Error Detection and Correction

Note

Data can be corrupted during transmission.

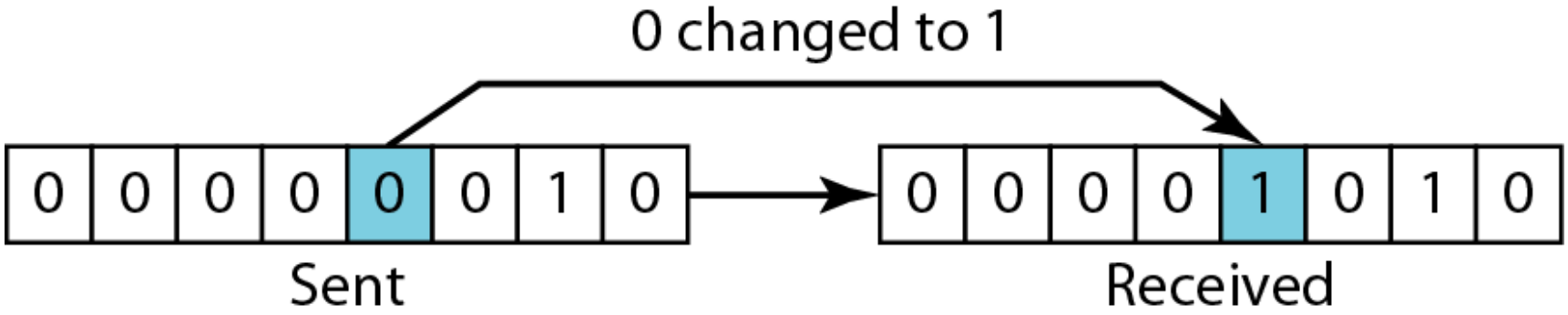
Some applications require that errors be detected and corrected.

1 INTRODUCTION

Some issues related, directly or indirectly, to error detection and correction.

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

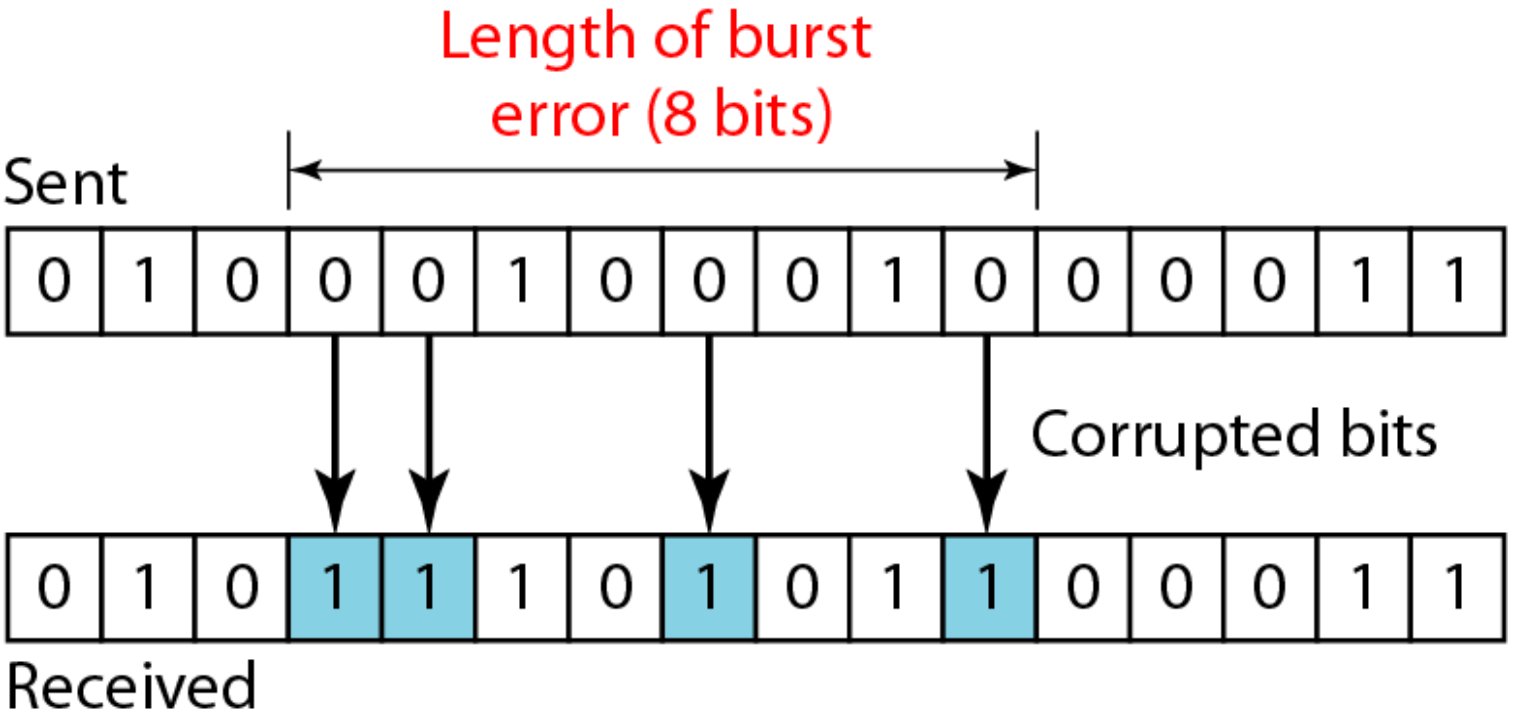
Figure 1 ***Single-bit error***



Note

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

Figure 2 *Burst error of length 8*



Note

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

Figure 3 *The structure of encoder and decoder*

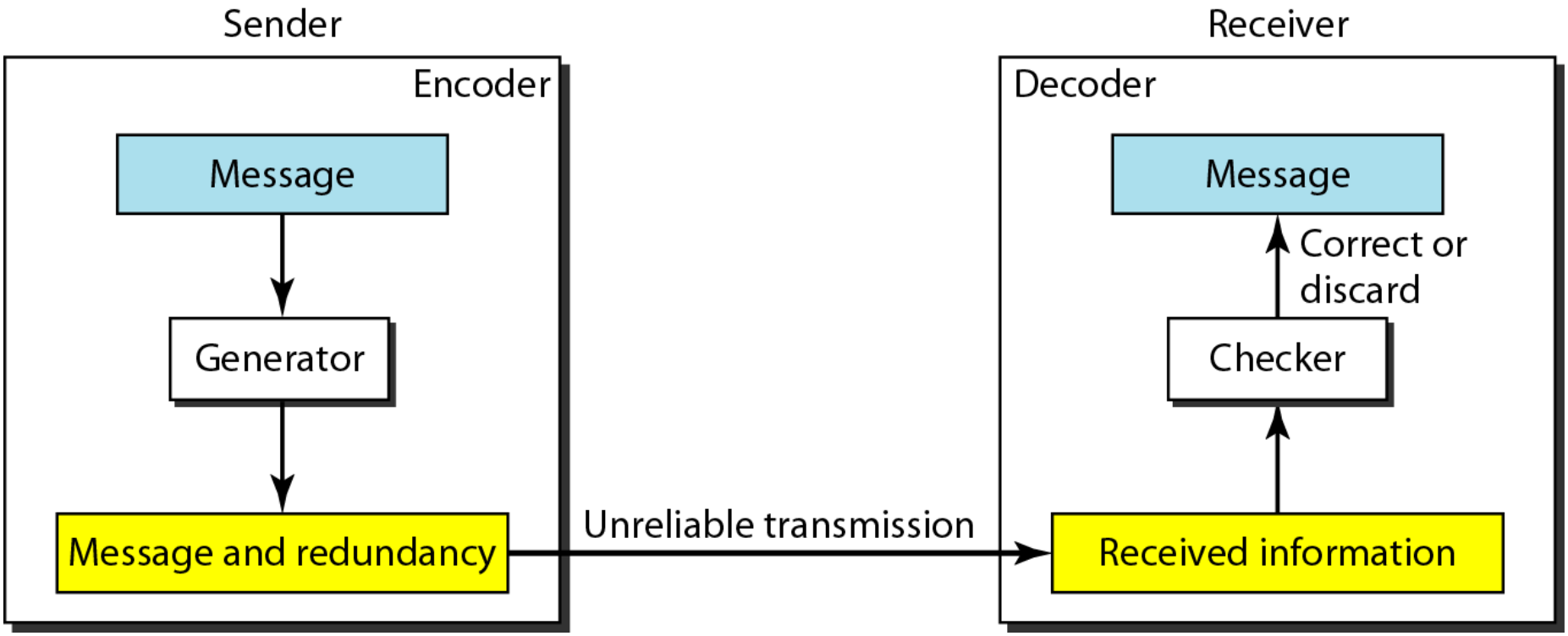
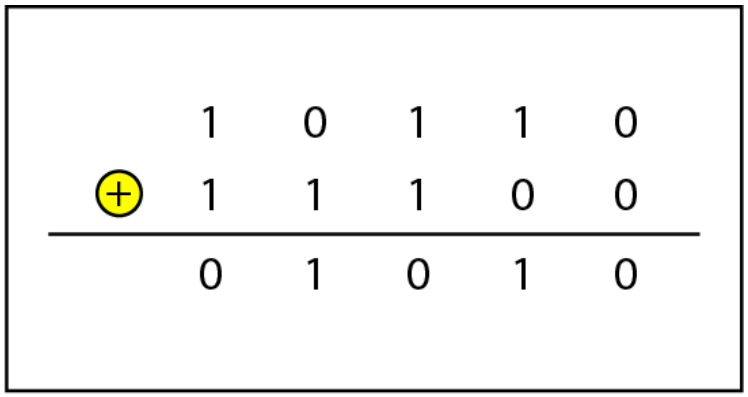


Figure 4 *XOR ing of two single bits or two words*

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

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



c. Result of XORing two patterns

2 BLOCK CODING

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.

Figure 5 *Datawords and codewords in block coding*

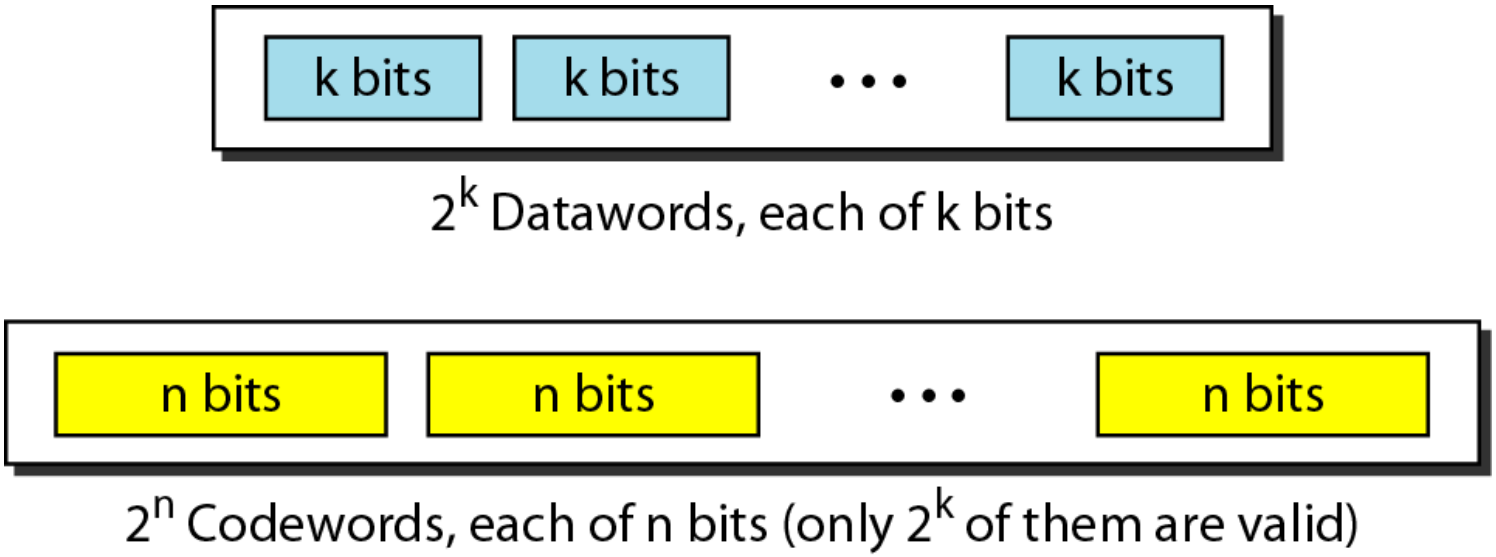


Figure 6 *Process of error detection in block coding*

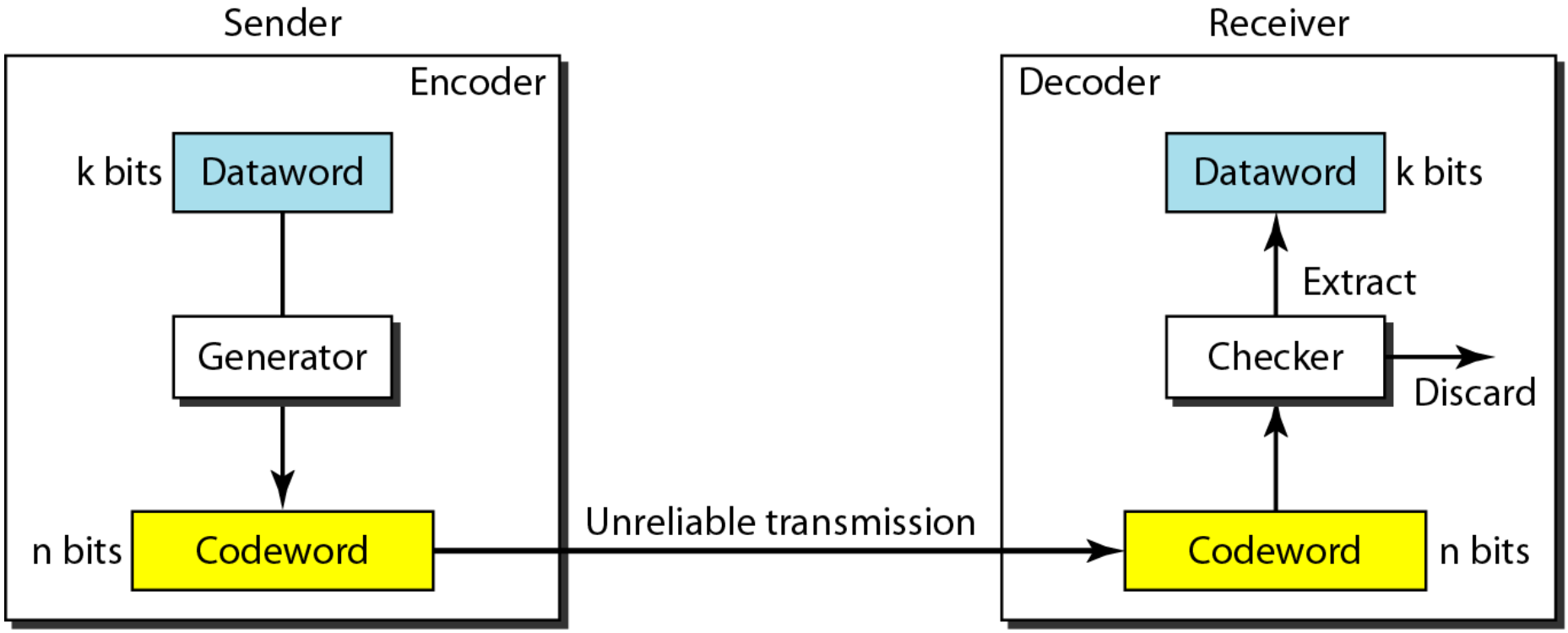


Table 1 *A code for error detection (Example 2)*

Datawords	*Codewords*
00	000
01	011
10	101
11	110

Note

An error-detecting code can detect only the types of errors for which it is designed; other types of errors may remain undetected.

Note

The Hamming distance between two words is the number of differences between corresponding bits.

Note

The minimum Hamming distance is the smallest Hamming distance between all possible pairs in a set of words.

3 LINEAR BLOCK CODES

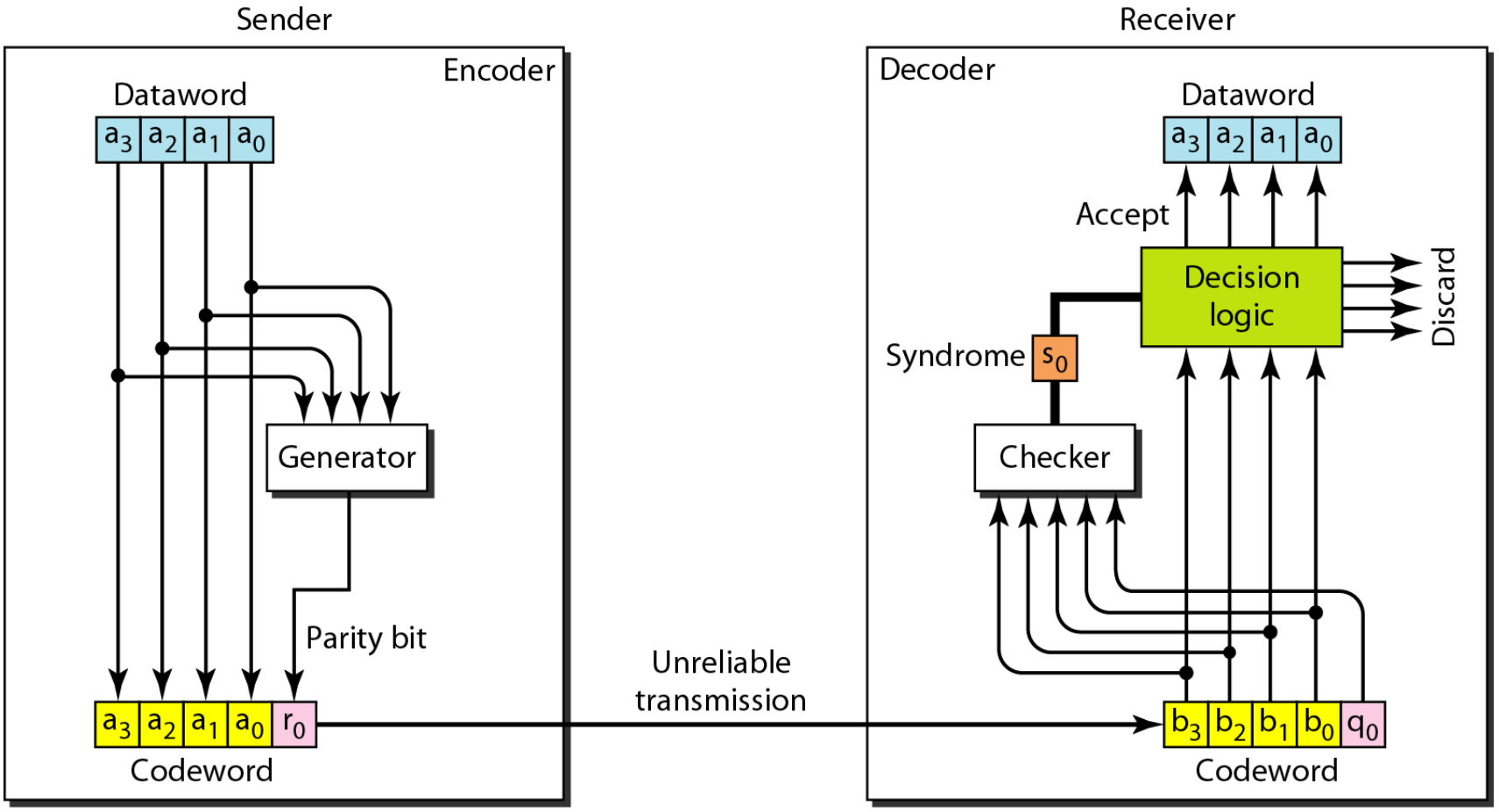
Almost all block codes used today belong to a subset called linear block codes. A linear block code is a code in which the exclusive OR (addition modulo-2) of two valid codewords creates another valid codeword.

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

Note

A simple parity-check code is a single-bit error-detecting code in which $n = k + 1$.

Figure 8 *Encoder and decoder for simple parity-check code*



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

Two-dimensional parity

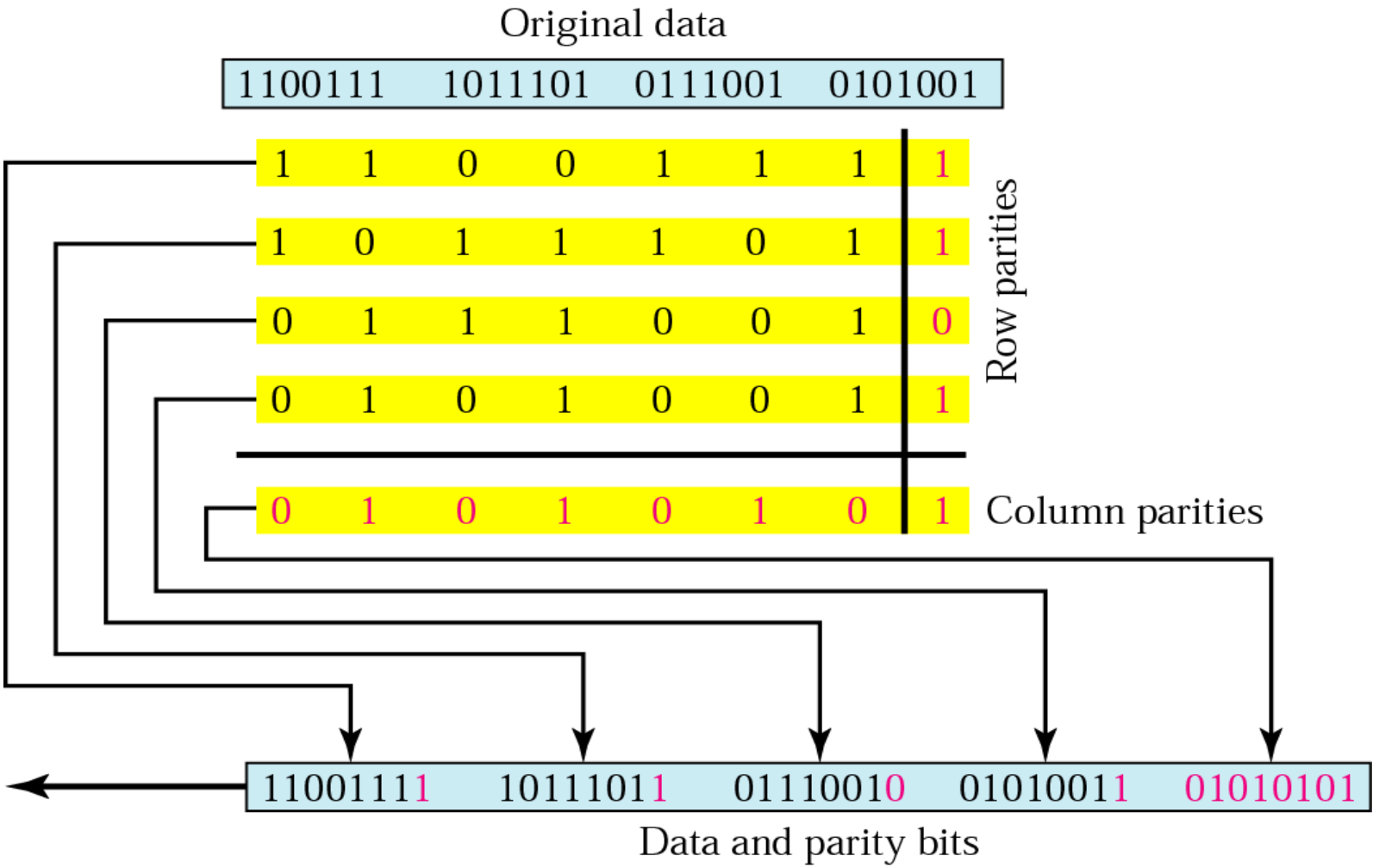
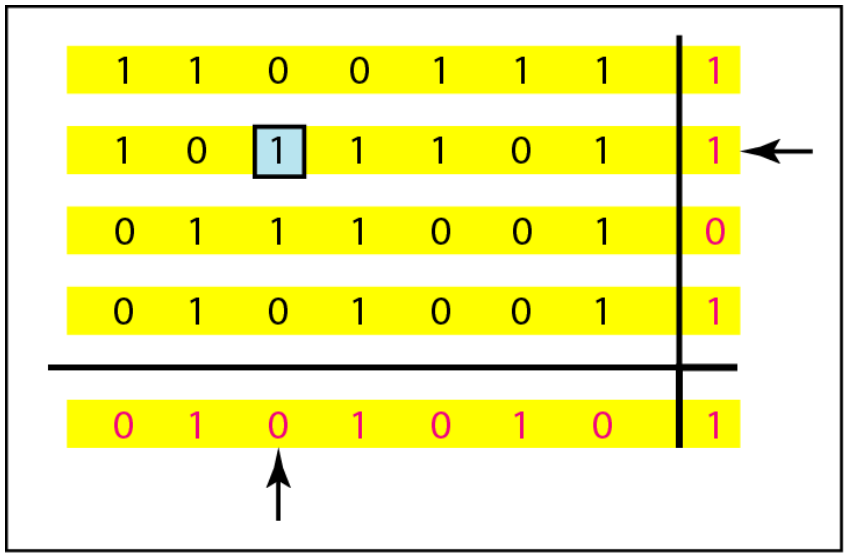
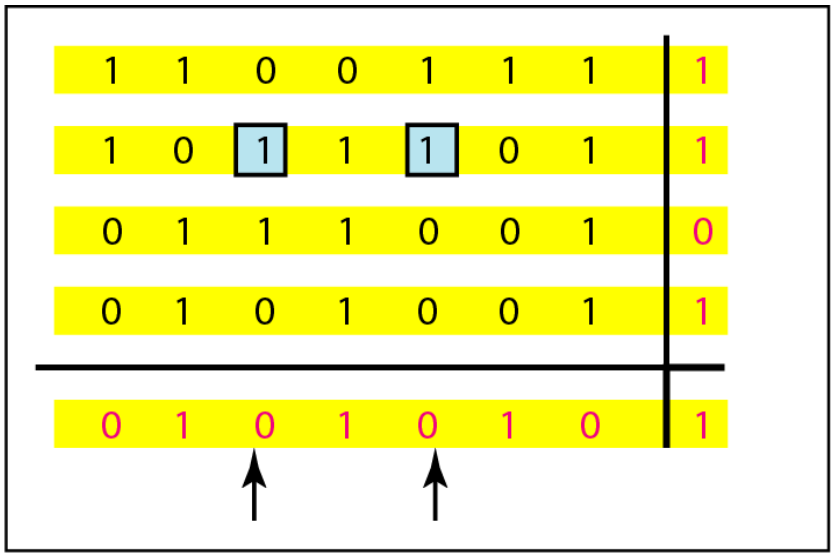


Figure 9 *Two-dimensional parity-check code*

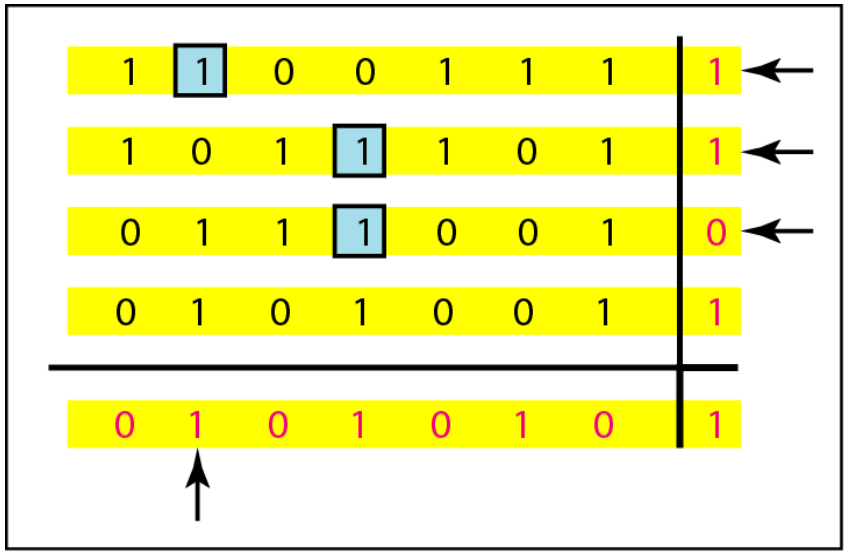


b. One error affects two parities

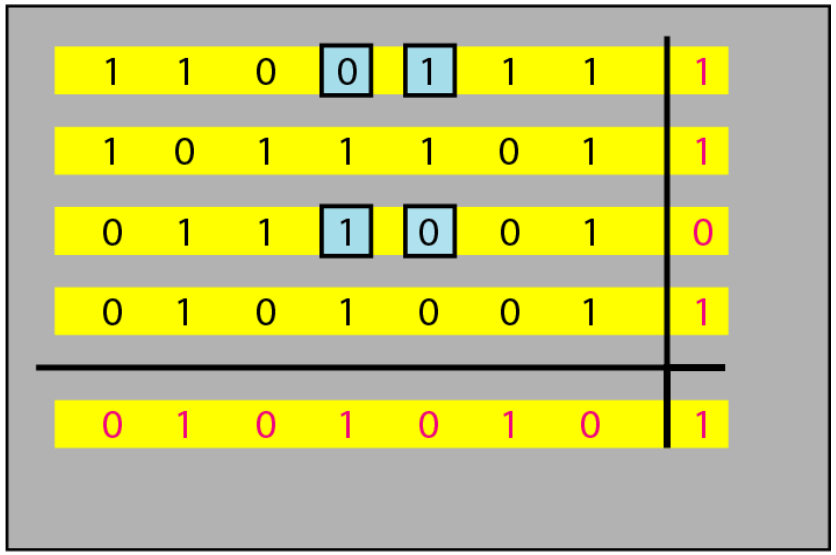


c. Two errors affect two parities

Figure 10 *Two-dimensional parity-check code*



d. Three errors affect four parities



e. Four errors cannot be detected

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

CRC generator and checker

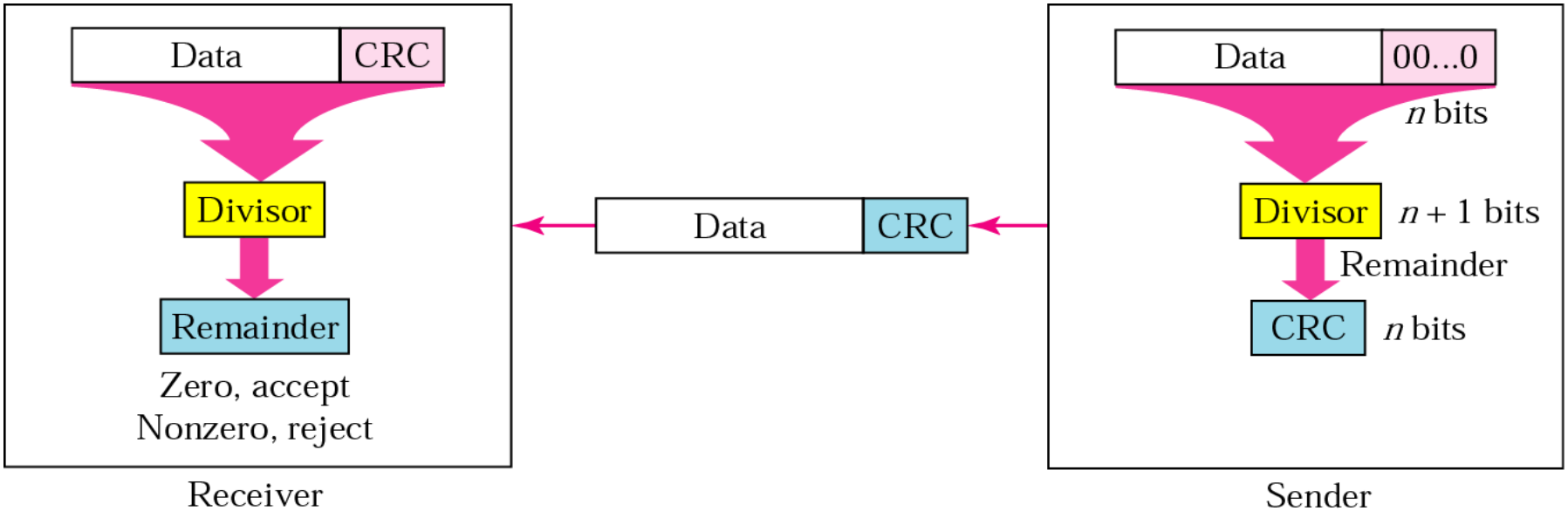
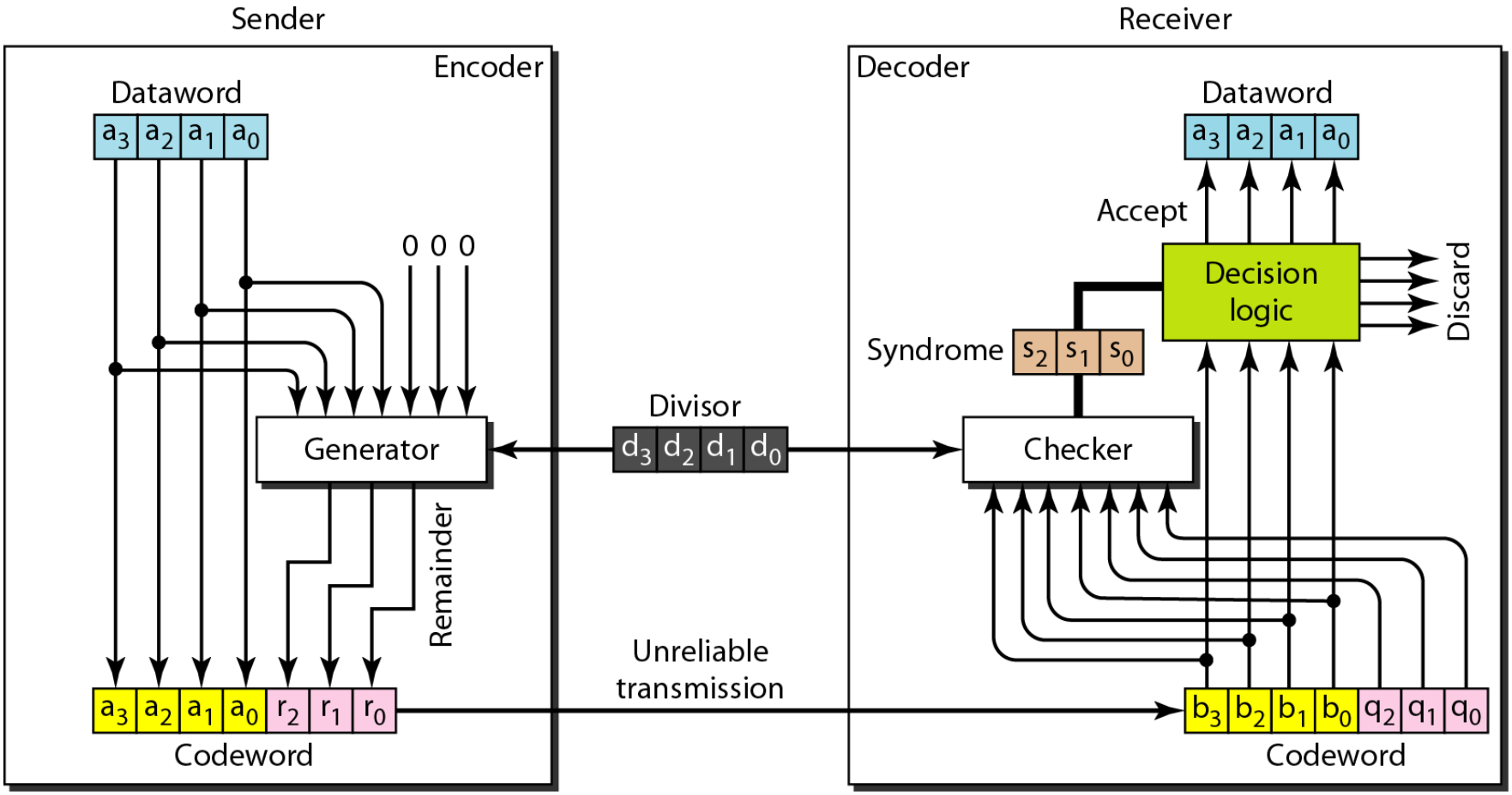


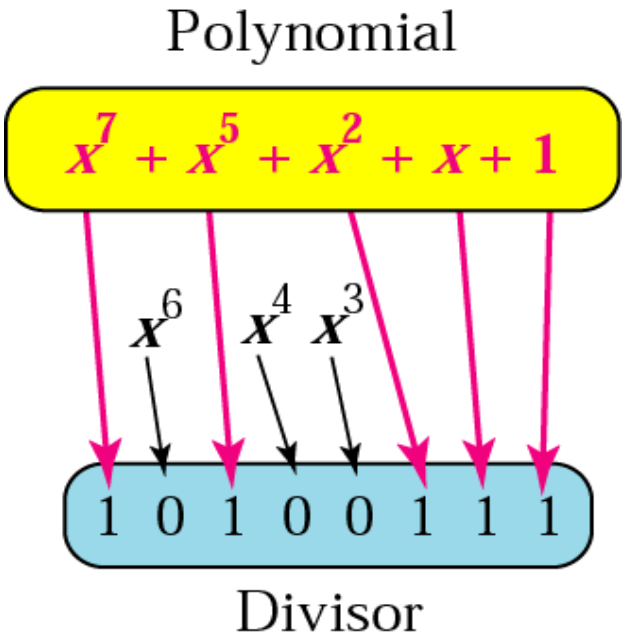
Figure 11 *CRC encoder and decoder*



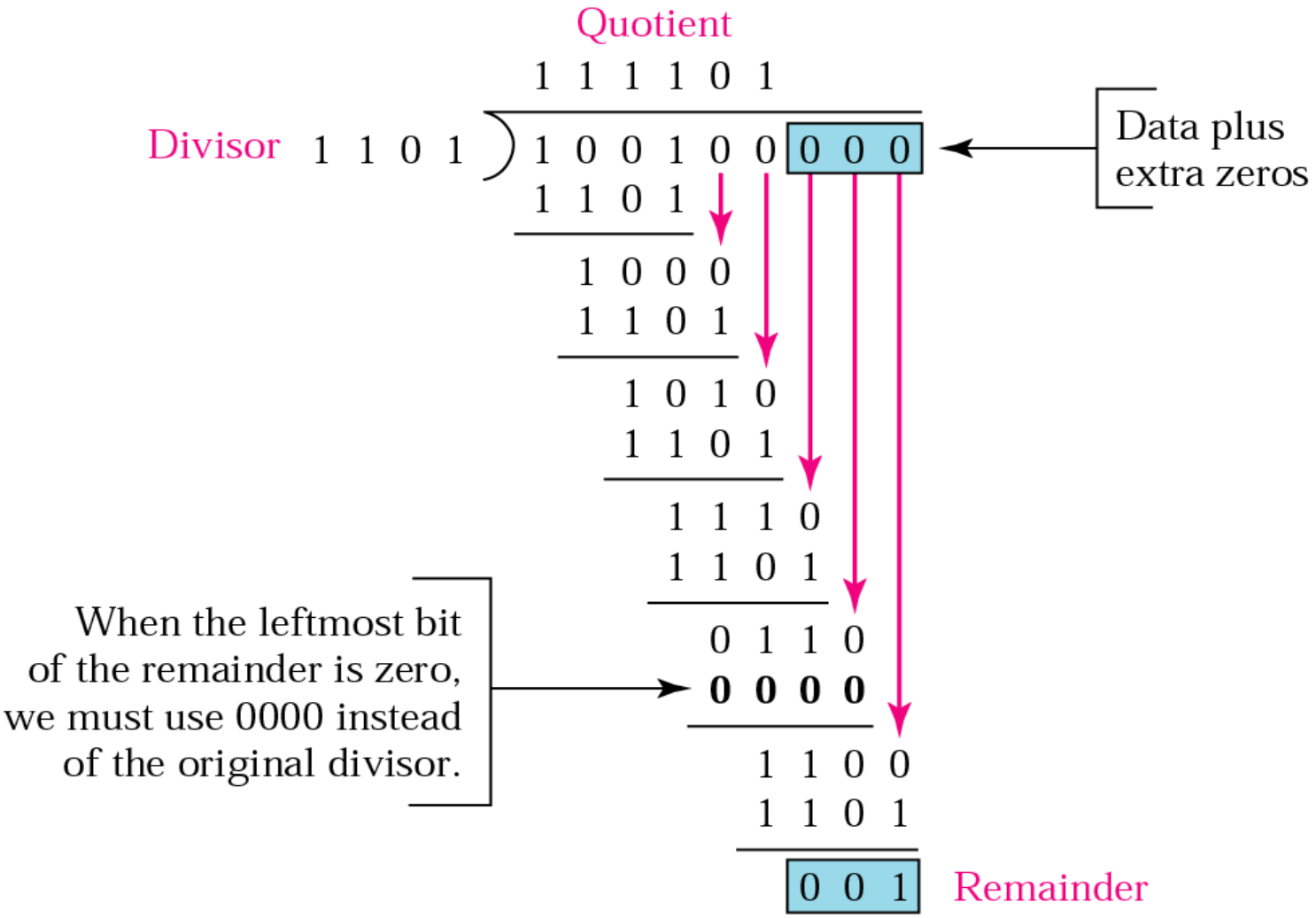
A polynomial

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

A polynomial representing a divisor



Binary division in a CRC generator



Binary division in CRC checker

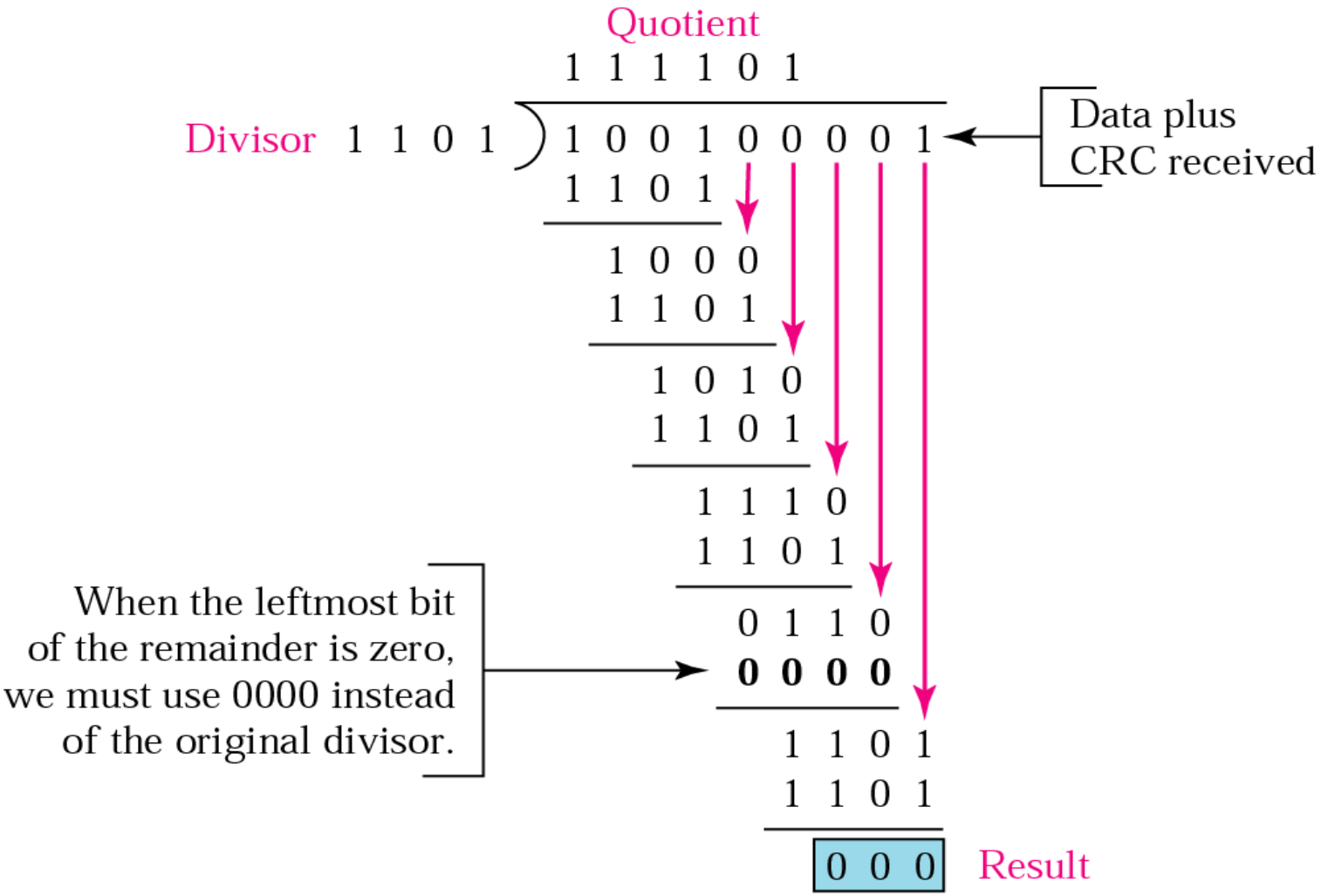


Table 4 Standard polynomials

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

Note

In a cyclic code,

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

If $s(x) = 0$, either

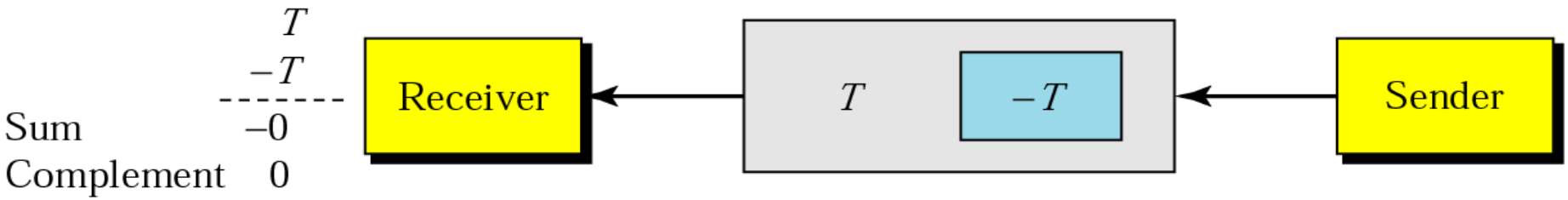
a. No bit is corrupted. or

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

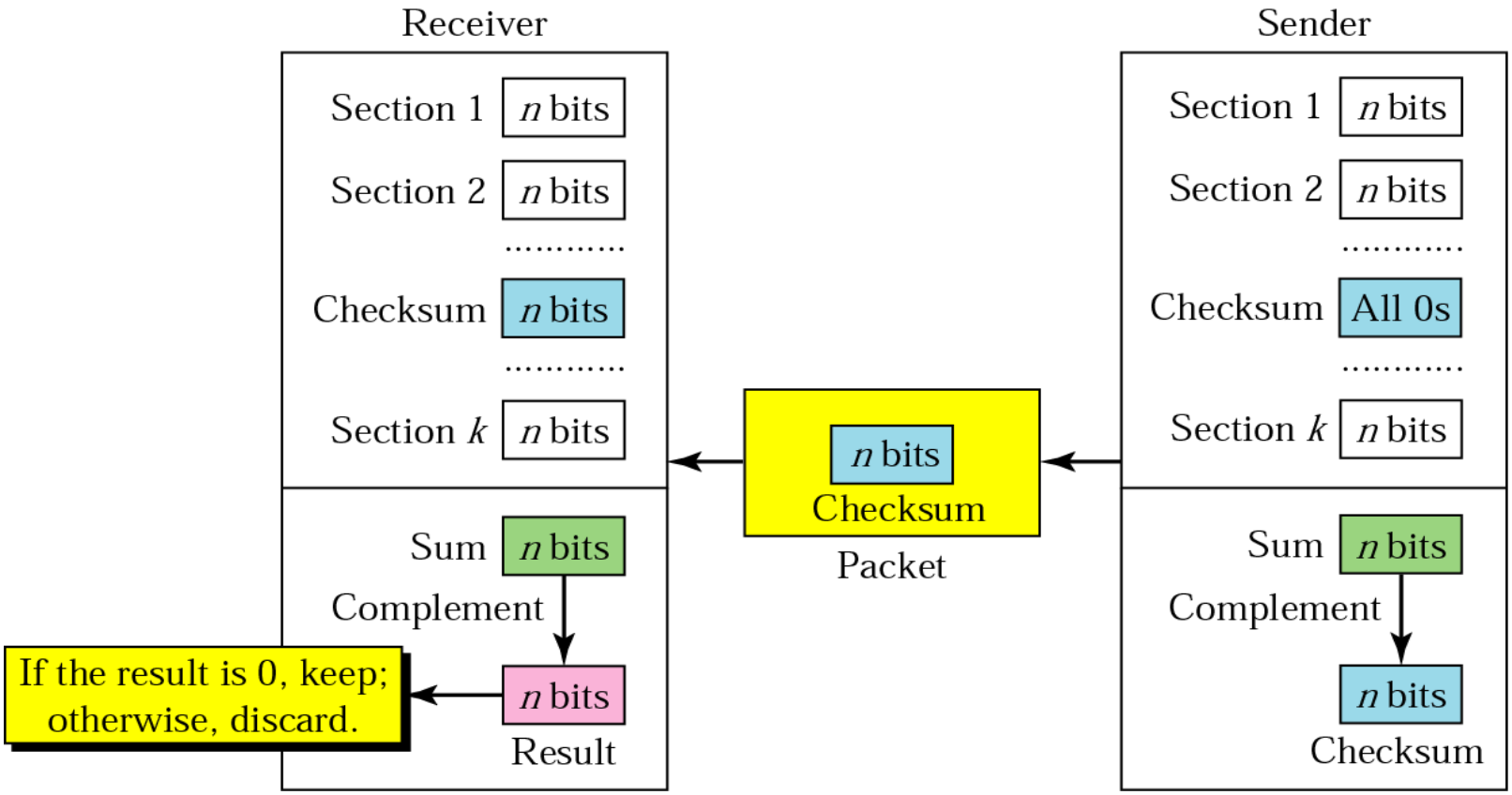
CHECKSUM

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

Data unit and checksum



Checksum


Receiver
Sender
Section 1
n bits
Section 2
n bits
Checksum
n bits
Section k
n bits
All 0s
Sum
Complement
Result
Checksum
Packet
If the result is 0, keep;
otherwise, discard.

The sender follows these steps:

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

The receiver follows these steps:

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

Correction

Retransmission

Error Correction

Hamming Code Algorithm

- *All bit positions that are powers of two are used as parity bits. (positions 1, 2, 4, 8, 16, 32, 64, etc.)*
- *All other bit positions are for the data to be encoded. (positions 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, etc.)*
- *Each parity bit calculates the parity for some of the bits in the code word. The position of the parity bit determines the sequence of bits that it alternately checks and skips.*
  1. *Position 1 (n=1): check 1 bit (n), skip 1 bit (n), check 1 bit (n), skip 1 bit (n), etc. (1,3,5,7,9,11,13,15,...)*
  2. *Position 2 (n=2): check 2 bits (n), skip 2 bits (n), check 2 bits (n), skip 2 bits (n), etc. (2,3,6,7,10,11,14,15,...)*
  3. *Position 4 (n=4): check 4 bits (n), skip 4 bits (n), check 4 bits (n), skip 4 bits (n), etc. (4,5,6,7,12,13,14,15,20,21,22,23,...)*
  4. *Position 8 (n=8): check 8 bits (n), skip 8 bits (n), check 8 bits (n), skip 8 bits (n), etc. (8-15,24-31,40-47,...)*
  5. *Position 16 (n=16): check 16 bits (n), skip 16 bits (n), check 16 bits (n), skip 16 bits (n), etc. (16-31,48-63,80-95,...)*
  6. *Position 32 (n=32): check 32 bits (n), skip 32 bits (n), check 32 bits (n), skip 32 bits (n), etc. (32-63,96-127,160-191,...)*
  - *General rule for position n: check n bits, skip n bits, check n bits...*

  And so on…..

Hamming Code Algorithm

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Encoded data bits		p1	p2	d1	p3	d2	d3	d4	p4	d5	d6	d7	d8	d9	d10	d11	p5	d12	d13	d14	d15
Parity bit coverage	p1	X		X		X		X		X		X		X		X		X		X	
	p2		X	X			X	X			X	X			X	X			X	X	
	p3				X	X	X	X					X	X	X	X					X
	p4								X	X	X	X	X	X	X	X					
	p5																X	X	X	X	X

Data Link Control

1 FRAMING

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.

Figure 1 *A frame in a character-oriented protocol*

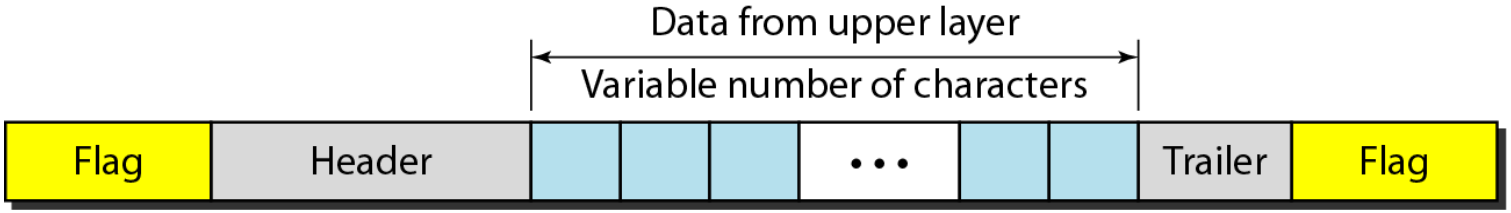
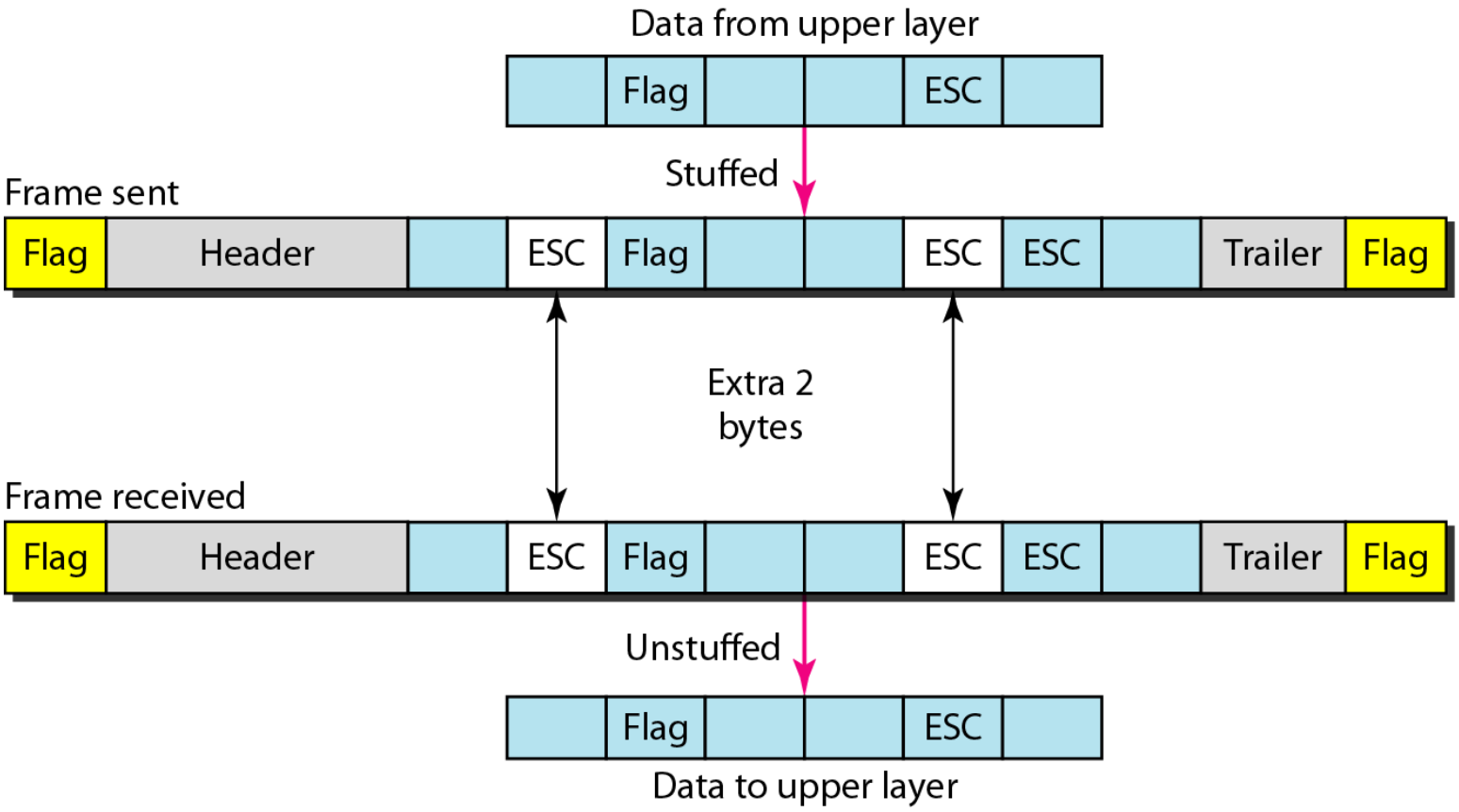


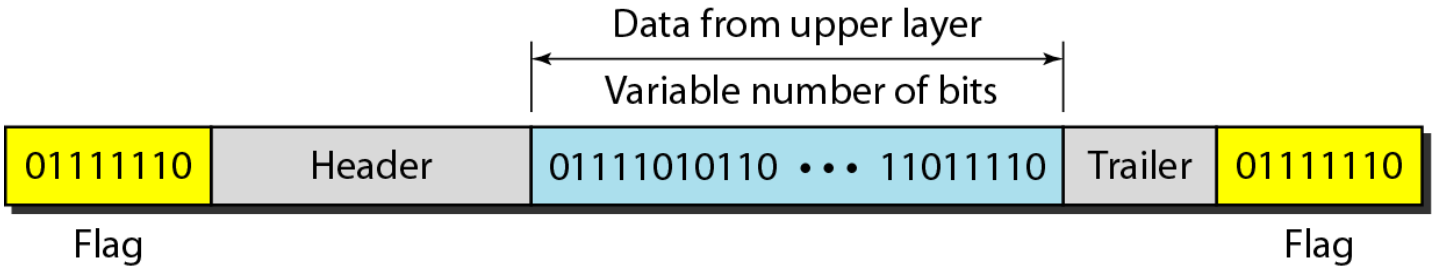
Figure 2 *Byte stuffing and unstuffing*



Note

Byte stuffing is the process of adding 1 extra byte whenever there is a flag or escape character in the text.

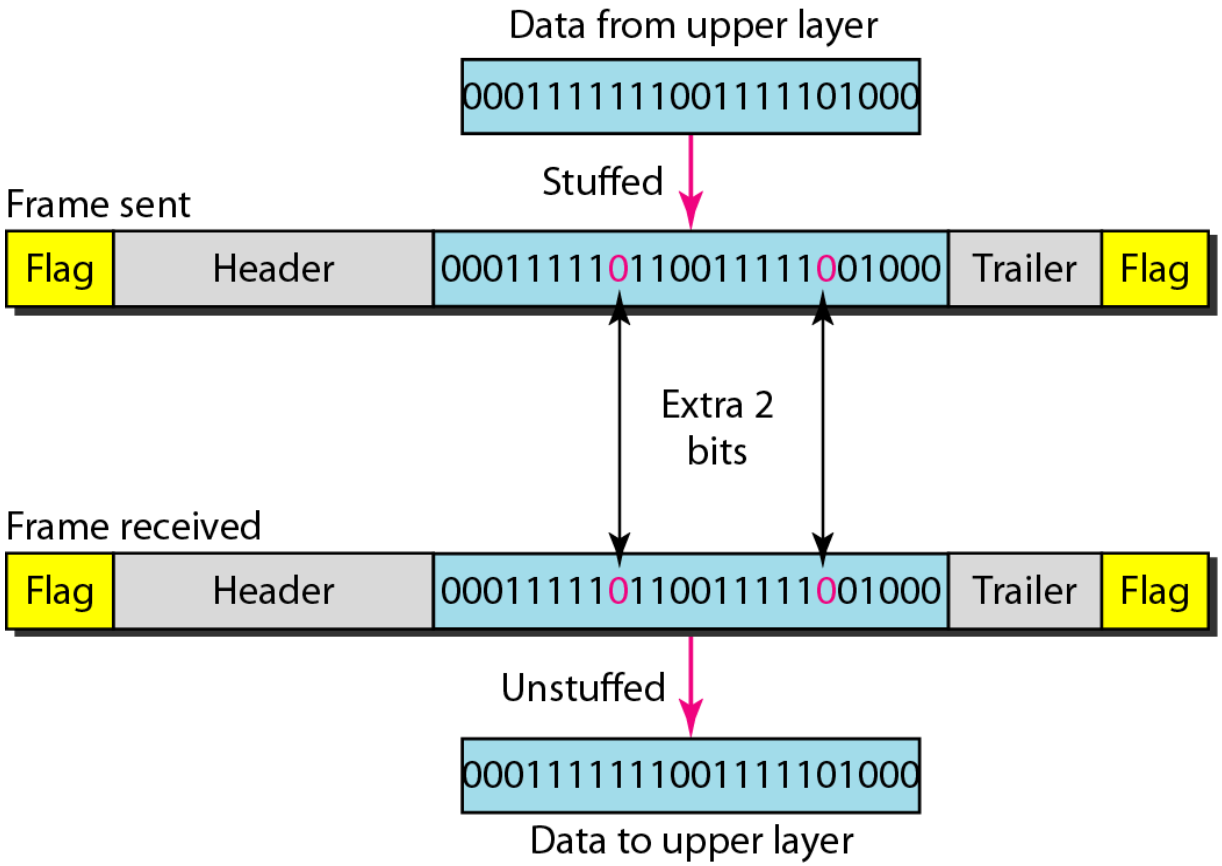
Figure 3 *A frame in a bit-oriented protocol*



Note

In HDLC 01111110 is a flag byte.
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 011111010 for a flag.

Figure 4 *Bit stuffing and unstuffing*



2 FLOW AND ERROR CONTROL

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

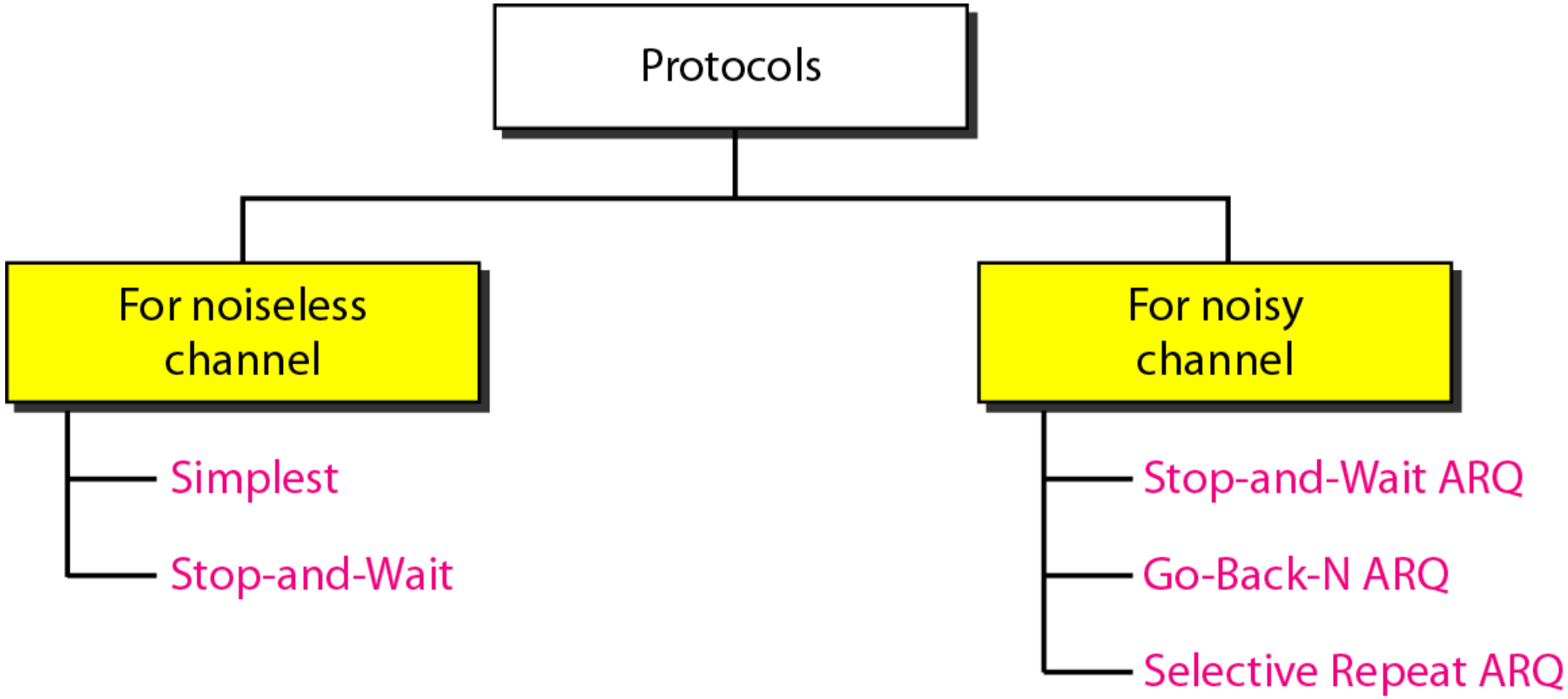
Note

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.

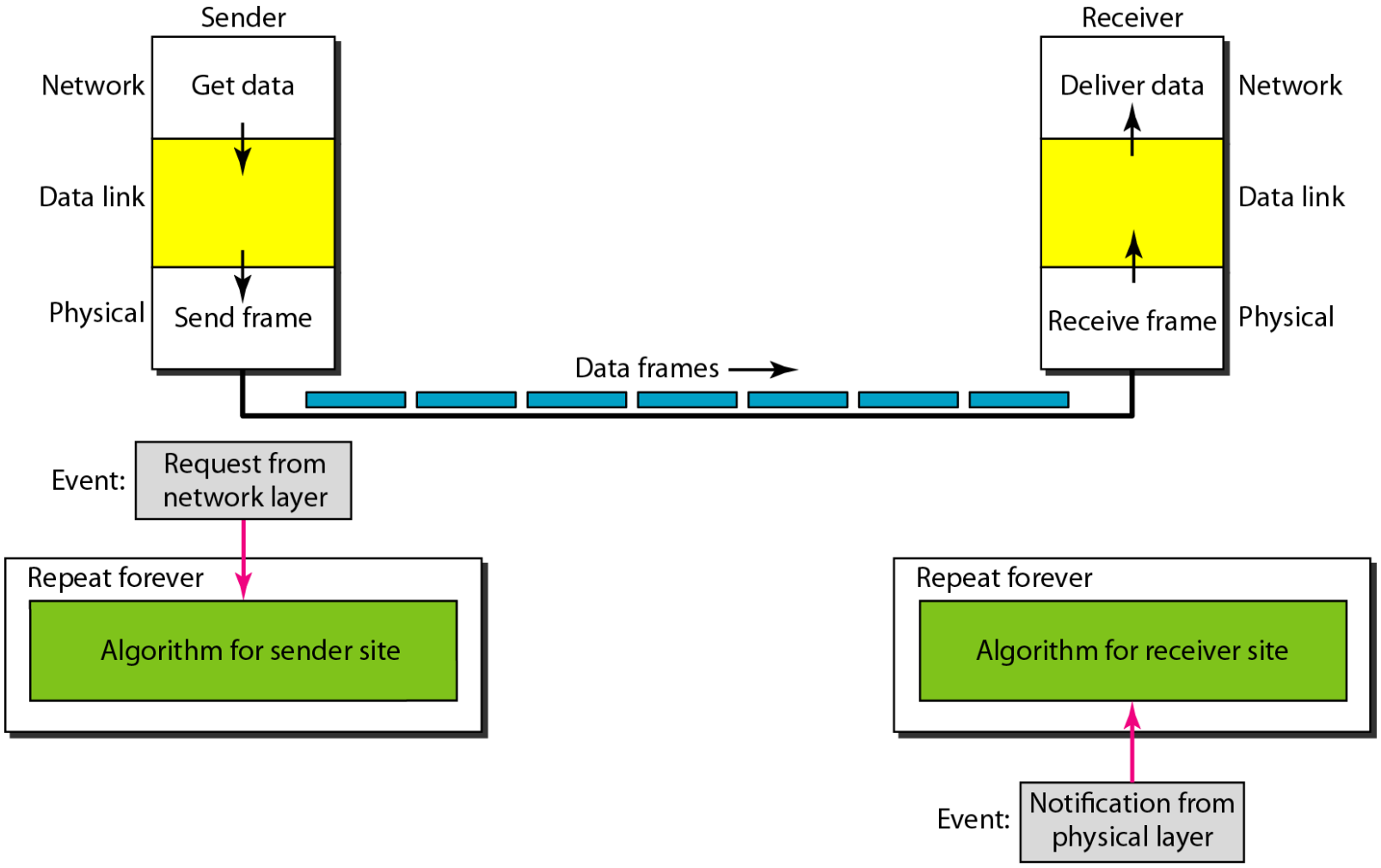
Figure 5 *Taxonomy of protocols discussed in this chapter*



4 NOISELESS CHANNELS

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.

Figure 6 *The design of the simplest protocol with no flow or error control*



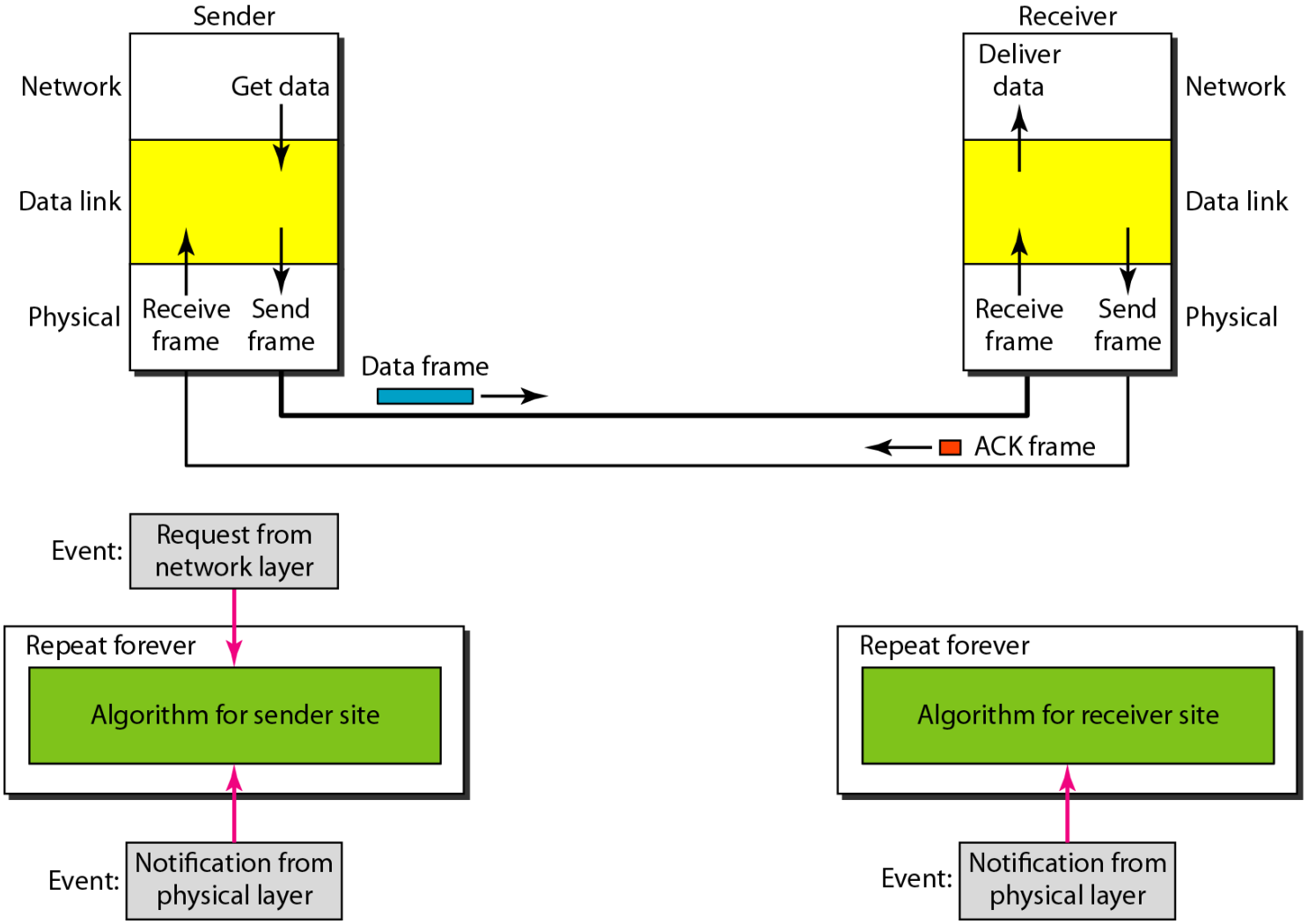
Algorithm 1 *Sender-site algorithm for the simplest protocol*

```
 1  while(true)                          // Repeat forever
 2  {
 3    WaitForEvent();                    // Sleep until an event occurs
 4    if(Event(RequestToSend))           //There is a packet to send
 5    {
 6       GetData();
 7       MakeFrame();
 8       SendFrame();                    //Send the frame
 9    }
10  }
```

Algorithm 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  }
```

Figure 8 *Design of Stop-and-Wait Protocol*



Algorithm 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 }
```

Algorithm 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 }
```

5 NOISY CHANNELS

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

Note

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.

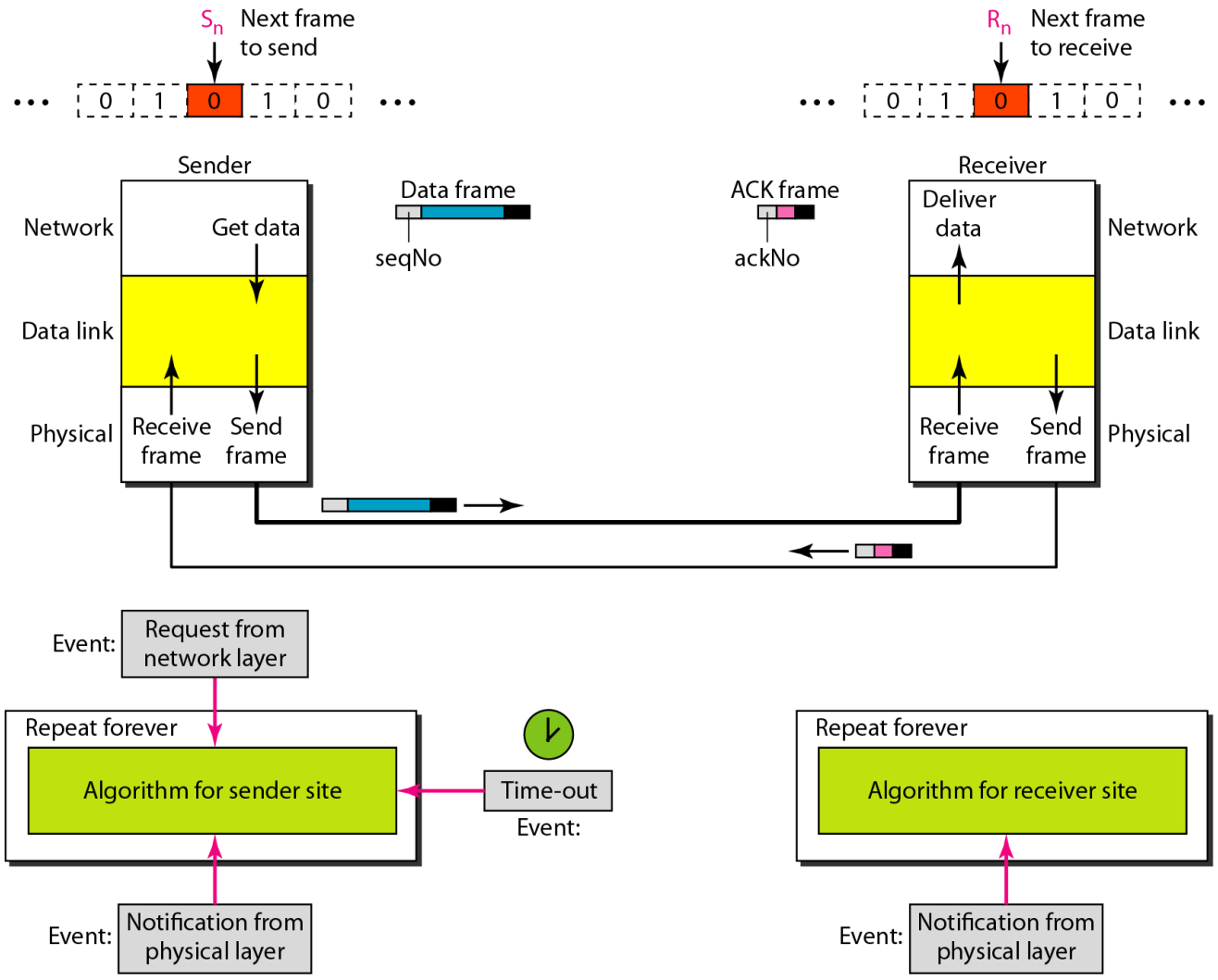
Note

In Stop-and-Wait ARQ, we use sequence numbers to number the frames. The sequence numbers are based on modulo-2 arithmetic.

Note

In Stop-and-Wait ARQ, the acknowledgment number always announces in modulo-2 arithmetic the sequence number of the next frame expected.

Figure 10 *Design of the Stop-and-Wait ARQ Protocol*



Algorithm 5 *Sender-site algorithm for Stop-and-Wait ARQ*

```
1  Sn = 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(Sn);                        //The seqNo is Sn
10      StoreFrame(Sn);                       //Keep copy
11      SendFrame(Sn);
12      StartTimer();
13      Sn = Sn + 1;
14      canSend = false;
15   }
16   WaitForEvent();                          // Sleep
```

(continue

Algorithm 5 *Sender-site algorithm for Stop-and-Wait ARQ* (continu

```
17      if(Event(ArrivalNotification)          // An ACK has arrived
18      {
19        ReceiveFrame(ackNo);                 //Receive the ACK frame
20        if(not corrupted AND ackNo == Sn)    //Valid ACK
21          {
22            Stoptimer();
23            PurgeFrame(Sn-1);                //Copy is not needed
24            canSend = true;
25          }
26       }
27
28      if(Event(TimeOut)                      // The timer expired
29      {
30       StartTimer();
31       ResendFrame(Sn-1);                    //Resend a copy check
32      }
33  }
```

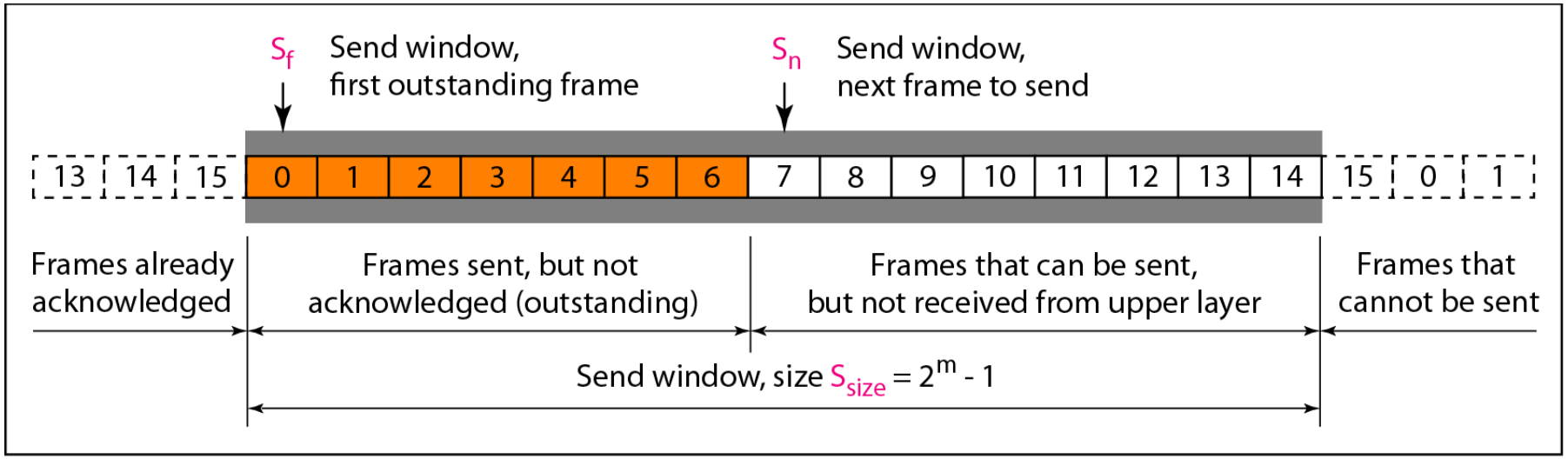
Algorithm 6 *Receiver-site algorithm for Stop-and-Wait ARQ Protocol*

```
1  Rn = 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 == Rn)              //Valid data frame
11      {
12       ExtractData();
13        DeliverData();             //Deliver data
14        Rn = Rn + 1;
15      }
16       SendFrame(Rn);             //Send an ACK
17   }
18 }
```

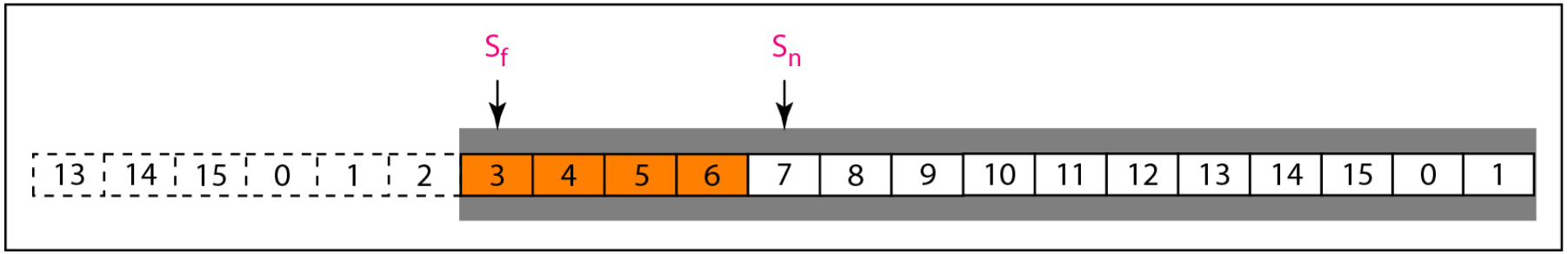
Note

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 12 *Send window for Go-Back-N ARQ*



a. Send window before sliding



b. Send window after sliding

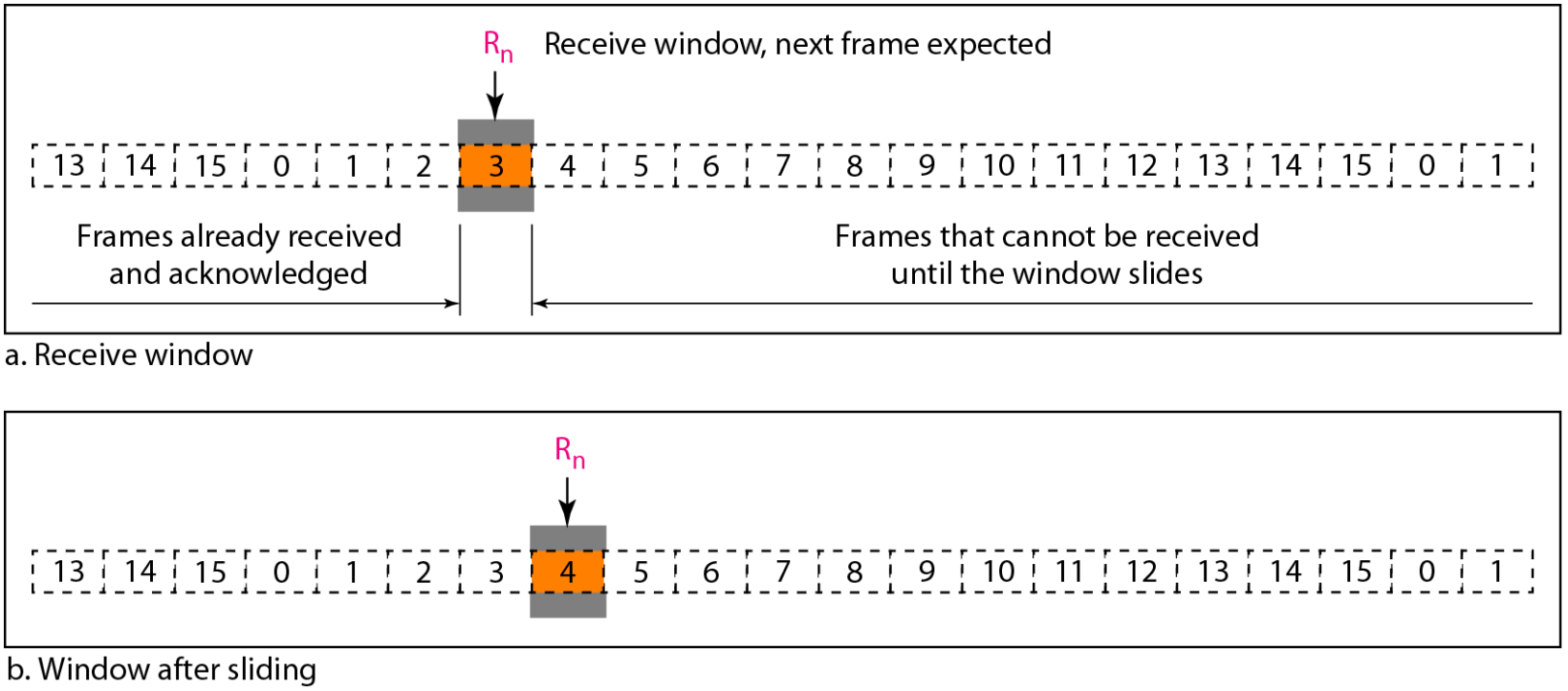
Note

The send window is an abstract concept defining an imaginary box of size $2^m - 1$ with three variables: Sf, Sn, and S_{size}.

Note

The send window can slide one or more slots when a valid acknowledgment arrives.

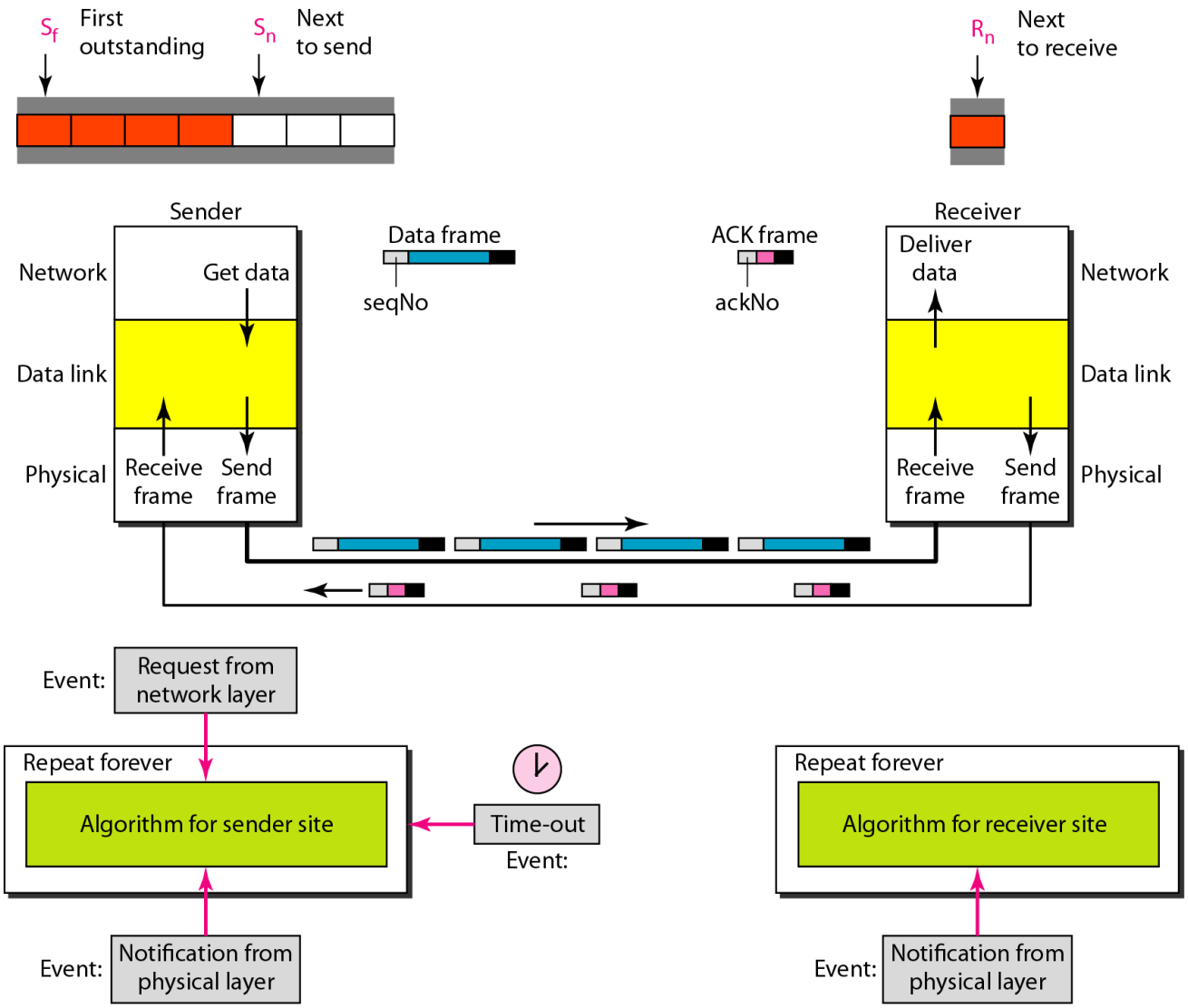
Figure 13 *Receive window for Go-Back-N ARQ*



Note

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 14 *Design of Go-Back-N ARQ*



Algorithm 7 *Go-Back-N sender algorithm*

```
1  Sw = 2^m - 1;
2  Sf = 0;
3  Sn = 0;
4
5  while (true)                          //Repeat forever
6  {
7   WaitForEvent();
8    if(Event(RequestToSend))            //A packet to send
9    {
10      if(Sn-Sf >= Sw)                  //If window is full
11            Sleep();
12      GetData();
13      MakeFrame(Sn);
14      StoreFrame(Sn);
15      SendFrame(Sn);
16      Sn = Sn + 1;
17      if(timer not running)
18           StartTimer();
19   }
20
```

(continue

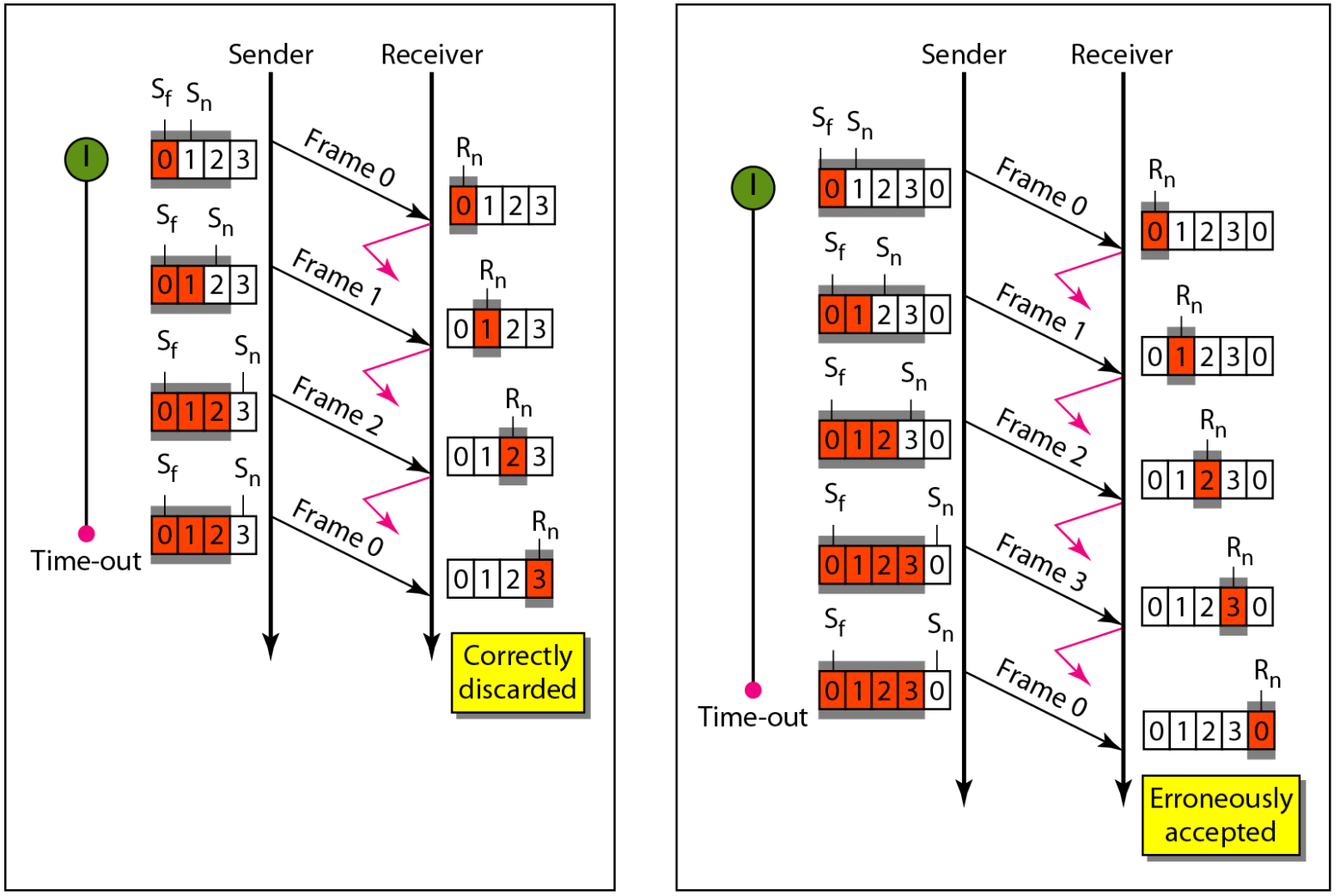
Algorithm 7 *Go-Back-N sender algorithm* (continue

```
21    if(Event(ArrivalNotification))  //ACK arrives
22    {
23       Receive(ACK);
24       if(corrupted(ACK))
25            Sleep();
26       if((ackNo>Sf)&&(ackNo<=Sn))  //If a valid ACK
27       While(Sf <= ackNo)
28        {
29         PurgeFrame(Sf);
30         Sf = Sf + 1;
31        }
32        StopTimer();
33    }
34
35    if(Event(TimeOut))                //The timer expires
36    {
37     StartTimer();
38     Temp = Sf;
39     while(Temp < Sn);
40      {
41       SendFrame(Sf);
42       Sf = Sf + 1;
43      }
44    }
45  }
```

Algorithm 8 *Go-Back-N receiver algorithm*

```
 1  Rn = 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 == Rn)                  //If expected frame
13       {
14         DeliverData();                 //Deliver data
15         Rn = Rn + 1;                   //Slide window
16         SendACK(Rn);
17       }
18    }
19  }
```

Figure 15 *Window size for Go-Back-N ARQ*



a. Window size < 2^m

b. Window size = 2^m

Note

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.

Note

Stop-and-Wait ARQ is a special case of Go-Back-N ARQ in which the size of the send window is 1.

Figure 18 *Send window for Selective Repeat ARQ*

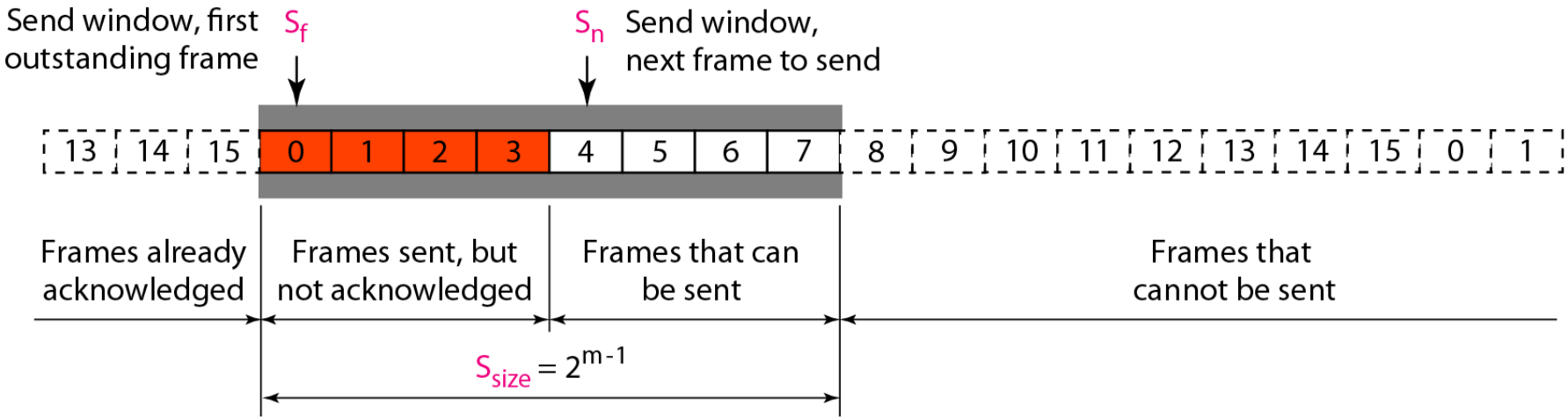


Figure 19 *Receive window for Selective Repeat ARQ*

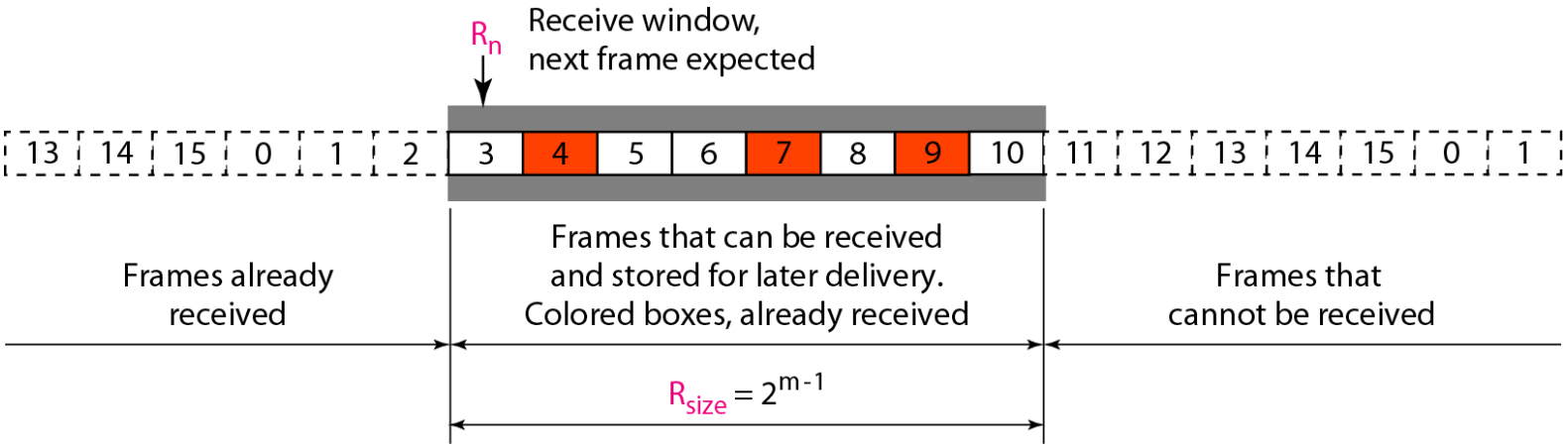
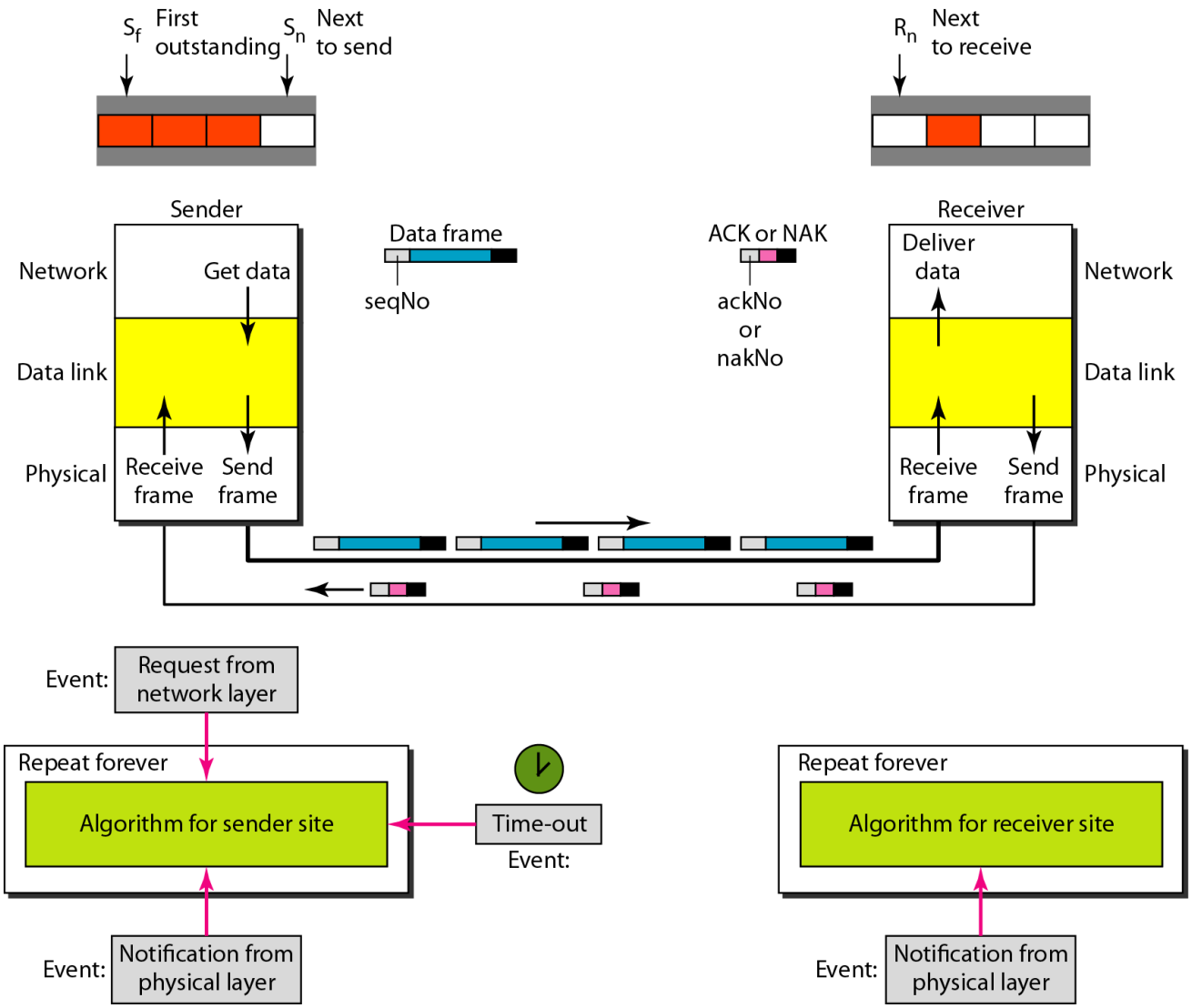


Figure 20 *Design of Selective Repeat ARQ*



Algorithm 9 *Sender-site Selective Repeat algorithm*

```
1  Sw = 2^(m-1) ;
2  Sf = 0;
3  Sn = 0;
4
5  while (true)                          //Repeat forever
6  {
7    WaitForEvent();
8    if(Event(RequestToSend))            //There is a packet to send
9    {
10     if(Sn-Sf >= Sw)                   //If window is full
11           Sleep();
12     GetData();
13     MakeFrame(Sn);
14     StoreFrame(Sn);
15     SendFrame(Sn);
16     Sn = Sn + 1;
17     StartTimer(Sn);
18   }
19
```

(continued)

Algorithm 9 *Sender-site Selective Repeat algorithm* (continue

```
20  if(Event(ArrivalNotification)) //ACK arrives
21  {
22     Receive(frame);              //Receive ACK or NAK
23     if(corrupted(frame))
24          Sleep();
25     if (FrameType == NAK)
26        if (nakNo between Sf and Sn)
27        {
28         resend(nakNo);
29         StartTimer(nakNo);
30        }
31     if (FrameType == ACK)
32        if (ackNo between Sf and Sn)
33        {
34          while(sf < ackNo)
35          {
36           Purge(sf);
37           StopTimer(sf);
38           Sf = Sf + 1;
39          }
40        }
41  }
```

(continue

Algorithm 9 *Sender-site Selective Repeat algorithm* (continue

```
42
43    if(Event(TimeOut(t)))              //The timer expires
44    {
45     StartTimer(t);
46     SendFrame(t);
47    }
48  }
```

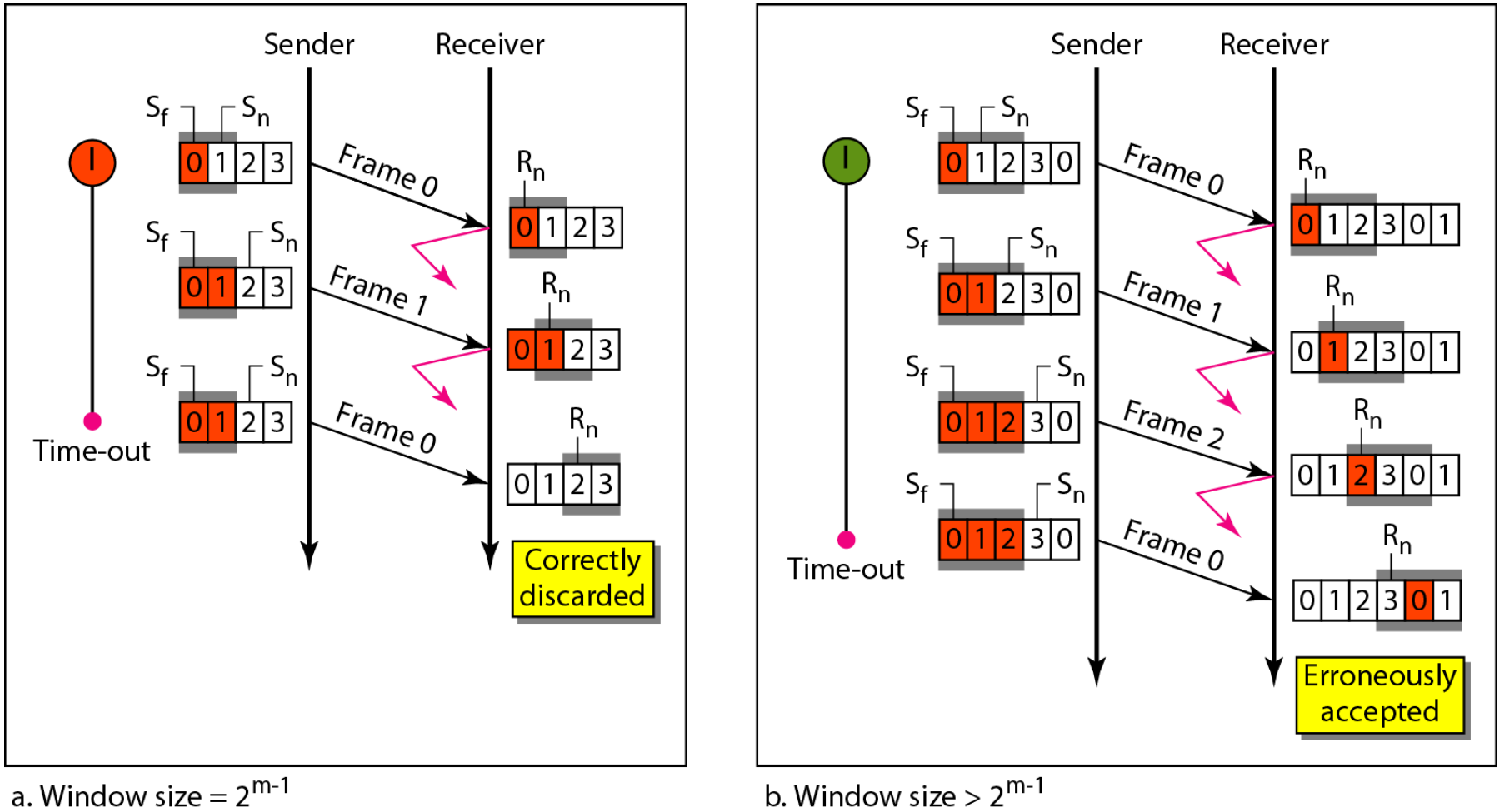
Algorithm 10 *Receiver-site Selective Repeat algorithm*

```
1  Rn = 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(Rn);
17       NakSent = true;
18       Sleep();
19      }
20      if(seqNo <> Rn)&& (NOT NakSent)
21      {
22       SendNAK(Rn);
```

Algorithm 10 *Receiver-site Selective Repeat algorithm*

```
23          NakSent = true;
24          if ((seqNo in window)&&(!Marked(seqNo))
25          {
26           StoreFrame(seqNo)
27           Marked(seqNo)= true;
28           while(Marked(Rn))
29           {
30            DeliverData(Rn);
31            Purge(Rn);
32            Rn = Rn + 1;
33            AckNeeded = true;
34           }
35            if(AckNeeded);
36            {
37            SendAck(Rn);
38            AckNeeded = false;
39            NakSent = false;
40            }
41          }
42        }
43    }
44  }
```

Figure 21 *Selective Repeat ARQ, window size*



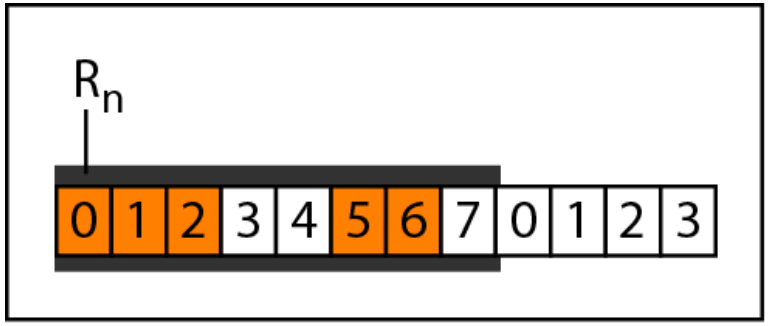
a. Window size = 2^{m-1}

b. Window size > 2^{m-1}

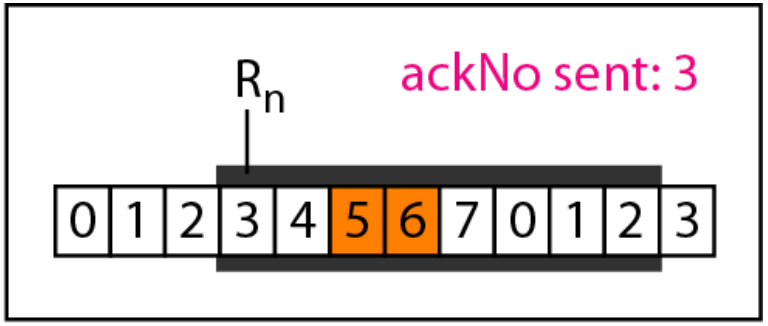
Note

In Selective Repeat ARQ, the size of the sender and receiver window must be at most one-half of 2^m.

Figure 22 *Delivery of data in Selective Repeat ARQ*



a. Before delivery



b. After delivery

Figure 24 *Design of piggybacking in Go-Back-N ARQ*