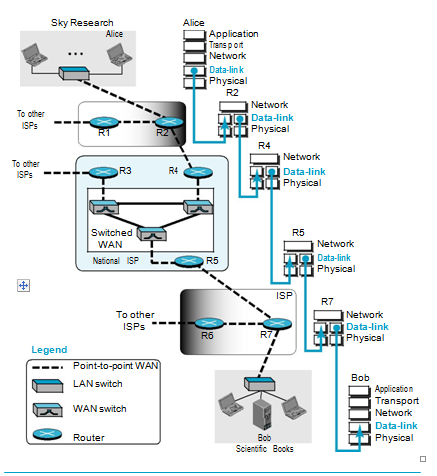
**Introduction To** **Data-Link Layer**: The Internet is a combination of networks glued together by connecting devices (routers or switches). If a packet is to travel from a host to another host, it needs to pass through these networks. Communication at the data-link layer is made up of five separate logical connections between the data-link layers in the path

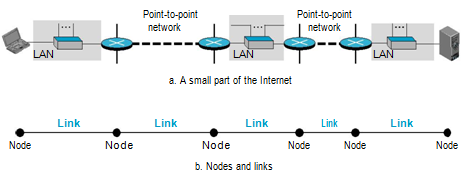


*Communication at the data-link layer*

The data-link layer at Alice’s computer communicates with the data-link layer at router R2. The data-link layer at router R2 communicates with the data-link layer at router R4, and so on. Finally, the data-link layer at router R7 communicates with the data-link layer at Bob’s computer. Only one data-link layer is involved at the source or the destination, but two data-link layers are involved at each router. The reason is that Alice’s and Bob’s computers are each connected to a single network, but each router takes input from one network and sends output to another network. Note that although switches are also involved in the data link-layer communication, for simplicity we have not shown them in the figure.

* + 1. **Nodes and Links**

Communication at the data-link layer is node-to-node. A data unit from one point in the Internet needs to pass through many networks (LANs and WANs) to reach another point. Theses LANs and WANs are connected by routers. It is customary to refer to the two end hosts and the routers as ***nodes*** and the networks in between as ***links*.** Figure is a simple representation of links and nodes when the path of the data unit is only six nodes.

****

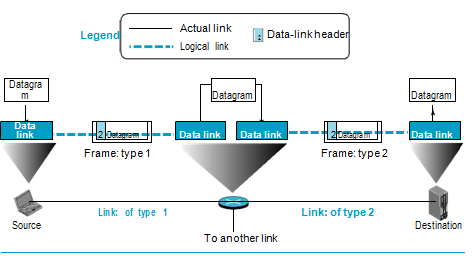
***Nodes and Links***

The first node is the source host; the last node is the destination host. The other four nodes are four routers. The first, the third, and the fifth links represent the three LANs; the second and the fourth links represent the two WANs.

**2.1.2.** **Services**

The data-link layer is located between the physical and the network layers. The data-link layer provides services to the network layer; it receives services from the physical layer

The duty scope of the data-link layer is node-to-node. When a packet is travelling in the Internet, the data link layer of a node (host or router) is responsible for delivering a datagram to the next node in the path. For this purpose, the data-link layer of the sending node needs to encapsulate the datagram received from the network in a frame, and the data-link layer of the receiving node needs to decapsulate the datagram from the frame.

****

***A communication with only three nodes***

Figure shows the encapsulation and decapsulation at the data-link layer. For simplicity, we have assumed that we have only one router between the source and destination. The datagram received by the data-link layer of the source host is encapsulated in a frame. The frame is logically transported from the source host to the router. The frame is decapsulated at the data-link layer of the router and encapsulated at another frame. The new frame is logically transported from the router to the destination host. Note that, although we have shown only two data-link layers at the router, the router actually has three data-link layers because it is connected to three physical links.

With the contents of the above figure, we can list the services provided by a data-link layer as shown below.

***Framing:*** The first service provided by the data-link layer is **framing**. The data-link layer at each node needs to encapsulate the datagram (packet received from the network layer) in a **frame** before sending it to the next node. The node also needs to decapsulate the datagram from the frame received on the logical channel. Different data-link layers have different formats for framing.

***Flow Control:*** The sending data-link layer at the end of a link is a producer of frames; the receiving data-link layer at the other end of a link is a consumer. If the rate of produced frames is higher than the rate of consumed frames, frames at the receiving end need to be buffered while waiting to be consumed (processed). we cannot have an unlimited buffer size at the receiving side. We have two choices. The first choice is to let the receiving data-link layer drop the frames if its buffer is full. The second choice is to let the receiving data-link layer send a feedback to the sending data-link layer to ask it to stop or slow down. Different data-link-layer protocols use different strategies for flow control.

***Error Control***

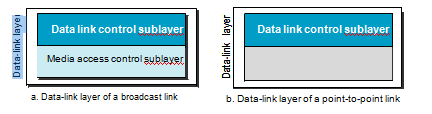
At the sending node, a frame in a data-link layer needs to be changed to bits, trans-formed to electromagnetic signals, and transmitted through the transmission media. At the receiving node, electromagnetic signals are received, transformed to bits, and put together to create a frame. Since electromagnetic signals are susceptible to error, a frame is susceptible to error. The error needs first to be detected. After detection, it needs to be either corrected at the receiver node or discarded and retransmitted by the sending node. Since error detection and correction is an issue in every layer.

***Congestion Control***

Although a link may be congested with frames, which may result in frame loss, most data-link-layer protocols do not directly use a congestion control to alleviate congestion, although some wide-area networks do

**Two Sublayers**

To better understand the functionality of and the services provided by the link layer, we can divide the data-link layer into two sublayers: **data link control (DLC)** and **media** **access control (MAC).** The data link control sublayer deals with all issues common to both point-to-point and broadcast links; the media access control sub-layer deals only with issues specific to broadcast links. In other words, we separate these two types of links at the data-link layer, as shown in Figure .

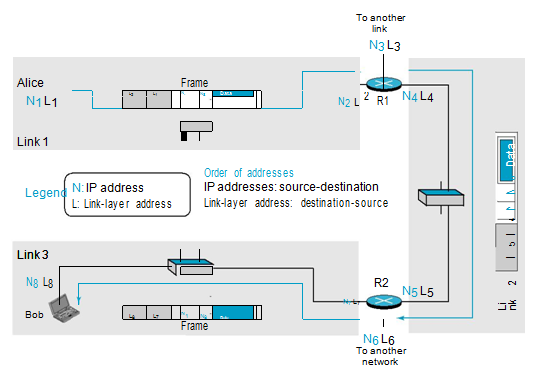


***Dividing the data-link layer into two sublayers***

* 1. **LINK-LAYER ADDRESSING**

A *link-layer* *address* is sometimes called a *link address*, sometimes a *physical address*, and some-times a *MAC address*.

Since a link is controlled at the data-link layer, the addresses need to belong to the data-link layer. When a datagram passes from the network layer to the data-link layer, the datagram will be encapsulated in a frame and two data-link addresses are added to the frame header. These two addresses are changed every time the frame moves from one link to another. Figure demonstrates the concept in a small internet.



***IP addresses and link-layer addresses in a small internet***

In the internet in Figure 9.5, we have three links and two routers. We also have shown only two hosts: Alice (source) and Bob (destination). For each host, we have shown two addresses, the IP addresses (N) and the link-layer addresses (L). Note that a router has as many pairs of addresses as the number of links the router is con-nected to. We have shown three frames, one in each link. Each frame carries the same datagram with the same source and destination addresses (**N1** and **N8**), but the link-layer addresses of the frame change from link to link. In link 1, the link-layer addresses are L1 and L2. In link 2, they are L4 and L5. In link 3, they are L7 and L8. Note that the IP addresses and the link-layer addresses are not in the same order. For IP addresses, the source address comes before the destination address; for link-layer addresses, the destination address comes before the source. The datagrams and frames are designed in this way, and we follow the design.

* + 1. **Three Types of addresses**

Some link-layer protocols define three types of addresses: unicast, multicast, and broadcast.

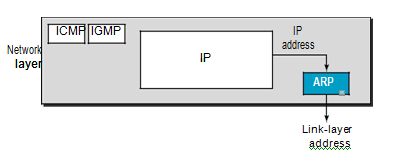
***Unicast Address***: Each host or each interface of a router is assigned a unicast address. Unicasting means one-to-one communication. A frame with a unicast address destination is destined only for one entity in the link.

***Multicast Address***: Some link-layer protocols define multicast addresses. Multicasting means one-to-many communication. However, the jurisdiction is local (inside the link).

***Broadcast Address:*** Some link-layer protocols define a broadcast address. Broadcasting means one-to-all communication. A frame with a destination broadcast address is sent to all entities in the link.

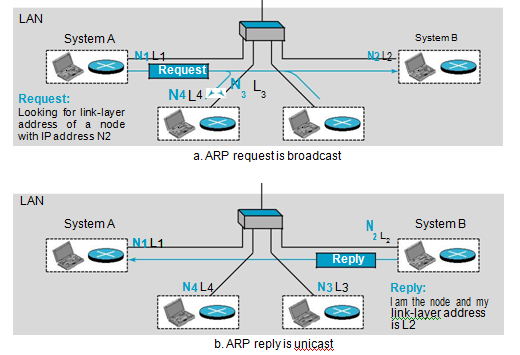
* + 1. **Address Resolution Protocol (ARP)**

The ARP protocol is one of the auxiliary protocols definedin the network layer, as shown in Figure . It belongs to the network layer, but we discuss it in this chapter because it maps an IP address to a logical-link address. ARP accepts an IP address from the IP protocol, maps the address to the corresponding link-layer address, and passes it to the data-link layer.

****

***Position of ARP in TCP/IP protocol suite***

Anytime a host or a router needs to find the link-layer address of another host or router in its network, it sends an ARP request packet. The packet includes the link-layer and IP addresses of the sender and the IP address of the receiver. Because the sender does not know the link-layer address of the receiver, the query is broadcast over the link using the link-layer broadcast address.



***ARP operation***

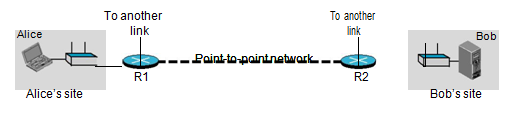
Every host or router on the network receives and processes the ARP request packet, but only the intended recipient recognizes its IP address and sends back an ARP response packet. The response packet contains the recipient’s IP and link-layer addresses. The packet is unicast directly to the node that sent the request packet.

In Figure (a), the system on the left (A) has a packet that needs to be delivered to another system (B) with IP address **N2**. System A needs to pass the packet to its data-link layer for the actual delivery, but it does not know the physical address of the recipient. It uses the services of ARP by asking the ARP protocol to send a broadcast ARP request packet to ask for the physical address of a system with an IP address of **N2**.

This packet is received by every system on the physical network, but only system B will answer it, as shown in Figure (b). System B sends an ARP reply packet that includes its physical address. Now system A can send all the packets it has for this destination using the physical address it received.

* + 1. **An Example of Communication**

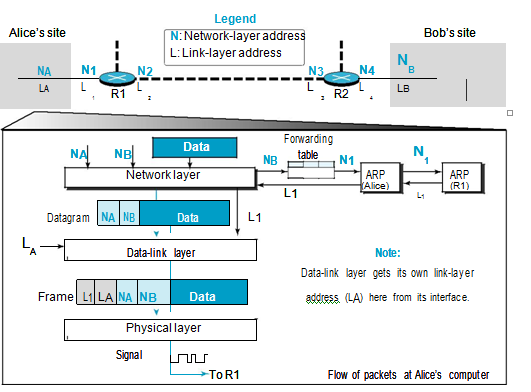
To show how communication is done at the data-link layer and how link-layer addresses are found, let us go through a simple example. Assume Alice needs to send a datagram to Bob, who is three nodes away in the Internet. How Alice finds the network-layer address of Bob. For the moment, assume that Alice knows the network-layer (IP) address of Bob. In other words, Alice’s host is given the data to be sent, the IP address of Bob, and the IP address of Alice’s host (each host needs to know its IP address). Figure shows the part of the internet for our example.



***The internet for our example***

***Activities at Alice’s Site***

We will use symbolic addresses to make the figures more readable. Figure shows what happens at Alice’s site.



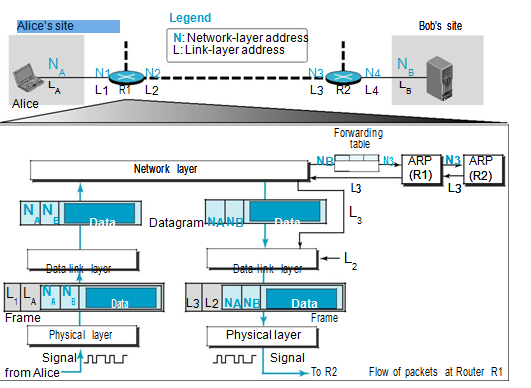
***Flow of packets at Alice’s computer***

The network layer knows it’s given **NA**, **NB**, and the packet, but it needs to find the link-layer address of the next node. The network layer consults its routing table and tries to find which router is next (the default router in this case) for the destination **NB**. the routing table gives **N1**, but the network layer needs to find the link-layer address of router R1. It uses its ARP to find the link-layer address **L1**.

The network layer can now pass the datagram with the link-layer address to the data-link layer. The data-link layer knows its own link-layer address, **LA**. It creates the frame and passes it to the physical layer, where the address is converted to signals and sent through the media.

***Activities at Router R1***

Now let us see what happens at Router R1. Router R1, as we know, has only three lower layers. The packet received needs to go up through these three layers and come down. Figure shows the activities.

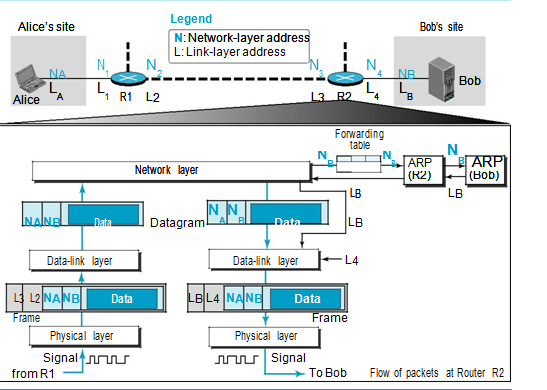


***Flow of activities at router R1***

At arrival, the physical layer of the left link creates the frame and passes it to the data-link layer. The data-link layer decapsulates the datagram and passes it to the net-work layer. The network layer examines the network-layer address of the datagram and finds that the datagram needs to be delivered to the device with IP address **NB**. The network layer consults its routing table to find out which is the next node (router) in the path to **NB**. The forwarding table returns **N3**. The IP address of router R2 is in the same link with R1. The network layer now uses the ARP to find the link-layer address of this router, which comes up as **L3.** The network layer passes the datagram and **L3** to the data-link layer belonging to the link at the right side. The link layer encapsulates the datagram, adds **L3** and **L2** (its own link-layer address), and passes the frame to the physical layer. The physical layer encodes the bits to signals and sends them through the medium to R2.

***Activities at Router R2***

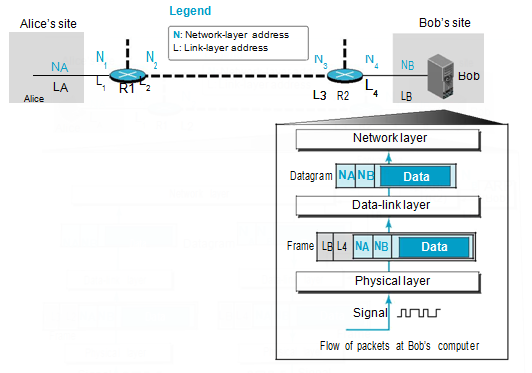
Activities at router R2 are almost the same as in R1, as shown in Figure.



***Activities at router R2****.*

***Activities at Bob’s Site*** : Now let us see what happens at Bob’s site. Figure shows how the signals at Bob’s site are changed to a message. At Bob’s site there are no more addresses or mapping needed. The signal received from the link is changed to a frame. The frame is passed to the data-link layer, which decapsulates the datagram and passes it to the network layer. The network layer decapsulates the message and passes it to the transport layer.

***Changes in Addresses***: This example shows that the source and destination network-layer addresses, NA and NB, have not been changed during the whole journey. However, all four network-layer addresses of routers R1 and R2 (N1, N2, N3, and N4) are needed to transfer a datagram from Alice’s computer to Bob’s computer.



***Error Detection and Correction***

* 1. **CYCLIC CODES:**

Cyclic codes are special linear block codes with one extra property. In a **cyclic code,** if a codeword is cyclically shifted (rotated), the result is another codeword. For example, if 1011000 is a codeword and we cyclically left-shift, then 0110001 is also a codeword. In this case, if we call the bits in the first word *a*0 to *a*6, and the bits in the second word *b*0to *b*6, we can shift the bits by using the following:

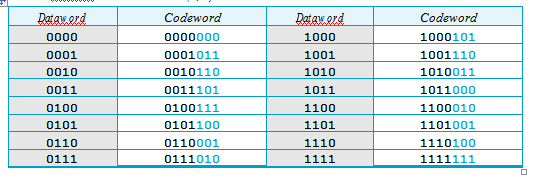
***b*1**5 ***a*0** ***b*2**5 ***a*1** ***b*3**5 ***a*2** ***b*4**5 ***a*3** ***b*5**5 ***a*4** ***b*6**5 ***a*5** ***b*0**5 ***a*6**

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

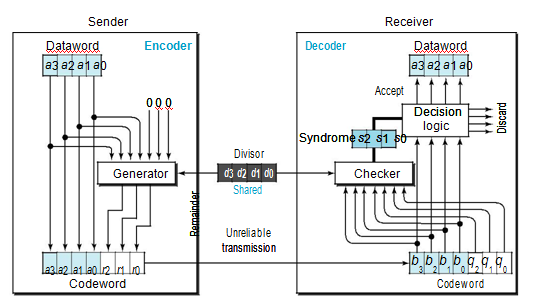
* + 1. **Cyclic Redundancy Check**

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

Table shows an example of a CRC code. We can see both the linear and cyclic properties of this code.



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



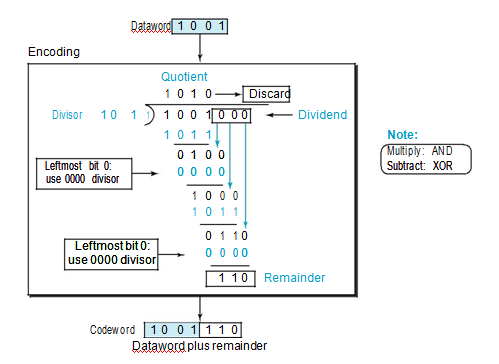
***CRC encoder and decoder***

In the encoder, the dataword has *k* bits (4 here); the codeword has *n* bits (7 here). The size of the dataword is augmented by adding *n* − *k* (3 here) 0s to the right-hand side of the word. The *n*-bit result is fed into the generator. The generator uses a divisor of size *n* − *k* 1 (4 here), predefined and agreed upon. The generator divides the augmenteddataword by the divisor (modulo-2 division). The quotient of the division is discarded; the remainder (*r*2*r*1*r*0) is appended to the dataword to create the codeword.

The decoder receives the codeword (possibly corrupted in transition). A copy of all *n* bits is fed to the checker, which is a replica of the generator. The remainder produced by the checker is a syndrome of *n* − *k* (3 here) bits, which is fed to the decision logic analyzer. The analyzer has a simple function. If the syndrome bits are all 0s, the 4 left-most bits of the codeword are accepted as the dataword (interpreted as no error); other-wise, the 4 bits are discarded (error).

***Encoder:***

Let us take a closer look at the encoder. The encoder takes a dataword and augments it with *n* − *k* number of 0s. It then divides the augmented dataword by the divisor, as shown in Figure.



***Division in CRC encoder***

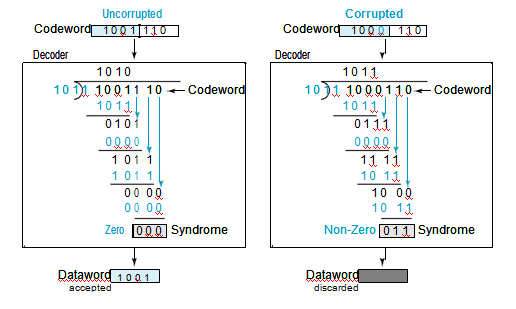
The process of modulo-2 binary division is the same as the familiar division process we use for decimal numbers. However, addition and subtraction in this case are the same; we use the XOR operation to do both.

As in decimal division, the process is done step by step. In each step, a copy of the divisor is XORed with the 4 bits of the dividend. The result of the XOR operation (remainder) is 3 bits (in this case), which is used for the next step after 1 extra bit is pulled down to make it 4 bits long. There is one important point we need to remember in this type of division. If the leftmost bit of the dividend (or the part used in each step) is 0, the step cannot use the regular divisor; we need to use an all-0s divisor.

When there are no bits left to pull down, we have a result. The 3-bit remainder forms the **check bits** (*r*2, *r*1, and *r*0). They are appended to the dataword to create the codeword.

***Decoder***

The codeword can change during transmission. The decoder does the same division process as the encoder. The remainder of the division is the syndrome. If the syndrome is all 0s, there is no error with a high probability; the dataword is separated from the received codeword and accepted. Otherwise, everything is discarded. Figure shows two cases: The left-hand figure shows the value of the syndrome when no error has occurred; the syndrome is 000. The right-hand part of the figure shows the case in which there is a single error. The syndrome is not all 0s (it is 011).

****

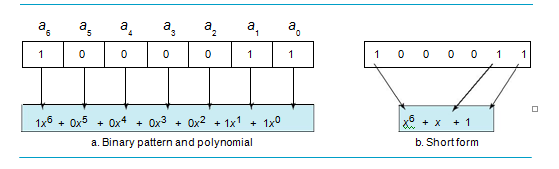
***Division in the CRC decoder for two cases***

***Divisor***: We may be wondering how the divisor 1011 is chosen. This depends on the expecta-tion we have from the code.

* + 1. **Polynomials**

A pattern of 0s and 1s can be represented as a **polynomial** with coefficients of 0 and 1. The power of each term shows the position of the bit; the coefficient shows the value of the bit. Figure shows a binary pattern and its polynomial representation. In Figure (a) we show how to translate a binary pattern into a polynomial; in Figure (b) we show how the polynomial can be shortened by removing all terms with zero coefficients and replacing *x*1 by *x* and *x*0 by 1.

Figure shows one immediate benefit; a 7-bit pattern can be replaced by three terms. The benefit is even more conspicuous when we have a polynomial such as *x*23 *x*31. Here the bit pattern is 24 bits in length (three 1s and twenty-one 0s) whilethe polynomial is just three terms.

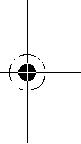
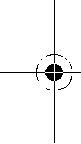


***A polynomial to represent a binary word***

***Degree of a Polynomial***

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

***Adding and Subtracting Polynomials***



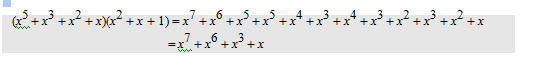
Adding and subtracting polynomials in mathematics are done by adding or subtracting the coefficients of terms with the same power. In our case, the coefficients are only 0 and 1, and adding is in modulo-2. This has two consequences. First, addition and sub-traction are the same. Second, adding or subtracting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

***Multiplying or Dividing Terms***

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

***Multiplying Two Polynomials***

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

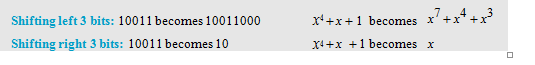


***Dividing One Polynomial by Another***

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

***Shifting***

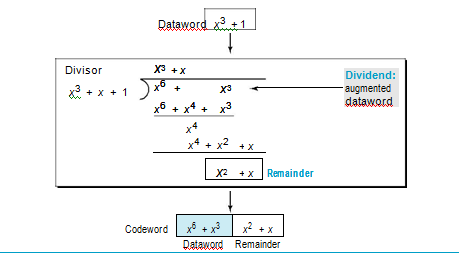
A binary pattern is often shifted a number of bits to the right or left. Shifting to the left means adding extra 0s as rightmost bits; shifting to the right means deleting some right-most bits. Shifting to the left is accomplished by multiplying each term of the polynomial by *xm*, where *m* is the number of shifted bits; shifting to the right is accomplished by dividing each term of the polynomial by *xm*. The following shows shifting to the left and to the right. Note that we do not have negative powers in the polynomial representation.



When we augmented the dataword in the encoder of Figure 10.6, we actually shifted the bits to the left. Also note that when we concatenate two bit patterns, we shift the first polynomial to the left and then add the second polynomial.

**2.3.3 Cyclic Code Encoder Using Polynomials**

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



***CRC division using polynomials***

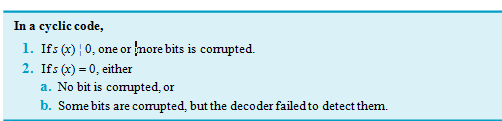
It can be seen that the polynomial representation can easily simplify the operation of division in this case, because the two steps involving all-0s divisors are not needed here. (Of course, one could argue that the all-0s divisor step can also be eliminated in binary division.) In a polynomial representation, the divisor is normally referred to as the ***generator polynomial*** *t*(*x*).

**2.3.4 Cyclic Code Analysis**

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

**Dataword: *d*(*x*)** **Codeword: *c*(*x*)** **Generator: *g*(*x*) Syndrome: *s*(*x*) Error: *e*(*x*)**

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



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

**Received codeword** 5 ***c*(*x*)** 1 ***e*(*x*)**

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Received**-------------------------------------------------**codeword** | 5 | ----------***c******x*** | 1 | ----------***e******x*** |
| ***g*** ***x*** |  | ***g*** ***x*** |  | ***g*** ***x*** |

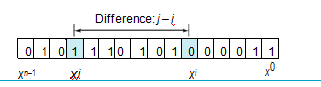
The first term at the right-hand side of the equality has a remainder of zero (according to the definition of codeword). So the syndrome is actually the remainder of the second term on the right-hand side. If this term does not have a remainder (syn-drome  0), either *e*(*x*) is 0 or *e*(*x*) is divisible by *g* (*x*). We do not have to worry about the first case (there is no error); the second case is very important. Those errors that are divisible by *g* (*x*) are not caught.

***Single-Bit Error***

What should the structure of *g* (*x*) be to guarantee the detection of a single-bit error? A single-bit error is *e*(*x*) = *xi*, where *i* is the position of the bit. If a single-bit error is caught, then *xi* is not divisible by *g*(*x*). (Note that when we say *not divisible,* we mean that there is a remainder.) If *g*(*x*) has at least two terms (which is normally the case) and the coeffi-cient of *x*0 is not zero (the rightmost bit is 1), then *e*(*x*) cannot be divided by *g*(*x*).

***Two Isolated Single-Bit Errors***

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



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

We can write *e*(*x*) =*xi*(*x* *j*–*i* + 1). If *g*(*x*) has more than one term and one term is *x*0, it cannot divide *xi*, as we saw in the previous section. So if *g*(*x*) is to divide *e*(*x*), it must divide *x j*–*i* +1. In other words, *g*(*x*) must not divide *xt* +1, where *t* is between 0 and *n* −1. How-ever, *t* = 0 is meaningless and *t* =1 is needed,. This means *t* should be between 2 and *n* – 1.

***Odd Numbers of Errors***

A generator with a factor of *x* + 1 can catch all odd numbers of errors. This means that we need to make *x* + 1 a factor of any generator. Note that we are not saying that the generator itself should be *x* + 1; we are saying that it should have a factor of *x* +1. If it is only *x* +1, it cannot catch the two adjacent isolated errors (see the previous section). For example, *x*4 + *x*2 + *x* + 1 can catch all odd-numbered errors since it can be written as a product of the two polynomials *x* + 1 and *x*3 + *x*2 + 1.

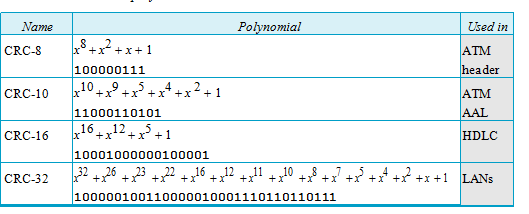
***Burst Errors***

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

1. If *j* − *i* < *r*, the remainder can never be zero. We can write *j* − *i*  *L* − 1, where *L* is the length of the error. So *L* − 1 < *r* or *L* < *r*  1 or *L* ð *r*. This means all burst errors with length smaller than or equal to the number of check bits *r* will be detected.
2. In some rare cases, if *j* − *i*  *r*, or *L*  *r*  1, the syndrome is 0 and the error is unde-tected. It can be proved that in these cases, the probability of undetected burst error of length *r*  1 is (1/2)*r*–1. For example, if our generator is *x*14  *x*3  1, in which *r*  14, a burst error of length *L*  15 can slip by undetected with the probability of (1/2)14–1 or almost 1 in 10,000.
3. In some rare cases, if *j* − *i* > *r*, or *L* > *r*  1, the syndrome is 0 and the error is unde-tected. It can be proved that in these cases, the probability of undetected burst error of length greater than *r*  1 is (1/2)*r*. For example, if our generator is *x*14  *x*3  1, in which *r*  14, a burst error of length greater than 15 can slip by undetected with the probability of (1/2)14 or almost 1 in 16,000 cases.

***Standard Polynomials***

Some standard polynomials used by popular protocols for CRC generation are shown in Table 10.4 along with the corresponding bit pattern.



***Standard polynomials***

**2.3.5 Hardware Implementation**

One of the advantages of a cyclic code is that the encoder and decoder can easily and cheaply be implemented in hardware by using a handful of electronic devices. Also, a hardware implementation increases the rate of check bit and syndrome bit calculation. In this section, we try to show, step by step, the process.

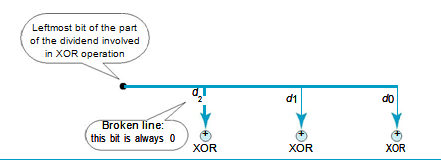
***Divisor***

Let us first consider the divisor. We need to note the following points:

1.The divisor is repeatedly XORed with part of the dividend.

2.The divisor has *n* − *k*  1 bits which either are predefined or are all 0s. In other words, the bits do not change from one dataword to another. In our previous example, the divisor bits were either 1011 or 0000. The choice was based on the leftmost bit of the part of the augmented data bits that are active in the XOR operation.

3. A close look shows that only *n* − *k* bits of the divisor are needed in the XOR opera-tion. The leftmost bit is not needed because the result of the operation is always 0, no matter what the value of this bit. The reason is that the inputs to this XOR operation are either both 0s or both 1s. In our previous example, only 3 bits, not 4, are actually used in the XOR operation.

****

***Hardwired design of the divisor in CRC***

Note that if the leftmost bit of the part of the dividend to be used in this step is 1, the divisor bits (*d*2*d*1*d*0) are 011; if the leftmost bit is 0, the divisor bits are 000. The design provides the right choice based on the leftmost bit.

***Augmented Dataword***

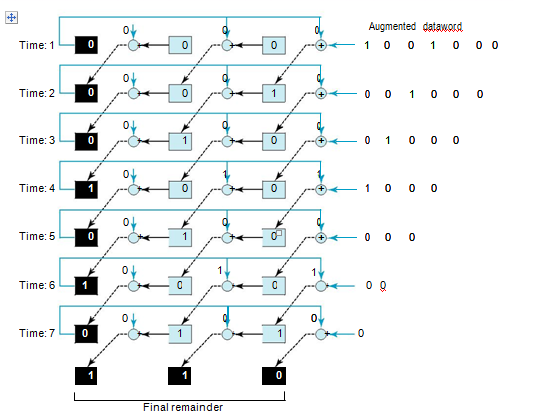
In our paper-and-pencil division process, we show the augmented data-word as fixed in position with the divisor bits shifting to the right, 1 bit in each step. The divisor bits are aligned with the appropriate part of the augmented dataword. Now that our divisor is fixed, we need instead to shift the bits of the augmented dataword to the left (opposite direction) to align the divisor bits with the appropriate part. There is no need to store the augmented dataword bits.

***Remainder***

In our previous example, the remainder is 3 bits (*n* − *k* bits in general) in length. We can use three **registers** (single-bit storage devices) to hold these bits. To find the final remainder of the division, we need to modify our division process. The following is the step-by-step process that can be used to simulate the division process in hardware (or even in software).

1. We assume that the remainder is originally all 0s (000 in our example).
2. At each time click (arrival of 1 bit from an augmented dataword), we repeat the following two actions:
   1. We use the leftmost bit to make a decision about the divisor (011 or 000).

b. The other 2 bits of the remainder and the next bit from the augmented dataword (total of 3 bits) are XORed with the 3-bit divisor to create the next remainder.

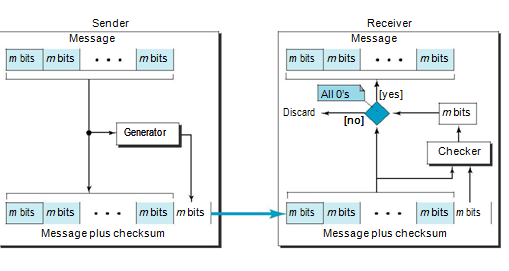


***Simulation of division in CRC encoder***

* 1. **CHECKSUM**

**Checksum** is an error-detecting technique that can be applied to a message of anylength. In the Internet, the checksum technique is mostly used at the network and trans-port layer rather than the data-link layer.

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

****

***Checksum***

**2.4.1 Concept**

The idea of the traditional checksum is simple. We show this using a simple example.

**Example**

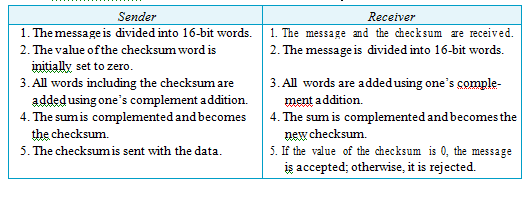
Suppose the message is a list of five 4-bit numbers that we want to send to a destination. In addition to sending these numbers, we send the sum of the numbers. For example, if the set of numbers is (7, 11, 12, 0, 6), we send (7, 11, 12, 0, 6, **36**), where 36 is the sum of the original numbers. The receiver adds the five numbers and compares the result with the sum. If the two are the same, the receiver assumes no error, accepts the five numbers, and discards the sum. Otherwise, there is an error somewhere and the message is not accepted.

***One’s Complement Addition***

The previous example has one major drawback. Each number can be written as a 4-bit word (each is less than 15) except for the sum. One solution is to use **one’s complement** arithmetic. In this arithmetic, we can represent unsigned numbers between 0and 2*m* − 1 using only *m* bits. If the number has more than *m* bits, the extra leftmost bits need to be added to the *m* rightmost bits (wrapping).

***Internet Checksum***

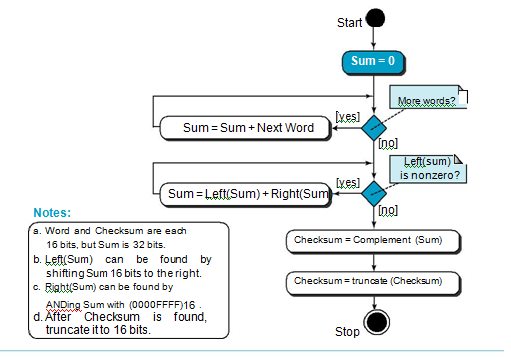
Traditionally, the Internet has used a 16-bit checksum. The sender and the receiver follow the steps depicted in Table. The sender or the receiver uses five steps



***Procedure to calculate the traditional checksum***

***Algorithm***

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

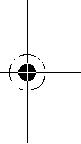
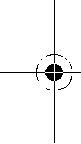


***Algorithm to calculate a traditional checksum***

**2.4.2 Other Approaches to the Checksum**

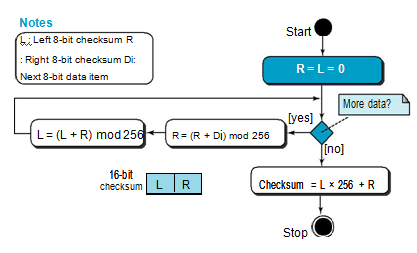
If two 16-bit items are transposed in transmission, the checksum cannot catch this error. The reason is that the traditional checksum is not weighted: it treats each data item equally. In other words, the order of data items is immaterial to the calculation. Several approaches have been used to prevent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

***Fletcher Checksum***



The Fletcher checksum was devised to weight each data item according to its position. Fletcher has proposed two algorithms: 8-bit and 16-bit. The first, 8-bit Fletcher, calculates on 8-bit data items and creates a 16-bit checksum. The second, 16-bit Fletcher, calculates on 16-bit data items and creates a 32-bit checksum.

The 8-bit Fletcher is calculated over data octets (bytes) and creates a 16-bit check-sum. The calculation is done modulo 256 (28), which means the intermediate results are divided by 256 and the remainder is kept. The algorithm uses two accumulators, L and R. The first simply adds data items together; the second adds a weight to the calculation. There are many variations of the 8-bit Fletcher algorithm; we show a simple one in Figure

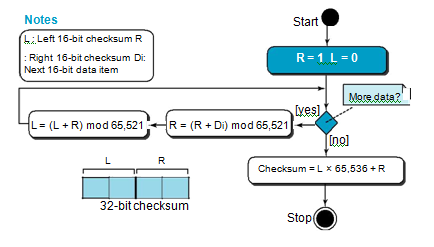


***Algorithm to calculate an 8-bit Fletcher checksum***

The 16-bit Fletcher checksum is similar to the 8-bit Fletcher checksum, but it is calculated over 16-bit data items and creates a 32-bit checksum. The calculation is done modulo 65,536.

***Adler Checksum***

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



***Algorithm to calculate an Adler checksum***

* 1. **FORWARD ERROR CORRECTION**

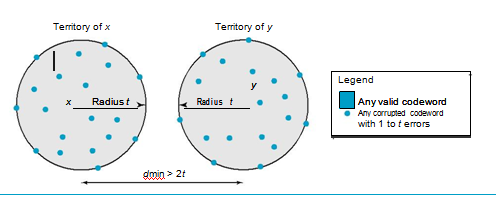
We discussed error detection and retransmission in the previous sections. However, retransmission of corrupted and lost packets is not useful for real-time multimedia transmission because it creates an unacceptable delay in reproducing: we need to wait until the lost or corrupted packet is resent. We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

**2.5.1 Using Hamming Distance**

To detect *s* errors, the minimum Hamming distance should be *d*min  *s*  1. For error detection, we definitely need more distance. It can be shown that to detect *t* errors, we need to have

*d*min2*t* 1. In other words, if we want to correct 10 bits in a packet, we need tomake the minimum hamming distance 21 bits, which means a lot of redundant bits

need to be sent with the data. To give an example, consider the famous BCH code. In this code, if data is 99 bits, we need to send 255 bits (extra 156 bits) to correct just 23 possible bit errors. Most of the time we cannot afford such a redundancy. We give some examples of how to calculate the required bits in the practice set. Figure 10.20 shows the geometrical representation of this concept.



***Hamming distance for error correction***

**2.5.2 Using XOR**

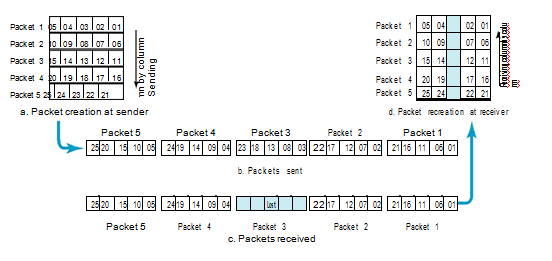
Another recommendation is to use the property of the exclusive OR operation as shown below.

**R**  **P1** ⊕ **P2** ⊕⊕ **P*i*** ⊕ **P*N*** → **P*i***  **P1** ⊕ **P2** ⊕⊕ **R** ⊕⊕ **P*N***

In other words, if we apply the exclusive OR operation on *N* data items (P1 to P*N*), we can recreate any of the data items by exclusive-ORing all of the items, replacing the one to be created by the result of the previous operation (R). This means that we can divide a packet into *N* chunks, create the exclusive OR of all the chunks and send *N*  1 chunks. If any chunk is lost or corrupted, it can be created at the receiver site. Now the question is what should the value of *N* be. If *N*  4, it means that we need to send 25 percent extra data and be able to correct the data if only one out of four chunks is lost.

**2.5.3 Chunk Interleaving**

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



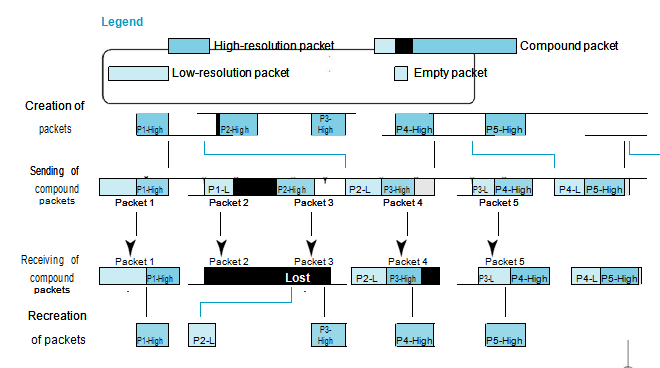
***Interleaving***

* + 1. **Combining Hamming Distance and Interleaving**

Hamming distance and interleaving can be combined. We can first create *n*-bit packets that can correct *t*-bit errors. Then we interleave *m* rows and send the bits column by column. In this way, we can automatically correct burst errors up to *m* + *t*-bit errors.

**2.5.5 Compounding High- and Low-Resolution Packets**

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



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