process at the sender and at the receiver. The sender initializes the checksum to 0 and adds all data items and the checksum (the checksum is considered as one data item and is shown in color). The result is 36. However, 36 cannot be expressed in 4 bits. The extra two bits are wrapped and added with the sum to create the wrapped sum value 6. In the figure, we have shown the details in binary. The sum is then complemented, resulting in the checksum value 9 (15 − 6 = 9). The sender now sends six data items to the receiver including the checksum 9.
Example 10.22 (continued)

The receiver follows the same procedure as the sender. It adds all data items (including the checksum); the result is 45. The sum is wrapped and becomes 15. The wrapped sum is complemented and becomes 0. Since the value of the checksum is 0, this means that the data is not corrupted. The receiver drops the checksum and keeps the other data items. If the checksum is not zero, the entire packet is dropped.
Figure 10.24  Example 10.22

Sender site

<table>
<thead>
<tr>
<th>Sum</th>
<th>36</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrapped sum</td>
<td>6</td>
</tr>
<tr>
<td>Checksum</td>
<td>9</td>
</tr>
</tbody>
</table>

Receiver site

<table>
<thead>
<tr>
<th>Sum</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrapped sum</td>
<td>15</td>
</tr>
<tr>
<td>Checksum</td>
<td>0</td>
</tr>
</tbody>
</table>

Packet

\[ 7, 11, 12, 0, 6, 9 \]

Details of wrapping and complementing

Sender site

\[ 100100 \]

\[ \rightarrow 10 \]

\[ 0110 \]

\[ 6 \]

\[ 1000 \]

\[ 9 \]

Receiver site

\[ 101101 \]

\[ \rightarrow 10 \]

\[ 0110 \]

\[ 15 \]

\[ 1000 \]

\[ 0 \]
Sender site:

1. The message is divided into 16-bit words.
2. The value of the checksum word is set to 0.
3. All words including the checksum are added using one’s complement addition.
4. The sum is complemented and becomes the checksum.
5. The checksum is sent with the data.
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.
Let us calculate the checksum for a text of 8 characters (“Forouzan”). The text needs to be divided into 2-byte (16-bit) words. We use ASCII (see Appendix A) to change each byte to a 2-digit hexadecimal number. For example, F is represented as 0x46 and o is represented as 0x6F. Figure 10.25 shows how the checksum is calculated at the sender and receiver sites. In part a of the figure, the value of partial sum for the first column is 0x36. We keep the rightmost digit (6) and insert the leftmost digit (3) as the carry in the second column. The process is repeated for each column. Note that if there is any corruption, the checksum recalculated by the receiver is not all 0s. We leave this as an exercise.
Figure 10.25  Example 10.23

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>Carries</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>6</td>
<td>F</td>
<td>(Fo)</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>(ro)</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>7</td>
<td>A</td>
<td>(uz)</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>6</td>
<td>E</td>
<td>(an)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Checksum (initial)</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>C</td>
<td>6</td>
<td>Sum (partial)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>C</td>
<td>7</td>
<td>Sum</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>Checksum (to send)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>Carries</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>6</td>
<td>F</td>
<td>(Fo)</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>(ro)</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>7</td>
<td>A</td>
<td>(uz)</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>6</td>
<td>E</td>
<td>(an)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Checksum (received)</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>Sum (partial)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>C</td>
<td>7</td>
<td>Sum</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>Checksum (new)</td>
</tr>
</tbody>
</table>

a. Checksum at the sender site

a. Checksum at the receiver site
Chapter 11

Data Link Control
The data link layer needs to pack bits into frames, so that each frame is distinguishable from another. Our postal system practices a type of framing. The simple act of inserting a letter into an envelope separates one piece of information from another; the envelope serves as the delimiter.

**Topics discussed in this section:**
Fixed-Size Framing
Variable-Size Framing
**Figure 11.1** A *frame in a character-oriented protocol*
Figure 11.2 *Byte stuffing and unstuffing*

![Diagram of byte stuffing and unstuffing](image.png)
Byte stuffing is the process of adding 1 extra byte whenever there is a flag or escape character in the text.
Figure 11.3 A frame in a bit-oriented protocol

![Frame in a bit-oriented protocol](image)
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

Frame sent

Flag | Header | Data from upper layer |
--- | --- | ---
Flag | Header | 000111111101000111101000 |

Stuffed

Frame received

Flag | Header | Data to upper layer |
--- | --- | ---
Flag | Header | 000111111101000111101000 |

Unstuffed

Extra 2 bits
The most important responsibilities of the data link layer are flow control and error control. Collectively, these functions are known as data link control.

Topics discussed in this section:
Flow Control
Error Control
Flow control refers to a set of procedures used to restrict the amount of data that the sender can send before waiting for acknowledgment.
Note

Error control in the data link layer is based on automatic repeat request, which is the retransmission of data.
Now let us see how the data link layer can combine framing, flow control, and error control to achieve the delivery of data from one node to another. The protocols are normally implemented in software by using one of the common programming languages. To make our discussions language-free, we have written in pseudocode a version of each protocol that concentrates mostly on the procedure instead of delving into the details of language rules.
Figure 11.5  Taxonomy of protocols discussed in this chapter

- Protocols
  - For noiseless channel
    - Simplest
    - Stop-and-Wait
  - For noisy channel
    - Stop-and-Wait ARQ
    - Go-Back-N ARQ
    - Selective Repeat ARQ
Let us first assume we have an ideal channel in which no frames are lost, duplicated, or corrupted. We introduce two protocols for this type of channel.

Topics discussed in this section:
Simplest Protocol
Stop-and-Wait Protocol
Figure 11.6 *The design of the simplest protocol with no flow or error control*
Algorithm 11.1 Sender-site algorithm for the simplest protocol

```plaintext
while(true) // Repeat forever
{
    WaitForEvent(); // Sleep until an event occurs
    if(Event(RequestToSend)) // There is a packet to send
    {
        GetData();
        MakeFrame();
        SendFrame(); // Send the frame
    }
}
```
Algorithm 11.2  Receiver-site algorithm for the simplest protocol

1. while (true)  // Repeat forever
2. {
3.  WaitForEvent();  // Sleep until an event occurs
4.  if (Event(ArrivalNotification))  // Data frame arrived
5.  {
6.    ReceiveFrame();
7.    ExtractData();
8.    DeliverData();  // Deliver data to network layer
9.  }
10.}

11.17
Figure 11.7 shows an example of communication using this protocol. It is very simple. The sender sends a sequence of frames without even thinking about the receiver. To send three frames, three events occur at the sender site and three events at the receiver site. Note that the data frames are shown by tilted boxes; the height of the box defines the transmission time difference between the first bit and the last bit in the frame.
Figure 11.7 Flow diagram for Example 11.1
Figure 11.8 Design of Stop-and-Wait Protocol
Algorithm 11.3  Sender-site algorithm for Stop-and-Wait Protocol

1. while(true) //Repeat forever
2. canSend = true //Allow the first frame to go
3. {
4.    WaitForEvent(); // Sleep until an event occurs
5.    if(Event(RequestToSend) AND canSend)
6.    {
7.        GetData();
8.        MakeFrame();
9.        SendFrame(); //Send the data frame
10.       canSend = false; //Cannot send until ACK arrives
11.    }
12.   WaitForEvent(); // Sleep until an event occurs
13.   if(Event(ArrivalNotification) // An ACK has arrived
14.   {
15.        ReceiveFrame(); //Receive the ACK frame
16.       canSend = true;
17.   }
18. }

11.21
Algorithm 11.4  Receiver-site algorithm for Stop-and-Wait Protocol

1. while(true)  //Repeat forever
2. {
3.     WaitForEvent();  // Sleep until an event occurs
4.     if(Event(ArrivalNotification))  //Data frame arrives
5.     {
6.         ReceiveFrame();
7.         ExtractData();
8.         Deliver(data);  //Deliver data to network layer
9.         SendFrame();  //Send an ACK frame
10.    }
11. }

11.22
Figure 11.9 shows an example of communication using this protocol. It is still very simple. The sender sends one frame and waits for feedback from the receiver. When the ACK arrives, the sender sends the next frame. Note that sending two frames in the protocol involves the sender in four events and the receiver in two events.
Figure 11.9  Flow diagram for Example 11.2
Although the Stop-and-Wait Protocol gives us an idea of how to add flow control to its predecessor, noiseless channels are nonexistent. We discuss three protocols in this section that use error control.

Topics discussed in this section:
- Stop-and-Wait Automatic Repeat Request
- Go-Back-N Automatic Repeat Request
- Selective Repeat Automatic Repeat Request
Error correction in Stop-and-Wait ARQ is done by keeping a copy of the sent frame and retransmitting of the frame when the timer expires.
In Stop-and-Wait ARQ, we use sequence numbers to number the frames. The sequence numbers are based on modulo-2 arithmetic.
In Stop-and-Wait ARQ, the acknowledgment number always announces in modulo-2 arithmetic the sequence number of the next frame expected.
Figure 11.10  Design of the Stop-and-Wait ARQ Protocol
Algorithm 11.5  Sender-site algorithm for Stop-and-Wait ARQ

1. $S_n = 0$; // Frame 0 should be sent first
2. canSend = true; // Allow the first request to go
3. while (true) // Repeat forever
4. {
5.     WaitForEvent(); // Sleep until an event occurs
6.     if (Event(RequestToSend) AND canSend)
7.         {
8.             GetData();
9.             MakeFrame($S_n$); // The seqNo is $S_n$
10.            StoreFrame($S_n$); // Keep copy
11.            SendFrame($S_n$);
12.            StartTimer();
13.            $S_n = S_n + 1$;
14.            canSend = false;
15.         }
16.     WaitForEvent(); // Sleep

(continued)
Algorithm 11.5  Sender-site algorithm for Stop-and-Wait ARQ  (continued)

17       if(Event(ArrivalNotification))  // An ACK has arrived 
18          {
19              ReceiveFrame(ackNo);  // Receive the ACK frame
20              if(not corrupted AND ackNo == S_n)  // Valid ACK
21                  {
22                      StopTimer();
23                      PurgeFrame(S_{n-1});  // Copy is not needed
24                      canSend = true;
25                  }
26          }
27
28       if(Event(TimeOut))  // The timer expired 
29          {
30              StartTimer();
31              ResendFrame(S_{n-1});  // Resend a copy check
32          }
Algorithm 11.6  Receiver-site algorithm for Stop-and-Wait ARQ Protocol

1. $R_n = 0$;  // Frame 0 expected to arrive first
2. while(true)
3. {
4.   WaitForEvent();  // Sleep until an event occurs
5.   if(Event(ArrivalNotification)) //Data frame arrives
6.   {
7.     ReceiveFrame();
8.     if(corrupted(frame));
9.       sleep();
10.    if(seqNo == $R_n$) //Valid data frame
11.        {
12.          ExtractData();
13.          DeliverData(); //Deliver data
14.          $R_n = R_n + 1$;
15.        }
16.     SendFrame($R_n$); //Send an ACK
17.   }
18. }
Figure 11.11 shows an example of Stop-and-Wait ARQ. Frame 0 is sent and acknowledged. Frame 1 is lost and resent after the time-out. The resent frame 1 is acknowledged and the timer stops. Frame 0 is sent and acknowledged, but the acknowledgment is lost. The sender has no idea if the frame or the acknowledgment is lost, so after the time-out, it resends frame 0, which is acknowledged.
Figure 11.11  Flow diagram for Example 11.3
Assume that, in a Stop-and-Wait ARQ system, the bandwidth of the line is 1 Mbps, and 1 bit takes 20 ms to make a round trip. What is the bandwidth-delay product? If the system data frames are 1000 bits in length, what is the utilization percentage of the link?

Solution

The bandwidth-delay product is

\[(1 \times 10^6) \times (20 \times 10^{-3}) = 20,000 \text{ bits}\]
The system can send 20,000 bits during the time it takes for the data to go from the sender to the receiver and then back again. However, the system sends only 1000 bits. We can say that the link utilization is only 1000/20,000, or 5 percent. For this reason, for a link with a high bandwidth or long delay, the use of Stop-and-Wait ARQ wastes the capacity of the link.
Example 11.5

What is the utilization percentage of the link in Example 11.4 if we have a protocol that can send up to 15 frames before stopping and worrying about the acknowledgments?

Solution
The bandwidth-delay product is still 20,000 bits. The system can send up to 15 frames or 15,000 bits during a round trip. This means the utilization is 15,000/20,000, or 75 percent. Of course, if there are damaged frames, the utilization percentage is much less because frames have to be resent.
In the Go-Back-N Protocol, the sequence numbers are modulo $2^m$, where $m$ is the size of the sequence number field in bits.
Figure 11.12  Send window for Go-Back-N ARQ

Frames already acknowledged  | Frames sent, but not acknowledged (outstanding)  | Frames that can be sent, but not received from upper layer  | Frames that cannot be sent

Send window, size $S_{\text{size}} = 2^m - 1$

a. Send window before sliding

b. Send window after sliding
The send window is an abstract concept defining an imaginary box of size $2^m - 1$ with three variables: $S_f$, $S_n$, and $S_{size}$. 

**Note**
The send window can slide one or more slots when a valid acknowledgment arrives.
**Figure 11.13** Receive window for Go-Back-N ARQ

a. Receive window

b. Window after sliding
The receive window is an abstract concept defining an imaginary box of size 1 with one single variable $R_n$. The window slides when a correct frame has arrived; sliding occurs one slot at a time.
Figure 11.14  Design of Go-Back-N ARQ
Figure 11.15  Window size for Go-Back-N ARQ

a. Window size < $2^m$

b. Window size = $2^m$
In Go-Back-N ARQ, the size of the send window must be less than $2^m$; the size of the receiver window is always 1.
Algorithm 11.7  Go-Back-N sender algorithm

1  $S_w = 2^m - 1$;
2  $S_f = 0$;
3  $S_n = 0$;
4
5  while (true)  
6      {  
7      WaitForEvent();  
8      if(Event(RequestToSend()))  
9          {  
10         if($S_n - S_f \geq S_w$)  
11             {  
12             Sleep();  
13             GetData();  
14             MakeFrame($S_n$);  
15             StoreFrame($S_n$);  
16             SendFrame($S_n$);  
17             $S_n = S_n + 1$;
18             if(timer not running)
19                 StartTimer();
20         }
21     }
22
23 (continued)
if(Event(ArrivalNotification))  //ACK arrives
{
    Receive(ACK);
    if(corruped(ACK))
        Sleep();
    if((ackNo>S_f)&&(ackNo<=S_n))  //If a valid ACK
        While(S_f <= ackNo)
        {
            PurgeFrame(S_f);
            S_f = S_f + 1;
        }
    StopTimer();
}

if(Event(TimeOut))  //The timer expires
{
    StartTimer();
    Temp = S_f;
    while(Temp < S_n);
    {
        SendFrame(S_f);
        S_f = S_f + 1;
    }
}
Algorithm 11.8  Go-Back-N receiver algorithm

```plaintext
1  \( R_n = 0; \)
2
3  while (true)  //Repeat forever
4  {
5    WaitForEvent();
6
7    if(Event(ArrivalNotification))  //Data frame arrives
8    {
9      Receive(Frame);
10     if(corrupted(Frame))
11        Sleep();
12     if(seqNo == R_n)  //If expected frame
13     {
14       DeliverData();  //Deliver data
15       R_n = R_n + 1;  //Slide window
16       SendACK(R_n);
17     }
18  }
19  
11.49
```
Figure 11.16 shows an example of Go-Back-N. This is an example of a case where the forward channel is reliable, but the reverse is not. No data frames are lost, but some ACKs are delayed and one is lost. The example also shows how cumulative acknowledgments can help if acknowledgments are delayed or lost. After initialization, there are seven sender events. Request events are triggered by data from the network layer; arrival events are triggered by acknowledgments from the physical layer. There is no time-out event here because all outstanding frames are acknowledged before the timer expires. Note that although ACK 2 is lost, ACK 3 serves as both ACK 2 and ACK 3.
Figure 11.16  Flow diagram for Example 11.6
Figure 11.17 shows what happens when a frame is lost. Frames 0, 1, 2, and 3 are sent. However, frame 1 is lost. The receiver receives frames 2 and 3, but they are discarded because they are received out of order. The sender receives no acknowledgment about frames 1, 2, or 3. Its timer finally expires. The sender sends all outstanding frames (1, 2, and 3) because it does not know what is wrong. Note that the resending of frames 1, 2, and 3 is the response to one single event. When the sender is responding to this event, it cannot accept the triggering of other events. This means that when ACK 2 arrives, the sender is still busy with sending frame 3.
The physical layer must wait until this event is completed and the data link layer goes back to its sleeping state. We have shown a vertical line to indicate the delay. It is the same story with ACK 3; but when ACK 3 arrives, the sender is busy responding to ACK 2. It happens again when ACK 4 arrives. Note that before the second timer expires, all outstanding frames have been sent and the timer is stopped.
Figure 11.17  Flow diagram for Example 11.7
Stop-and-Wait ARQ is a special case of Go-Back-N ARQ in which the size of the send window is 1.
Figure 11.18  Send window for Selective Repeat ARQ

Send window, first outstanding frame  \( S_f \)  

Send window, next frame to send  \( S_n \)

Frames already acknowledged  

Frames sent, but not acknowledged  

Frames that can be sent  

Frames that cannot be sent

\[ S_{\text{size}} = 2^{m-1} \]
Figure 11.19  Receive window for Selective Repeat ARQ

Frames already received

Frames that can be received and stored for later delivery. Colored boxes, already received

Frames that cannot be received

$R_{size} = 2^{m-1}$
Figure 11.20 *Design of Selective Repeat ARQ*

The figure illustrates the design of Selective Repeat ARQ (Automatic Repeat Request). The network layer requests data from the sender. The sender then sends a packet, and the receiver acknowledges or requests a retransmission based on the received data. The process repeats until all data is successfully delivered, with timeouts and notifications from the physical layer.
Figure 11.21 Selective Repeat ARQ, window size

a. Window size = $2^{m-1}$

b. Window size > $2^{m-1}$
In Selective Repeat ARQ, the size of the sender and receiver window must be at most one-half of $2^m$. 

*Note*
Algorithm 11.9  Sender-site Selective Repeat algorithm

1. $S_w = 2^{m-1}$
2. $S_f = 0$
3. $S_n = 0$

4. while (true)  \hspace{1cm} //Repeat forever
5. {
6.     WaitForEvent();
7.     if(Event(RequestToSend()))  \hspace{1cm} //There is a packet to send
8.         {
9.             if($S_n - S_f >= S_w$)  \hspace{1cm} //If window is full
10.                Sleep();
11.                GetData();
12.                MakeFrame($S_n$);
13.                StoreFrame($S_n$);
14.                SendFrame($S_n$);
15.                $S_n = S_n + 1$;
16.                StartTimer($S_n$);
17.         }
18. }

(continued)
Algorithm 11.9  *Sender-site Selective Repeat algorithm*  

(continued)

if(Event(ArrivalNotification)) //ACK arrives
{
    Receive(frame);  //Receive ACK or NAK
    if(corrupted(frame))
        Sleep();
    if (FrameType == NAK)
        if (nakNo between $S_f$ and $S_n$)
            {  
                resend(nakNo);
                StartTimer(nakNo);
            }
    if (FrameType == ACK)
        if (ackNo between $S_f$ and $S_n$)
            {  
                while($s_f < \text{ackNo}$)
                    {  
                        Purge($s_f$);
                        StopTimer($s_f$);
                        $S_f = S_f + 1$;
                    }
            }
}
Algorithm 11.9 Sender-site Selective Repeat algorithm (continued)

42 if(Event(TimeOut(t))) //The timer expires
43 {
44     StartTimer(t);
45     SendFrame(t);
46 }
48"}
Algorithm 11.10  Receiver-site Selective Repeat algorithm

1  \( R_n = 0; \)
2  NakSent = false;
3  AckNeeded = false;
4  Repeat (for all slots)
5      Marked(slot) = false;
6
7  while (true)  //Repeat forever
8  {
9      WaitForEvent();
10
11     if(Event(ArrivalNotification))  //Data frame arrives
12        {
13          Receive(Frame);
14          if(corrupted(Frame)) && (NOT NakSent)
15              {
16              SendNAK(R_n);
17              NakSent = true;
18              Sleep();
19          }
20          if(seqNo <> R_n) && (NOT NakSent)
21              {
22              SendNAK(R_n);
23          }
Algorithm 11.10  Receiver-site Selective Repeat algorithm

NakSent = true;
if ((seqNo in window) && (!Marked(seqNo))
 {
    StoreFrame(seqNo)
    Marked(seqNo) = true;
    while (Marked(R_n))
    {
        DeliverData(R_n);
        Purge(R_n);
        R_n = R_n + 1;
        AckNeeded = true;
    }
    if (AckNeeded);
    {
        SendAck(R_n);
        AckNeeded = false;
        NakSent = false;
    }
}
}

11.65
Figure 11.22  Delivery of data in Selective Repeat ARQ

a. Before delivery

b. After delivery

ackNo sent: 3
This example is similar to Example 11.3 in which frame 1 is lost. We show how Selective Repeat behaves in this case. Figure 11.23 shows the situation. One main difference is the number of timers. Here, each frame sent or resent needs a timer, which means that the timers need to be numbered (0, 1, 2, and 3). The timer for frame 0 starts at the first request, but stops when the ACK for this frame arrives. The timer for frame 1 starts at the second request, restarts when a NAK arrives, and finally stops when the last ACK arrives. The other two timers start when the corresponding frames are sent and stop at the last arrival event.
At the receiver site we need to distinguish between the acceptance of a frame and its delivery to the network layer. At the second arrival, frame 2 arrives and is stored and marked, but it cannot be delivered because frame 1 is missing. At the next arrival, frame 3 arrives and is marked and stored, but still none of the frames can be delivered. Only at the last arrival, when finally a copy of frame 1 arrives, can frames 1, 2, and 3 be delivered to the network layer. There are two conditions for the delivery of frames to the network layer: First, a set of consecutive frames must have arrived. Second, the set starts from the beginning of the window.
Another important point is that a NAK is sent after the second arrival, but not after the third, although both situations look the same. The reason is that the protocol does not want to crowd the network with unnecessary NAKs and unnecessary resent frames. The second NAK would still be NAK1 to inform the sender to resend frame 1 again; this has already been done. The first NAK sent is remembered (using the nakSent variable) and is not sent again until the frame slides. A NAK is sent once for each window position and defines the first slot in the window.
The next point is about the ACKs. Notice that only two ACKs are sent here. The first one acknowledges only the first frame; the second one acknowledges three frames. In Selective Repeat, ACKs are sent when data are delivered to the network layer. If the data belonging to $n$ frames are delivered in one shot, only one ACK is sent for all of them.
Figure 11.23  Flow diagram for Example 11.8
**Figure 11.24** Design of piggybacking in Go-Back-N ARQ
High-level Data Link Control (HDLC) is a bit-oriented protocol for communication over point-to-point and multipoint links. It implements the ARQ mechanisms we discussed in this chapter.

Topics discussed in this section:
Configurations and Transfer Modes
Frames
Control Field
Figure 11.25 Normal response mode

a. Point-to-point

b. Multipoint
Figure 11.26 Asynchronous balanced mode
Figure 11.27  HDLC frames
Figure 11.28  Control field format for the different frame types
<table>
<thead>
<tr>
<th>Code</th>
<th>Command</th>
<th>Response</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 001</td>
<td>SNRM</td>
<td></td>
<td>Set normal response mode</td>
</tr>
<tr>
<td>11 011</td>
<td>SNRME</td>
<td></td>
<td>Set normal response mode, extended</td>
</tr>
<tr>
<td>11 100</td>
<td>SABM</td>
<td>DM</td>
<td>Set asynchronous balanced mode or disconnect mode</td>
</tr>
<tr>
<td>11 110</td>
<td>SABME</td>
<td></td>
<td>Set asynchronous balanced mode, extended</td>
</tr>
<tr>
<td>00 000</td>
<td>UI</td>
<td>UI</td>
<td>Unnumbered information</td>
</tr>
<tr>
<td>00 110</td>
<td>UA</td>
<td></td>
<td>Unnumbered acknowledgment</td>
</tr>
<tr>
<td>00 010</td>
<td>DISC</td>
<td>RD</td>
<td>Disconnect or request disconnect</td>
</tr>
<tr>
<td>10 000</td>
<td>SIM</td>
<td>RIM</td>
<td>Set initialization mode or request information mode</td>
</tr>
<tr>
<td>00 100</td>
<td>UP</td>
<td></td>
<td>Unnumbered poll</td>
</tr>
<tr>
<td>11 001</td>
<td>RSET</td>
<td></td>
<td>Reset</td>
</tr>
<tr>
<td>11 101</td>
<td>XID</td>
<td>XID</td>
<td>Exchange ID</td>
</tr>
<tr>
<td>10 001</td>
<td>FRMR</td>
<td>FRMR</td>
<td>Frame reject</td>
</tr>
</tbody>
</table>
Figure 11.29 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 11.29  Example of connection and disconnection
Figure 11.30 shows an exchange using piggybacking. Node A begins the exchange of information with an I-frame numbered 0 followed by another I-frame numbered 1. Node B piggybacks its acknowledgment of both frames onto an I-frame of its own. Node B’s first I-frame is also numbered 0 [N(S) field] and contains a 2 in its N(R) field, acknowledging the receipt of A’s frames 1 and 0 and indicating that it expects frame 2 to arrive next. Node B transmits its second and third I-frames (numbered 1 and 2) before accepting further frames from node A.
Its N(R) information, therefore, has not changed: B frames 1 and 2 indicate that node B is still expecting A’s frame 2 to arrive next. Node A has sent all its data. Therefore, it cannot piggyback an acknowledgment onto an I-frame and sends an S-frame instead. The RR code indicates that A is still ready to receive. The number 3 in the N(R) field tells B that frames 0, 1, and 2 have all been accepted and that A is now expecting frame number 3.
Figure 11.30 Example of piggybacking without error
Figure 11.31 shows an exchange in which a frame is lost. Node B sends three data frames (0, 1, and 2), but frame 1 is lost. When node A receives frame 2, it discards it and sends a REJ frame for frame 1. Note that the protocol being used is Go-Back-N with the special use of an REJ frame as a NAK frame. The NAK frame does two things here: It confirms the receipt of frame 0 and declares that frame 1 and any following frames must be resent. Node B, after receiving the REJ frame, resends frames 1 and 2. Node A acknowledges the receipt by sending an RR frame (ACK) with acknowledgment number 3.
Figure 11.31  Example of piggybacking with error
Although HDLC is a general protocol that can be used for both point-to-point and multipoint configurations, one of the most common protocols for point-to-point access is the **Point-to-Point Protocol (PPP)**. PPP is a byte-oriented protocol.

**Topics discussed in this section:**

Framing
Transition Phases
Multiplexing
Multilink PPP
Figure 11.32  PPP frame format
PPP is a byte-oriented protocol using byte stuffing with the escape byte 01111101.
Figure 11.33 Transition phases
Figure 11.34  *Multiplexing in PPP*

<table>
<thead>
<tr>
<th>Network layer</th>
<th>Data from different networking protocols</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OSI CP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPCP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flag Address Control Protocol FCS Flag

LCP: Link Control Protocol
AP: Authentication Protocol
NCP: Network Control Protocol

LCP: 0xC021
AP: 0xC023 and 0xC223
NCP: 0x8021 and ....
Data: 0x0021 and ....
Figure 11.35  *LCP packet encapsulated in a frame*
Table 11.2  **LCP packets**

<table>
<thead>
<tr>
<th>Code</th>
<th>Packet Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>Configure-request</td>
<td>Contains the list of proposed options and their values</td>
</tr>
<tr>
<td>0x02</td>
<td>Configure-ack</td>
<td>Accepts all options proposed</td>
</tr>
<tr>
<td>0x03</td>
<td>Configure-nak</td>
<td>Announces that some options are not acceptable</td>
</tr>
<tr>
<td>0x04</td>
<td>Configure-reject</td>
<td>Announces that some options are not recognized</td>
</tr>
<tr>
<td>0x05</td>
<td>Terminate-request</td>
<td>Request to shut down the line</td>
</tr>
<tr>
<td>0x06</td>
<td>Terminate-ack</td>
<td>Accept the shutdown request</td>
</tr>
<tr>
<td>0x07</td>
<td>Code-reject</td>
<td>Announces an unknown code</td>
</tr>
<tr>
<td>0x08</td>
<td>Protocol-reject</td>
<td>Announces an unknown protocol</td>
</tr>
<tr>
<td>0x09</td>
<td>Echo-request</td>
<td>A type of hello message to check if the other end is alive</td>
</tr>
<tr>
<td>0x0A</td>
<td>Echo-reply</td>
<td>The response to the echo-request message</td>
</tr>
<tr>
<td>0x0B</td>
<td>Discard-request</td>
<td>A request to discard the packet</td>
</tr>
</tbody>
</table>
### Table 11.3 Common options

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum receive unit (payload field size)</td>
<td>1500</td>
</tr>
<tr>
<td>Authentication protocol</td>
<td>None</td>
</tr>
<tr>
<td>Protocol field compression</td>
<td>Off</td>
</tr>
<tr>
<td>Address and control field compression</td>
<td>Off</td>
</tr>
</tbody>
</table>
**Figure 11.36**  *PAP packets encapsulated in a PPP frame*

<table>
<thead>
<tr>
<th>USER</th>
<th>PPP Frame</th>
<th>SYSTEM</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Authenticate-request</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code: 1</td>
<td>ID</td>
<td>Length</td>
</tr>
<tr>
<td>Code: 2</td>
<td>ID</td>
<td>Length</td>
</tr>
<tr>
<td>Code: 3</td>
<td>ID</td>
<td>Length</td>
</tr>
</tbody>
</table>

**PAP packets**

**Payload (and padding)**

\[C023_{16}\]
Figure 11.37  CHAP packets encapsulated in a PPP frame
Figure 11.38  *IPCP packet encapsulated in PPP frame*
<table>
<thead>
<tr>
<th>Code</th>
<th>IPCP Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>Configure-request</td>
</tr>
<tr>
<td>0x02</td>
<td>Configure-ack</td>
</tr>
<tr>
<td>0x03</td>
<td>Configure-nak</td>
</tr>
<tr>
<td>0x04</td>
<td>Configure-reject</td>
</tr>
<tr>
<td>0x05</td>
<td>Terminate-request</td>
</tr>
<tr>
<td>0x06</td>
<td>Terminate-ack</td>
</tr>
<tr>
<td>0x07</td>
<td>Code-reject</td>
</tr>
</tbody>
</table>
Figure 11.39  *IP datagram encapsulated in a PPP frame*
Figure 11.40 *Multilink PPP*
Let us go through the phases followed by a network layer packet as it is transmitted through a PPP connection. Figure 11.41 shows the steps. For simplicity, we assume unidirectional movement of data from the user site to the system site (such as sending an e-mail through an ISP).

The first two frames show link establishment. We have chosen two options (not shown in the figure): using PAP for authentication and suppressing the address control fields. Frames 3 and 4 are for authentication. Frames 5 and 6 establish the network layer connection using IPCP.
The next several frames show that some IP packets are encapsulated in the PPP frame. The system (receiver) may have been running several network layer protocols, but it knows that the incoming data must be delivered to the IP protocol because the NCP protocol used before the data transfer was IPCP.

After data transfer, the user then terminates the data link connection, which is acknowledged by the system. Of course the user or the system could have chosen to terminate the network layer IPCP and keep the data link layer running if it wanted to run another NCP protocol.
Figure 11.41  An example
Figure 11.41 An example (continued)
Chapter 12

Multiple Access
Figure 12.1 Data link layer divided into two functionality-oriented sublayers

Data link layer

- Data link control
- Multiple-access resolution
Figure 12.2 Taxonomy of multiple-access protocols discussed in this chapter

- Random access protocols
  - ALOHA
  - CSMA
  - CSMA/CD
  - CSMA/CA

- Controlled-access protocols
  - Reservation
  - Polling
  - Token passing

- Channelization protocols
  - FDMA
  - TDMA
  - CDMA
In random access or contention methods, no station is superior to another station and none is assigned the control over another. No station permits, or does not permit, another station to send. At each instance, a station that has data to send uses a procedure defined by the protocol to make a decision on whether or not to send.

**Topics discussed in this section:**

ALOHA  
Carrier Sense Multiple Access  
Carrier Sense Multiple Access with Collision Detection  
Carrier Sense Multiple Access with Collision Avoidance
Figure 12.3  *Frames in a pure ALOHA network*
Figure 12.4  Procedure for pure ALOHA protocol

- **K**: Number of attempts
- **T_p**: Maximum propagation time
- **T_{fr}**: Average transmission time for a frame
- **T_B**: Back-off time

Start

Station has a frame to send

**K = 0**

Wait **T_B** time

\( T_B = R \times T_p \) or \( R \times T_{fr} \)

Choose a random number R between 0 and \( 2^K - 1 \)

**K > K_{max}**

- No: **K = K + 1**
- Yes: Abort

**ACK received?**

- No: **K = K + 1**
- Yes: Success

Wait **time-out time**

\( 2 \times T_p \)
The stations on a wireless ALOHA network are a maximum of 600 km apart. If we assume that signals propagate at $3 \times 10^8$ m/s, we find

$$T_p = \frac{(600 \times 10^5)}{(3 \times 10^8)} = 2 \text{ ms}.$$ 

Now we can find the value of $T_B$ for different values of $K$.

a. For $K = 1$, the range is {0, 1}. The station needs to generate a random number with a value of 0 or 1. This means that $T_B$ is either 0 ms (0 \times 2) or 2 ms (1 \times 2), based on the outcome of the random variable.
b. For $K = 2$, the range is $\{0, 1, 2, 3\}$. This means that $T_B$ can be 0, 2, 4, or 6 ms, based on the outcome of the random variable.

c. For $K = 3$, the range is $\{0, 1, 2, 3, 4, 5, 6, 7\}$. This means that $T_B$ can be 0, 2, 4, . . . , 14 ms, based on the outcome of the random variable.

d. We need to mention that if $K > 10$, it is normally set to 10.
Figure 12.5  Vulnerable time for pure ALOHA protocol

Vulnerable time = 2 × T_{fr}
A pure ALOHA network transmits 200-bit frames on a shared channel of 200 kbps. What is the requirement to make this frame collision-free?

Solution

Average frame transmission time $T_{fr}$ is $\frac{200 \text{ bits}}{200 \text{ kbps}}$ or 1 ms. The vulnerable time is $2 \times 1 \text{ ms} = 2 \text{ ms}$. This means no station should send later than 1 ms before this station starts transmission and no station should start sending during the one 1-ms period that this station is sending.
The throughput for pure ALOHA is
\[ S = G \times e^{-2G}. \]
The maximum throughput \( S_{\text{max}} = 0.184 \) when \( G = (1/2) \).
A pure ALOHA network transmits 200-bit frames on a shared channel of 200 kbps. What is the throughput if the system (all stations together) produces

a. 1000 frames per second  b. 500 frames per second  
c. 250 frames per second.

Solution

The frame transmission time is 200/200 kbps or 1 ms.

a. If the system creates 1000 frames per second, this is 1 frame per millisecond. The load is 1. In this case $S = G \times e^{-2G}$ or $S = 0.135$ (13.5 percent). This means that the throughput is $1000 \times 0.135 = 135$ frames. Only 135 frames out of 1000 will probably survive.
b. If the system creates 500 frames per second, this is (1/2) frame per millisecond. The load is (1/2). In this case \( S = G \times e^{-2G} \) or \( S = 0.184 \) (18.4 percent). This means that the throughput is \( 500 \times 0.184 = 92 \) and that only 92 frames out of 500 will probably survive. Note that this is the maximum throughput case, percentagewise.

c. If the system creates 250 frames per second, this is (1/4) frame per millisecond. The load is (1/4). In this case \( S = G \times e^{-2G} \) or \( S = 0.152 \) (15.2 percent). This means that the throughput is \( 250 \times 0.152 = 38 \). Only 38 frames out of 250 will probably survive.
Figure 12.6 *Frames in a slotted ALOHA network*
The throughput for slotted ALOHA is
\[ S = G \times e^{-G} . \]
The maximum throughput
\[ S_{\text{max}} = 0.368 \text{ when } G = 1. \]
Figure 12.7 Vulnerable time for slotted ALOHA protocol

A collides with C

Begin B End

Begin A End

Begin C End

Vulnerable time = $T_{fr}$
A slotted ALOHA network transmits 200-bit frames on a shared channel of 200 kbps. What is the throughput if the system (all stations together) produces
a. 1000 frames per second  b. 500 frames per second  c. 250 frames per second.

Solution
The frame transmission time is 200/200 kbps or 1 ms.

a. If the system creates 1000 frames per second, this is 1 frame per millisecond. The load is 1. In this case $S = G \times e^{-G}$ or $S = 0.368$ (36.8 percent). This means that the throughput is $1000 \times 0.0368 = 368$ frames. Only 386 frames out of 1000 will probably survive.
Example 12.4 (continued)

b. If the system creates 500 frames per second, this is (1/2) frame per millisecond. The load is (1/2). In this case \( S = G \times e^{-G} \) or \( S = 0.303 \) (30.3 percent). This means that the throughput is \( 500 \times 0.0303 = 151 \). Only 151 frames out of 500 will probably survive.

c. If the system creates 250 frames per second, this is (1/4) frame per millisecond. The load is (1/4). In this case \( S = G \times e^{-G} \) or \( S = 0.195 \) (19.5 percent). This means that the throughput is \( 250 \times 0.195 = 49 \). Only 49 frames out of 250 will probably survive.
Figure 12.8  *Space/time model of the collision in CSMA*
Figure 12.9 *Vulnerable time in CSMA*
Figure 12.10  Behavior of three persistence methods

a. 1-persistent

b. Nonpersistent

c. p-persistent
Figure 12.11 *Flow diagram for three persistence methods*

a. 1-persistent

- **Channel?**
  - **Busy**
  - **Idle**

- **Station can transmit.**

b. Nonpersistent

- **Channel?**
  - **Busy**
  - **Idle**

- **Wait randomly**

- **Station can transmit.**

c. p-persistent

- **Channel?**
  - **Busy**
  - **Idle**

- **Wait a slot**
  - >P

- **Probability outcome?**
  - ≤P

- **Use back-off process as though collision occurred.**

- **Station can transmit.**
Figure 12.12  Collision of the first bit in CSMA/CD
Figure 12.13 Collision and abortion in CSMA/CD
A network using CSMA/CD has a bandwidth of 10 Mbps. If the maximum propagation time (including the delays in the devices and ignoring the time needed to send a jamming signal, as we see later) is 25.6 μs, what is the minimum size of the frame?

Solution

The frame transmission time is \( T_{fr} = 2 \times T_p = 51.2 \) μs. This means, in the worst case, a station needs to transmit for a period of 51.2 μs to detect the collision. The minimum size of the frame is 10 Mbps × 51.2 μs = 512 bits or 64 bytes. This is actually the minimum size of the frame for Standard Ethernet.
Figure 12.14  *Flow diagram for the CSMA/CD*

- **K**: Number of attempts
- **T_p**: Maximum propagation time
- **T_f**: Average transmission time for a frame
- **T_B**: Back-off time

**Start**

1. **K = 0**
2. Apply one of the persistence methods (1-persistent, nonpersistent, or p-persistent)
3. **Eligible for transmission**
   - **(Transmission done) or (Collision detected)**
     - **No**
       - **Transmit and receive**
     - **Yes**
       - **Collision detected?**
         - **Yes**
           - Send a jamming signal
         - **No**

4. **Choose a random number R between 0 and 2^K - 1**
5. **Wait T_B time (T_B = R x T_p or R x T_f)**
6. **K > K_{max}**
   - **No**
     - Abort
   - **Yes**
     - **K = K + 1**
Figure 12.15  *Energy level during transmission, idleness, or collision*
Figure 12.16  Timing in CSMA/CA
In CSMA/CA, the IFS can also be used to define the priority of a station or a frame.
In CSMA/CA, if the station finds the channel busy, it does not restart the timer of the contention window; it stops the timer and restarts it when the channel becomes idle.
Figure 12.17 Flow diagram for CSMA/CA
In **controlled access**, the stations consult one another to find which station has the right to send. A station cannot send unless it has been authorized by other stations. **We discuss three popular controlled-access methods.**

**Topics discussed in this section:**
- Reservation
- Polling
- Token Passing
Figure 12.18  Reservation access method
Figure 12.19 Select and poll functions in polling access method
Figure 12.20  Logical ring and physical topology in token-passing access method

a. Physical ring

b. Dual ring

c. Bus ring

d. Star ring
Channelization is a multiple-access method in which the available bandwidth of a link is shared in time, frequency, or through code, between different stations. In this section, we discuss three channelization protocols.

**Topics discussed in this section:**
- Frequency-Division Multiple Access (FDMA)
- Time-Division Multiple Access (TDMA)
- Code-Division Multiple Access (CDMA)
We see the application of all these methods in Chapter 16 when we discuss cellular phone systems.
Figure 12.21  *Frequency-division multiple access (FDMA)*
In FDMA, the available bandwidth of the common channel is divided into bands that are separated by guard bands.
Figure 12.22  *Time-division multiple access (TDMA)*
In TDMA, the bandwidth is just one channel that is timeshared between different stations.
In CDMA, one channel carries all transmissions simultaneously.
Figure 12.23  Simple idea of communication with code

\[ d_1 \cdot c_1 + d_2 \cdot c_2 + d_3 \cdot c_3 + d_4 \cdot c_4 \]
Figure 12.24 Chip sequences

C_1: [+1  1  1  1  1]  
C_2: [+1 -1  1 -1]  
C_3: [+1  1 -1 -1]  
C_4: [+1 -1 -1  1]
Figure 12.25  *Data representation in CDMA*
Figure 12.26  Sharing channel in CDMA
Figure 12.27  Digital signal created by four stations in CDMA

Bit 0

1

[-1 -1 -1 -1]

Time

Bit 0

2

[-1 +1 -1 +1]

Time

Silent

3

[0 0 0 0]

Data on the channel

Time

Bit 1

4

[+1 -1 -1 +1]

Time

Data on the channel
Figure 12.28  Decoding of the composite signal for one in CDMA
Figure 12.29  *General rule and examples of creating Walsh tables*

\[
\begin{align*}
W_1 &= \begin{bmatrix} +1 \end{bmatrix} \\
W_2 &= \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \\
W_{2N} &= \begin{bmatrix} W_N & W_N \\ W_N & \overline{W_N} \end{bmatrix}
\end{align*}
\]

a. Two basic rules

\[
\begin{align*}
W_1 &= \begin{bmatrix} +1 \end{bmatrix} \\
W_2 &= \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \\
W_4 &= \begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1
\end{bmatrix}
\end{align*}
\]

b. Generation of \(W_1, W_2, \) and \(W_4\)
Note

The number of sequences in a Walsh table needs to be \( N = 2^m \).
Example 12.6

Find the chips for a network with

a. Two stations  

b. Four stations

Solution

We can use the rows of $W_2$ and $W_4$ in Figure 12.29:

a. For a two-station network, we have

$[+1 +1]$ and $[+1 -1]$.

b. For a four-station network we have

$[+1 +1 +1 +1]$, $[+1 -1 +1 -1]$, $[+1 +1 -1 -1]$, and $[+1 -1 -1 +1]$. 
What is the number of sequences if we have 90 stations in our network?

Solution

The number of sequences needs to be $2^m$. We need to choose $m = 7$ and $N = 2^7$ or 128. We can then use 90 of the sequences as the chips.
Example 12.8

Prove that a receiving station can get the data sent by a specific sender if it multiplies the entire data on the channel by the sender’s chip code and then divides it by the number of stations.

Solution

Let us prove this for the first station, using our previous four-station example. We can say that the data on the channel

\[ D = (d_1 \cdot c_1 + d_2 \cdot c_2 + d_3 \cdot c_3 + d_4 \cdot c_4). \]

The receiver which wants to get the data sent by station 1 multiplies these data by \( c_1 \).
Example 12.8 (continued)

When we divide the result by $N$, we get $d_1$. 

\[
D \cdot c_1 = (d_1 \cdot c_1 + d_2 \cdot c_2 + d_3 \cdot c_3 + d_4 \cdot c_4) \cdot c_1 \\
= d_1 \cdot c_1 \cdot c_1 + d_2 \cdot c_2 \cdot c_1 + d_3 \cdot c_3 \cdot c_1 + d_4 \cdot c_4 \cdot c_1 \\
= d_1 \times N + d_2 \times 0 + d_3 \times 0 + d_4 \times 0 \\
= d_1 \times N
\]
Chapter 13

Wired LANs: Ethernet
In 1985, the Computer Society of the IEEE started a project, called Project 802, to set standards to enable intercommunication among equipment from a variety of manufacturers. Project 802 is a way of specifying functions of the physical layer and the data link layer of major LAN protocols.

**Topics discussed in this section:**

- Data Link Layer
- Physical Layer
Figure 13.1  **IEEE standard for LANs**
Figure 13.2  HDLC frame compared with LLC and MAC frames

DSAP: Destination service access point
SSAP: Source service access point
The original Ethernet was created in 1976 at Xerox’s Palo Alto Research Center (PARC). Since then, it has gone through four generations. We briefly discuss the Standard (or traditional) Ethernet in this section.

Topics discussed in this section:
- MAC Sublayer
- Physical Layer
Figure 13.3 *Ethernet evolution through four generations*

- **Standard Ethernet**: 10 Mbps
- **Fast Ethernet**: 100 Mbps
- **Gigabit Ethernet**: 1 Gbps
- **Ten-Gigabit Ethernet**: 10 Gbps
**Figure 13.4 802.3 MAC frame**

**Preamble**: 56 bits of alternating 1s and 0s.

**SFD**: Start frame delimiter, flag (10101011)

<table>
<thead>
<tr>
<th>Preamble</th>
<th>SFD</th>
<th>Destination address</th>
<th>Source address</th>
<th>Length or type</th>
<th>Data and padding</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 bytes</td>
<td>1 byte</td>
<td>6 bytes</td>
<td>6 bytes</td>
<td>2 bytes</td>
<td></td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

Physical layer header
Figure 13.5 *Minimum and maximum lengths*

<table>
<thead>
<tr>
<th>Destination address</th>
<th>Source address</th>
<th>Length PDU</th>
<th>Data and padding</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bytes</td>
<td>6 bytes</td>
<td>2 bytes</td>
<td></td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

Minimum frame length: 512 bits or 64 bytes
Maximum frame length: 12,144 bits or 1518 bytes

Minimum payload length: 46 bytes
Maximum payload length: 1500 bytes
Frame length:
Minimum: 64 bytes (512 bits)
Maximum: 1518 bytes (12,144 bits)
Figure 13.6 Example of an Ethernet address in hexadecimal notation

06:01:02:01:2C:4B

6 bytes = 12 hex digits = 48 bits
Figure 13.7  Unicast and multicast addresses
The least significant bit of the first byte defines the type of address. If the bit is 0, the address is unicast; otherwise, it is multicast.
The broadcast destination address is a special case of the multicast address in which all bits are 1s.
Define the type of the following destination addresses:

a. 4A:30:10:21:10:1A  
b. 47:20:1B:2E:08:EE  

Solution

To find the type of the address, we need to look at the second hexadecimal digit from the left. If it is even, the address is unicast. If it is odd, the address is multicast. If all digits are F’s, the address is broadcast. Therefore, we have the following:

a. This is a unicast address because A in binary is 1010.

b. This is a multicast address because 7 in binary is 0111.

c. This is a broadcast address because all digits are F’s.
Show how the address 47:20:1B:2E:08:EE is sent out on line.

Solution
The address is sent left-to-right, byte by byte; for each byte, it is sent right-to-left, bit by bit, as shown below:
Figure 13.8 Categories of Standard Ethernet

- **10Base5**: Bus, thick coaxial
- **10Base2**: Bus, thin coaxial
- **10Base-T**: Star, UTP
- **10Base-F**: Star, fiber
Figure 13.9 Encoding in a Standard Ethernet implementation

Diagram showing a network setup with a station connected to twisted pairs or fibers via a Manchester encoder and decoder, transmitting 10 Mbps data.
Figure 13.10 10Base5 implementation
**Figure 13.11** 10Base2 implementation

- **10 Base2**
  - 10 Mbps
  - 185 m
- Baseband (digital)
- Cable end
- Thin coaxial cable, maximum 185 m
- Cable end
Figure 13.12  10Base-T implementation
Figure 13.13 10Base-F implementation
Table 13.1  **Summary of Standard Ethernet implementations**

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>10Base5</th>
<th>10Base2</th>
<th>10Base-T</th>
<th>10Base-F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Media</td>
<td>Thick coaxial</td>
<td>Thin coaxial</td>
<td>2 UTP</td>
<td>2 Fiber</td>
</tr>
<tr>
<td></td>
<td>cable</td>
<td>cable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum length</td>
<td>500 m</td>
<td>185 m</td>
<td>100 m</td>
<td>2000 m</td>
</tr>
<tr>
<td>Line encoding</td>
<td>Manchester</td>
<td>Manchester</td>
<td>Manchester</td>
<td>Manchester</td>
</tr>
</tbody>
</table>
The 10-Mbps Standard Ethernet has gone through several changes before moving to the higher data rates. These changes actually opened the road to the evolution of the Ethernet to become compatible with other high-data-rate LANs.

**Topics discussed in this section:**
- Bridged Ethernet
- Switched Ethernet
- Full-Duplex Ethernet
Figure 13.14  *Sharing bandwidth*

a. First station

b. Second station
**Figure 13.15** A network with and without a bridge

a. Without bridging

b. With bridging
**Figure 13.16** Collision domains in an unbridged network and a bridged network.
Figure 13.17  *Switched Ethernet*
Figure 13.18  *Full-duplex switched Ethernet*
Fast Ethernet was designed to compete with LAN protocols such as FDDI or Fiber Channel. IEEE created Fast Ethernet under the name 802.3u. Fast Ethernet is backward-compatible with Standard Ethernet, but it can transmit data 10 times faster at a rate of 100 Mbps.

Topics discussed in this section:
MAC Sublayer
Physical Layer
Figure 13.19  Fast Ethernet topology

a. Point-to-point

b. Star
Figure 13.20  *Fast Ethernet implementations*

- **100Base-TX**: Two wires, category 5 UTP
- **100Base-FX**: Two wires, fiber
- **100Base-T4**: Four wires, category 3 UTP
Figure 13.21 Encoding for Fast Ethernet implementation
<table>
<thead>
<tr>
<th>Characteristics</th>
<th>100Base-TX</th>
<th>100Base-FX</th>
<th>100Base-T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Media</td>
<td>Cat 5 UTP or STP</td>
<td>Fiber</td>
<td>Cat 4 UTP</td>
</tr>
<tr>
<td>Number of wires</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Maximum length</td>
<td>100 m</td>
<td>100 m</td>
<td>100 m</td>
</tr>
<tr>
<td>Block encoding</td>
<td>4B/5B</td>
<td>4B/5B</td>
<td></td>
</tr>
<tr>
<td>Line encoding</td>
<td>MLT-3</td>
<td>NRZ-I</td>
<td>8B/6T</td>
</tr>
</tbody>
</table>

Table 13.2  Summary of Fast Ethernet implementations
The need for an even higher data rate resulted in the design of the Gigabit Ethernet protocol (1000 Mbps). The IEEE committee calls the standard 802.3z.

*Topics discussed in this section:*
- MAC Sublayer
- Physical Layer
- Ten-Gigabit Ethernet
In the full-duplex mode of Gigabit Ethernet, there is no collision; the maximum length of the cable is determined by the signal attenuation in the cable.
Figure 13.22 Topologies of Gigabit Ethernet

- a. Point-to-point
- b. Star
- c. Two stars
- d. Hierarchy of stars
Figure 13.23  Gigabit Ethernet implementations

- **1000Base-SX**: Two-wire short-wave fiber
- **1000Base-LX**: Two-wire long-wave fiber
- **1000Base-CX**: Two-wire copper (STP)
- **1000Base-T**: Four-wire UTP
Figure 13.24  Encoding in Gigabit Ethernet implementations

1000Base-SX, 1000Base-LX, and 1000Base-CX

1000Base-T

Station

Two fibers or two STPs

4 UTP cables

8 × 125 Mbps

8 × 125 Mbps

8B/10B block encoder

NRZ line encoder

1.25 Gbps

1.25 Gbps

8B/10B block decoder

NRZ line decoder

1000Base-T

4D-PAM5 encoder

8 × 125 Mbps

4D-PAM5 decoder
Table 13.3  Summary of Gigabit Ethernet implementations

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>1000Base-SX</th>
<th>1000Base-LX</th>
<th>1000Base-CX</th>
<th>1000Base-T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Media</td>
<td>Fiber short-wave</td>
<td>Fiber long-wave</td>
<td>STP</td>
<td>Cat 5 UTP</td>
</tr>
<tr>
<td>Number of wires</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Maximum length</td>
<td>550 m</td>
<td>5000 m</td>
<td>25 m</td>
<td>100 m</td>
</tr>
<tr>
<td>Block encoding</td>
<td>8B/10B</td>
<td>8B/10B</td>
<td>8B/10B</td>
<td></td>
</tr>
<tr>
<td>Line encoding</td>
<td>NRZ</td>
<td>NRZ</td>
<td>NRZ</td>
<td>4D-PAM5</td>
</tr>
</tbody>
</table>
### Table 13.4 Summary of Ten-Gigabit Ethernet Implementations

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>10GBase-S</th>
<th>10GBase-L</th>
<th>10GBase-E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Media</td>
<td>Short-wave</td>
<td>Long-wave</td>
<td>Extended</td>
</tr>
<tr>
<td></td>
<td>850-nm multimode</td>
<td>1310-nm single mode</td>
<td>1550-mm single mode</td>
</tr>
<tr>
<td>Maximum length</td>
<td>300 m</td>
<td>10 km</td>
<td>40 km</td>
</tr>
</tbody>
</table>
Chapter 14

Wireless LANs
IEEE has defined the specifications for a wireless LAN, called IEEE 802.11, which covers the physical and data link layers.

Topics discussed in this section:
- Architecture
- MAC Sublayer
- Physical Layer
A BSS without an AP is called an ad hoc network; a BSS with an AP is called an infrastructure network.
**Figure 14.1** Basic service sets (BSSs)

BSS: Basic service set
AP: Access point

Ad hoc network (BSS without an AP) vs. Infrastructure (BSS with an AP)
Figure 14.2 *Extended service sets (ESSs)*

**ESS**: Extended service set

**BSS**: Basic service set

**AP**: Access point
Figure 14.3  MAC layers in IEEE 802.11 standard

- IEEE 802.1
  - Point coordination function (PCF)
  - Distributed coordination function (DCF)
    - 802.11 FHSS
    - 802.11 DSSS
    - 802.11 Infrared
    - 802.11a DSSS
    - 802.11a OFDM
    - 802.11g DSSS

- LLC sublayer
- MAC sublayer

Contention-free service
Contention service
Figure 14.4  CSMA/CA flowchart
Figure 14.5  CSMA/CA and NAV

Time

Source

DIFS

SIFS

RTS

CTS

Data

ACK

Time

Destination

SIFS

SIFS

Time

All other stations

...  

NAV
  (No carrier sensing)

Time

Time

14.8
**Figure 14.6** Example of repetition interval
Figure 14.7 *Frame format*

```
2 bytes 2 bytes 6 bytes 6 bytes 6 bytes 2 bytes 6 bytes 0 to 2312 bytes 4 bytes
FC    D   Address 1 Address 2 Address 3 SC Address 4 Frame body  FCS
Protocol version Type  Subtype  To DS From DS More flag Retry Pwr mgt More data WEP Rsvd
2 bits 2 bits 4 bits 1 bit 1 bit 1 bit 1 bit 1 bit 1 bit 1 bit
```
Table 14.1  **Subfields in FC field**

<table>
<thead>
<tr>
<th>Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>Current version is 0</td>
</tr>
<tr>
<td>Type</td>
<td>Type of information: management (00), control (01), or data (10)</td>
</tr>
<tr>
<td>Subtype</td>
<td>Subtype of each type (see Table 14.2)</td>
</tr>
<tr>
<td>To DS</td>
<td>Defined later</td>
</tr>
<tr>
<td>From DS</td>
<td>Defined later</td>
</tr>
<tr>
<td>More flag</td>
<td>When set to 1, means more fragments</td>
</tr>
<tr>
<td>Retry</td>
<td>When set to 1, means retransmitted frame</td>
</tr>
<tr>
<td>Pwr mgt</td>
<td>When set to 1, means station is in power management mode</td>
</tr>
<tr>
<td>More data</td>
<td>When set to 1, means station has more data to send</td>
</tr>
<tr>
<td>WEP</td>
<td>Wired equivalent privacy (encryption implemented)</td>
</tr>
<tr>
<td>Rsvd</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Figure 14.8  Control frames
**Table 14.2**  Values of subfields in control frames

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011</td>
<td>Request to send (RTS)</td>
</tr>
<tr>
<td>1100</td>
<td>Clear to send (CTS)</td>
</tr>
<tr>
<td>1101</td>
<td>Acknowledgment (ACK)</td>
</tr>
</tbody>
</table>
### Table 14.3 Addresses

<table>
<thead>
<tr>
<th>To DS</th>
<th>From DS</th>
<th>Address 1</th>
<th>Address 2</th>
<th>Address 3</th>
<th>Address 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Destination</td>
<td>Source</td>
<td>BSS ID</td>
<td>N/A</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Destination</td>
<td>Sending AP</td>
<td>Source</td>
<td>N/A</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Receiving AP</td>
<td>Source</td>
<td>Destination</td>
<td>N/A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Receiving AP</td>
<td>Sending AP</td>
<td>Destination</td>
<td>Source</td>
</tr>
</tbody>
</table>
Figure 14.9 *Addressing mechanisms*

a. Case 1

b. Case 2

c. Case 3

d. Case 4
Figure 14.10  *Hidden station problem*

B and C are hidden from each other with respect to A.
The CTS frame in CSMA/CA handshake can prevent collision from a hidden station.
Figure 14.11  Use of handshaking to prevent hidden station problem
Figure 14.12  Exposed station problem

C is exposed to transmission from A to B.
Figure 14.13  *Use of handshaking in exposed station problem*
## Table 14.4  Physical layers

<table>
<thead>
<tr>
<th>IEEE</th>
<th>Technique</th>
<th>Band</th>
<th>Modulation</th>
<th>Rate (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>802.11</td>
<td>FHSS</td>
<td>2.4 GHz</td>
<td>FSK</td>
<td>1 and 2</td>
</tr>
<tr>
<td></td>
<td>DSSS</td>
<td>2.4 GHz</td>
<td>PSK</td>
<td>1 and 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Infrared</td>
<td>PPM</td>
<td>1 and 2</td>
</tr>
<tr>
<td>802.11a</td>
<td>OFDM</td>
<td>5.725 GHz</td>
<td>PSK or QAM</td>
<td>6 to 54</td>
</tr>
<tr>
<td>802.11b</td>
<td>DSSS</td>
<td>2.4 GHz</td>
<td>PSK</td>
<td>5.5 and 11</td>
</tr>
<tr>
<td>802.11g</td>
<td>OFDM</td>
<td>2.4 GHz</td>
<td>Different</td>
<td>22 and 54</td>
</tr>
</tbody>
</table>
Figure 14.14 Industrial, scientific, and medical (ISM) band
Figure 14.15  Physical layer of IEEE 802.11 FHSS

Diagram showing the physical layer of IEEE 802.11 FHSS, with a 1 or 2 Mbps digital data input, passing through a modulator that converts it to a 2-level or 4-level FSK, and finally to a 1-MHz analog signal.
Figure 14.16  Physical layer of IEEE 802.11 DSSS
Figure 14.17  Physical layer of IEEE 802.11 infrared

1 or 2 Mbps
Digital data

Encoder
4 to 16 or 2 to 4

Modulator
Pulse position modulation

Analog signal
Figure 14.18  Physical layer of IEEE 802.11b
Bluetooth is a wireless LAN technology designed to connect devices of different functions such as telephones, notebooks, computers, cameras, printers, coffee makers, and so on. A Bluetooth LAN is an ad hoc network, which means that the network is formed spontaneously.

Topics discussed in this section:

- Architecture
- Bluetooth Layers
- Baseband Layer
- L2CAP
Figure 14.19  *Piconet*

Piconet

Primary

Figure 14.20 *Scatternet*
Figure 14.21  Bluetooth layers
Figure 14.22 *Single-secondary communication*

![Diagram showing single-secondary communication with a 625 ms cycle and a 366 ms interval.]
Figure 14.23  *Multiple-secondary communication*
**Figure 14.24** Frame format types

The diagram illustrates the structure of a frame, consisting of:

- **Access code**: 72 bits
- **Header**: 54 bits
- **Data**: 0 to N bits

The header includes:

- **Address**: 3 bits
- **Type**: 4 bits
- **F A S**: 1 1 1
- **HEC**: 8 bits

For different frame types:

- N = 240 for 1-slot frame
- N = 1490 for 3-slot frame
- N = 2740 for 5-slot frame

This 18-bit part is repeated 3 times.
Figure 14.25  L2CAP data packet format