Module-3

DATA LINK LAYER

Error Detection and Correction

Data can be corrupted during transmission. Some applications require that errors be detected and corrected.

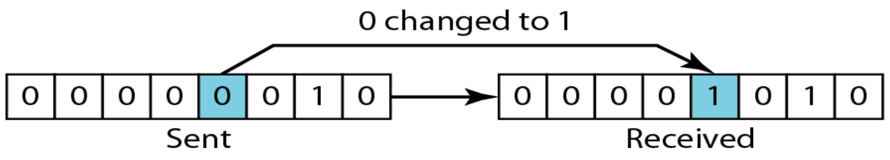
3.1 Introduction

Types of Errors

There are two types of error: Single bit error and Burst error.

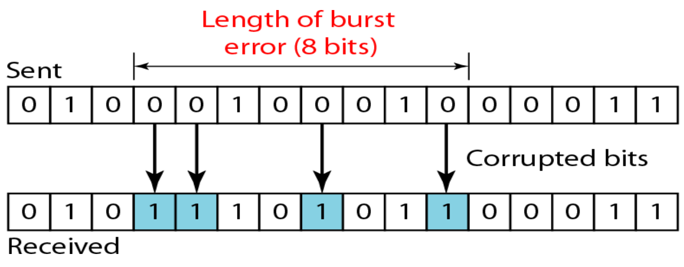
1) Single-Bit Error

The term single-bit error means that only 1 bit of a given data unit (such as a byte, character, or packet) is changed from 1 to 0 or from 0 to 1. Single-bit errors are the least likely type of error in serial data transmission.



2) Burst Error

The term burst error means that 2 or more bits in the data unit have changed from 1 to 0 or from 0 to 1. In the below figure, 0100010001000011 was sent, but 0101110101100011 was received.



Note that a burst error does not necessarily mean that the errors occur in consecutive bits. The length of the burst is measured from the first corrupted bit to the last corrupted bit. Some bits in between may not have been corrupted.

Redundancy

To detect or correct errors some extra bits are sent with data. These redundant bits are added by the sender and removed by the receiver. Their presence allows the receiver to detect or correct corrupted bits.

Detection versus Correction

- The correction of errors is more difficult than the detection.
- In error detection, we are looking only to see if any error has occurred.
- In error correction, we need to know the exact number of bits that are corrupted and more importantly, their location in the message. The number of the errors and the size of the message are important factors.
  If we need to correct one single error in an 8-bit data unit, we need to consider eight possible error locations; if we need to correct two errors in a data unit of the same size, we need to consider 28 possibilities.

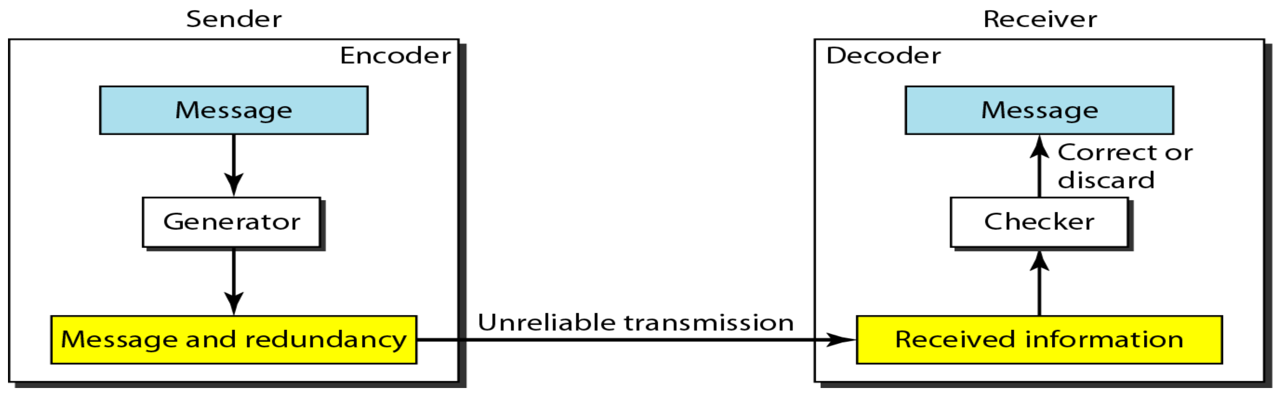
Forward Error Correction versus Retransmission

There are two main methods of error correction.

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

Coding

- Redundancy is achieved through various coding schemes.
- The sender adds redundant bits through a process that creates a relationship between the redundant bits and the actual data bits.
- The receiver checks the relationships between the two sets of bits to detect or correct the errors.
- Coding schemes can be divided into two broad categories: block coding and convolution coding.



Modular Arithmetic

In modular arithmetic only integers in the range 0 to N-1 is used. This is known as *modulo-N* arithmetic.

For example, if the modulus is 12, we use only the integers 0 to 11, inclusive.

Modulo-2 Arithmetic

In this arithmetic, the modulus *N* is 2. We can use only 0 and 1. Operations in this arithmetic are very simple. The following shows how we can add or subtract 2 bits.

Adding:

0+0=0 0+1=1 1+0=1 1+1=0

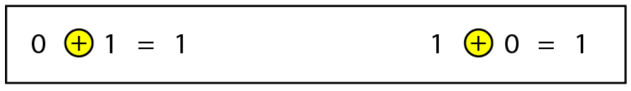
Subtracting:

0-0=0 0-1=1 1-0=1 1-1=0

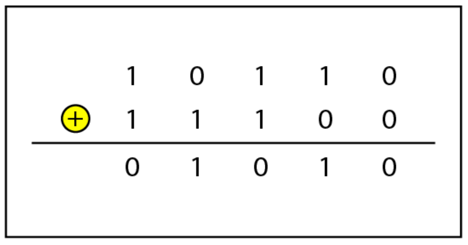
In this arithmetic we use the XOR (exclusive OR) operation for both addition and subtraction. The result of an XOR operation is 0 if two bits are the same; the result is I if two bits are different.

$0 \oplus 0 = 0$
$1 \oplus 1 = 0$

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



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

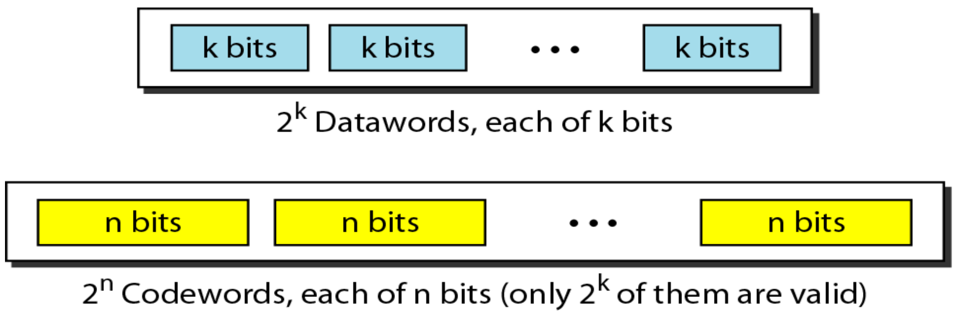


$$\begin{array}{rccccc} & 1 & 0 & 1 & 1 & 0 \\ \oplus & 1 & 1 & 1 & 0 & 0 \\ \hline & 0 & 1 & 0 & 1 & 0 \end{array}$$

c. Result of XORing two patterns

3.2 Block Coding

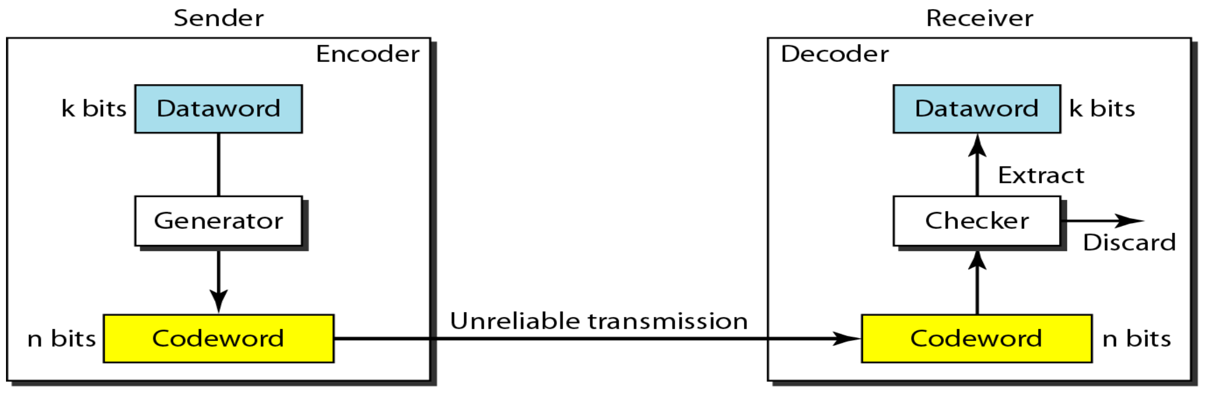
- In block coding message is divided into k bits blocks called **datawords**. Then *r* redundant bits are added to each block to make the length *n* = *k* + *r*. The resulting *n-bit* blocks are called **codewords**.
- With k bits, we can create a combination of 2k datawords; with n bits, we can create a combination of 2n codewords.
- Since n > k, the number of possible codewords is larger than the number of possible datawords.
- The block coding process is one-to-one; the same dataword is always encoded as the same codeword. This means that we have 2n - 2k codewords that are not used. We call these codewords invalid or illegal.


k bits
k bits
k bits
2^k Datawords, each of k bits
n bits
n bits
n bits
2^n Codewords, each of n bits (only 2^k of them are valid)

Error Detection

If the following two conditions are met, the receiver can detect a change in the original codeword.

1. The receiver has (or can find) a list of valid codewords.
2. The original codeword has changed to an invalid one.



Process of error detection in block coding

- The sender creates codewords out of datawords by using a generator that applies the rules and procedures of encoding.
- Each codeword sent to the receiver may change during transmission.
- If the received codeword is the same as one of the valid codewords, the word is accepted; the corresponding dataword is extracted for use. If the received codeword is not valid, it is discarded.
- However, if the codeword is corrupted during transmission but the received word still matches a valid codeword, the error remains undetected.
- This type of coding can detect only single errors. Two or more errors may remain undetected.

Example:

Let us assume that k =2 and n =3. Below Table shows the list of datawords and codewords.

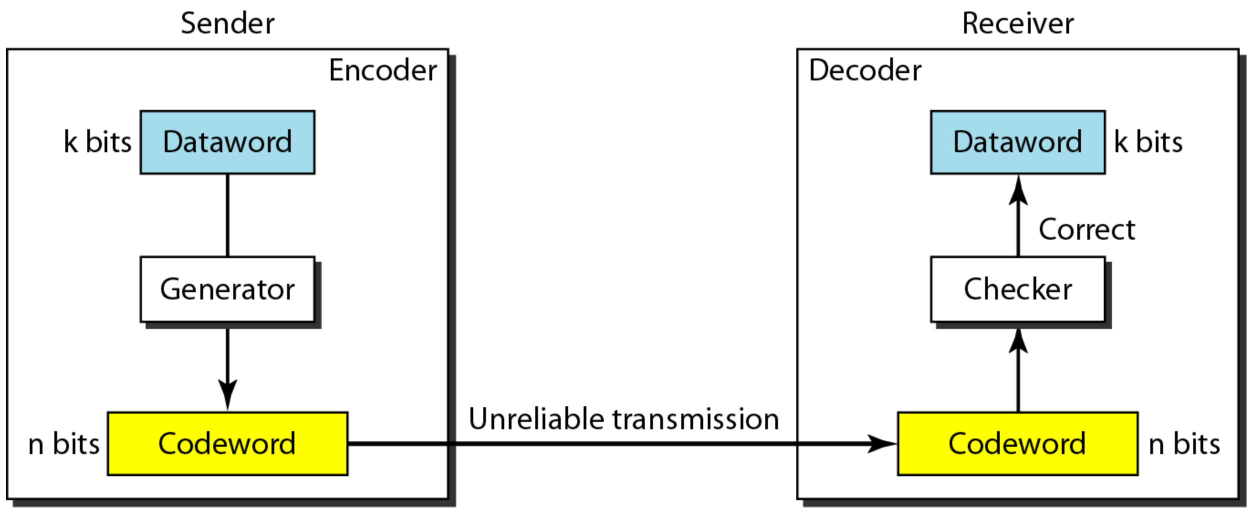
Dataword	Codeword
00	000
01	011
10	101
11	110

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 (the leftmost bit is corrupted).
   This is not a valid codeword and is discarded.
3. The codeword is corrupted during transmission, and 000 is received (the right two bits are corrupted). This is a valid codeword. The receiver incorrectly extracts the dataword 00. Two corrupted bits have made the error undetectable.

Error Correction

In error detection, the receiver needs to know only that the received codeword is invalid; in error correction the receiver needs to find (or guess) the original codeword sent.



Below Table shows the datawords and codewords.

Dataword	Codeword
00	00000
01	01011
10	10101
11	11110

Assume the dataword is 01. The sender consults the table (or uses an algorithm) to create the codeword 01011.

The codeword is corrupted during transmission, and 01001 is received (error in the second bit from the right).

First, the receiver finds that the received codeword is not in the table. This means an error has occurred. (Detection must come before correction.)

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.

Hamming Distance

The Hamming distance between two words (of the same size) is the number of differences between the corresponding bits. Hamming distance between two words x and y is represented as d(x, y).The Hamming distance can be found by applying the XOR operation on the two words and counting the number of 1s in the result.

Example:

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).

Minimum Hamming Distance: The minimum Hamming distance is the smallest Hamming distance between all possible pairs in a set of words. It is represented as d_{min}.

Example 1

Find the minimum Hamming distance of the coding scheme in below table:

Dataword	Codeword
00	000
01	011
10	101
11	110

Solution

d(000,011) = 2, d(000,101) = 2, d(000,110) = 2, d(011,101) = 2, d(011,110) = 2, d(101,110) = 2

The $\mathbf{d_{min}}$ in this case is 2.

Example 2

Find the minimum Hamming distance of the coding scheme in below table:

Dataword	Codeword
00	00000
01	01011
10	10101
11	11110

Solution

d(00000,01011) = 3, d(00000,10101) = 3, d(00000,11110) = 4

d(01011,10101) = 4, d(01011,11110) = 3, d(10101,11110) = 3

The $\mathbf{d_{min}}$ in this case is 3.

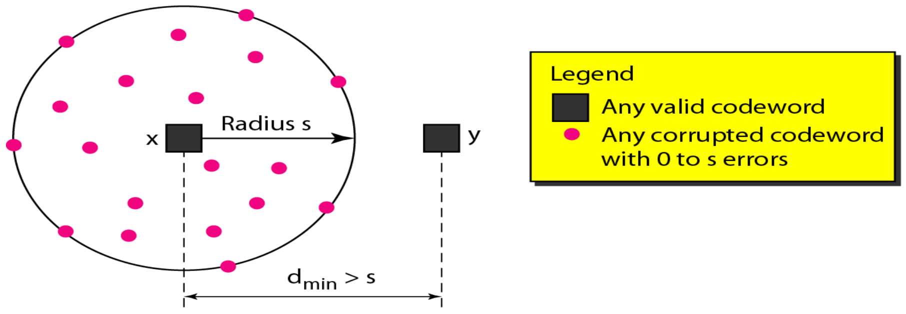
Coding scheme needs to have at least three parameters: the codeword size *n,* the dataword size *k,* and the minimum Hamming distance *dmin.* A coding scheme C is written as *C(n, k)* with a separate expression for *dmin.*
Ex: *C(5,* 2) with dmin = 3.

Hamming Distance and Error

- When a codeword is corrupted during transmission, the Hamming distance between the sent and received codewords is the number of bits affected by the error.
- The Hamming distance between the received codeword and the sent codeword is the number of bits that are corrupted during transmission.
- For example, if the codeword 00000 is sent and 01101 is received, 3 bits are in error and the Hamming distance between the two is d(00000, 01101) =3.

Minimum Distance for Error Detection

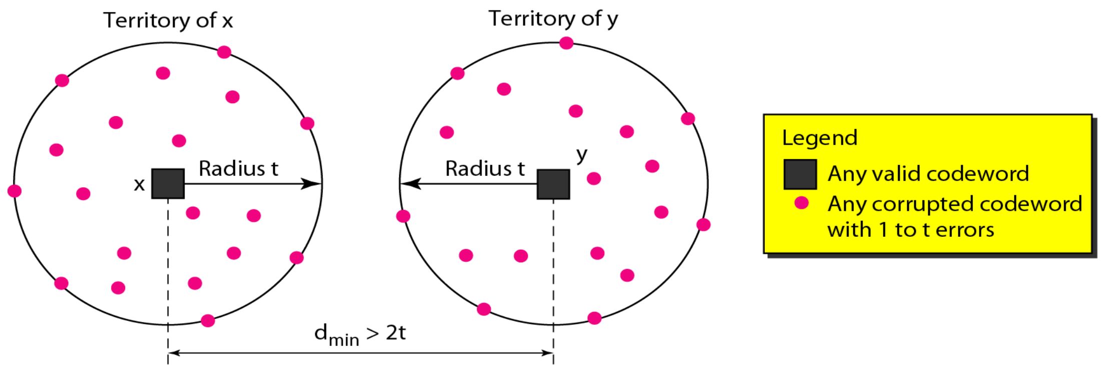
- If S errors occur during transmission, the Hamming distance between the sent codeword and received codeword is S.
- To guarantee the detection of up to S errors in all cases, the minimum Hamming distance in a block code must be dmin =*S* + 1.
- Let us assume that the sent codeword *x* is at the center of a circle with radius *S.* All other received codewords that are created by 1 to *S* errors are points inside the circle or on the perimeter of the circle. All other valid codewords must be outside the circle.



Minimum Distance for Error Correction

- When a received codeword is not a valid codeword, the receiver needs to decide which valid codeword was actually sent. The decision is based on the concept of territory, an exclusive area surrounding the codeword. Each valid codeword has its own territory.

- We use a geometric approach to define each territory. We assume that each valid codeword has a circular territory with a radius of *t* and that the valid codeword is at the center.
- For example, suppose a codeword *x* is corrupted by *t* bits or less. Then this corrupted codeword is located either inside or on the perimeter of this circle. If the receiver receives a codeword that belongs to this territory, it decides that the original codeword is the one at the center.
- To guarantee correction of up to t errors in all cases, the minimum Hamming distance in a block code must be dmin == 2t + 1.



3.3 Linear Block Codes

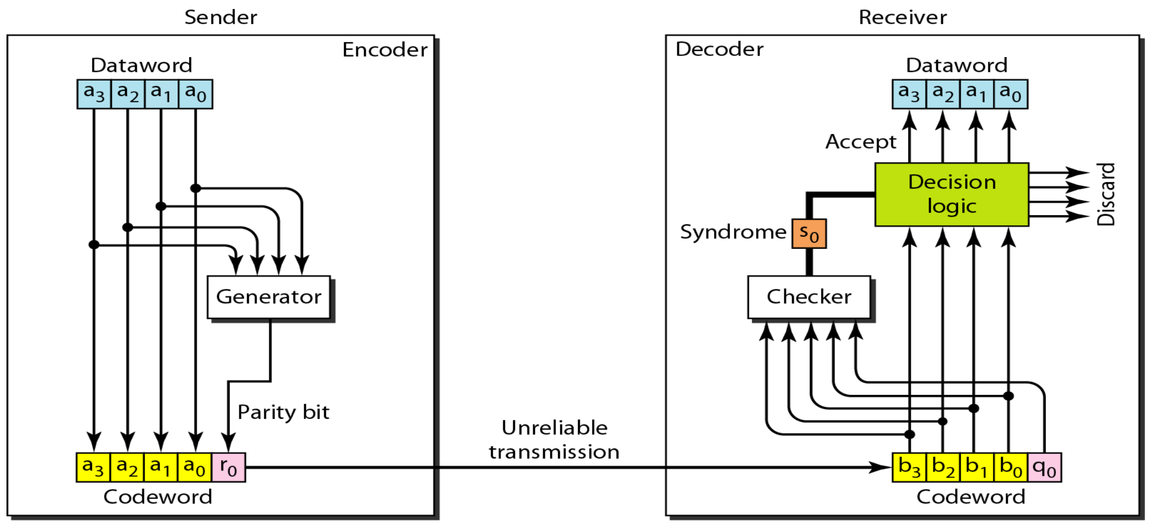
Linear block code is a code in which the exclusive OR of two valid codewords creates another valid codeword.

Minimum Distance for Linear Block Codes: The minimum Hamming distance is the number of 1s in the nonzero valid codeword with the smallest number of 1s.

Some Linear Block Codes

1) <u>Simple Parity-Check Code</u>

- In this code, a *k-bit* dataword is changed to an n-bit codeword where *n* = *k* + 1. The extra bit, called the parity bit, is selected to make the total number of 1s in the codeword even.
- A simple parity-check code is a single-bit error-detecting code in which *n* =*k* + 1 with dmin =2.



- The encoder uses a generator that takes a copy of a 4-bit dataword (a_0, a_1, a_2, and a_3) and generates a parity bit r_0.
- The dataword bits and the parity bit create the 5-bit codeword. The parity bit that is added makes the number of 1s in the codeword even.

  Example: Simple parity-check code C(5, 4)

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

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

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

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

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

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

Example:

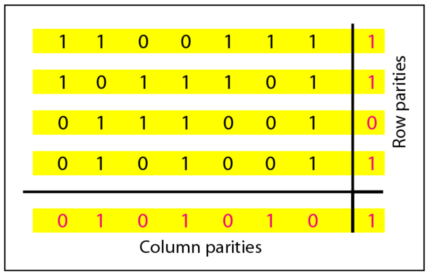
Assume the sender sends the dataword 1011. The codeword created from this dataword is 10111, which is sent to the receiver.

1. No error occurs; the received codeword is 10111. The syndrome is O. 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. Note that although none of the dataword bits are corrupted, no dataword is created because the code is not sophisticated enough to show the position of the corrupted bit.
4. An error changes *ro* and a second error changes 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. The simple parity-check decoder cannot detect an even number of errors. The errors cancel each other out and give the syndrome a value of 0.

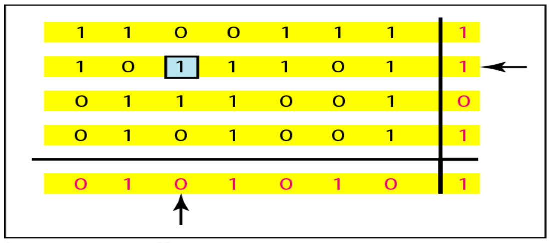
5. Three bits-a_3, a_2, and a_1 are changed by errors. The received codeword is 01011. The syndrome is 1. The dataword is not created. This shows that the simple parity check, guaranteed to detect one single error, can also find any odd number of errors.

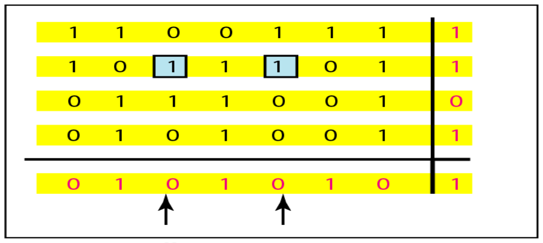
Limitation: A simple parity-check code can detect an odd number of errors.

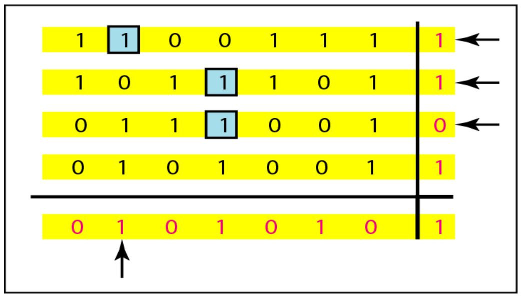
- A better approach is the two-dimensional parity check. In this method, the dataword is organized in a table.
- The data to be sent, five 7-bit bytes, are put in separate rows.
- For each row and each column, 1 parity-check bit is calculated.
- The whole table is then sent to the receiver, which finds the syndrome for each row and each column.
- The two-dimensional parity check can detect up to three errors that occur anywhere in the table. However, errors affecting 4 bits may not be detected.

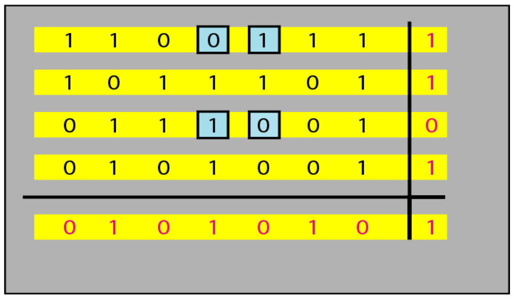


a. Design of row and column parities


b. One error affects two parities


c. Two errors affect two parities


d. Three errors affect four parities


e. Four errors cannot be detected

Hamming Codes

These codes were originally designed with *dmin* = 3, which means that they can detect up to two errors or correct one single error.

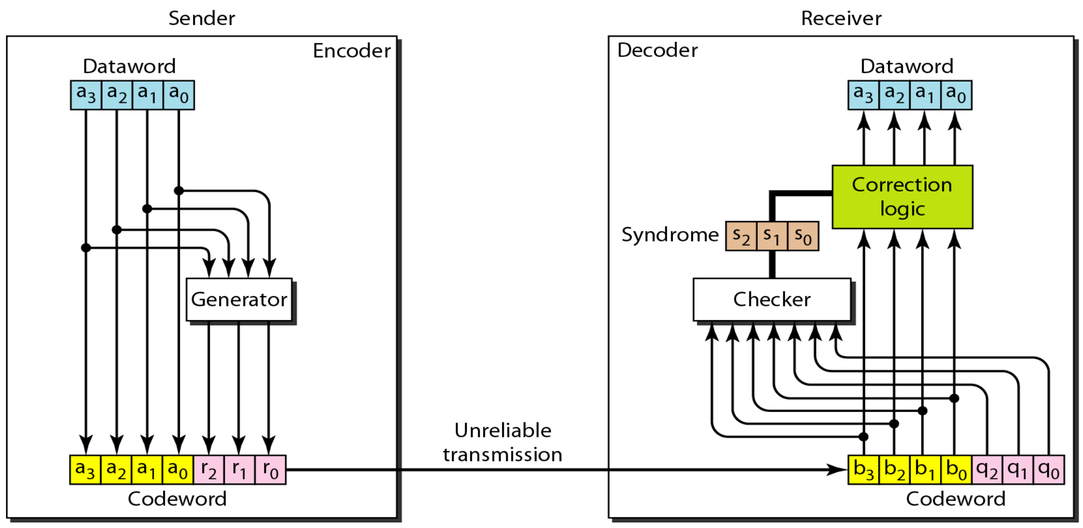
In hamming code we need to choose an integer m, say *m* >= 3. The values of n and *k* are then calculated from m as *n* = *2m* – 1 and *k* = *n* - *m*. The number of check bits *r* =*m.*

Eg: if m = 3, n=7, k = 4

Hamming code C(7, 4) - n=7, k = 4:

Datawords	Codewords	Datawords	Codewords
0000	0000000	1000	1000110
0001	0001101	1001	1001011
0010	0010111	1010	1010001
0011	0011010	1011	1011100
0100	0100011	1100	1100101
0101	0101110	1101	1101000
0110	0110100	1110	1110010
0111	0111001	1111	1111111

Below figure shows the structure of the encoder and decoder:



A copy of a 4-bit dataword is fed into the generator that creates three parity checks.

$\mathbf{r_0 = a_2 + a_1 + a_0}$ modulo-2

$\mathbf{r_1 = a_3 + a_2 + a_1}$ modulo-2

$\mathbf{r_2 = a_1 + a_0 + a_3}$ modulo-2

The checker in the decoder creates a 3-bit syndrome (s2s1s0) in which each bit is the parity check for 4 out of the 7 bits in the received codeword:

$\mathbf{s_0 = b_2 + b_1 + b_0}$ modulo-2

$\mathbf{s_1 = b_3 + b_2 + b_1}$ modulo-2

$\mathbf{s_2 = b_1 + b_0 + b_3}$ modulo-2

The 3-bit syndrome creates eight different bit patterns (000 to 111) that can represent eight different conditions. These conditions define a lack of error or an error in 1 of the 7 bits of the received codeword.

Syndrome	000	001	010	011	100	101	110	111
Error	None	q_0	q_1	b_2	q_2	b_0	b_3	b_1

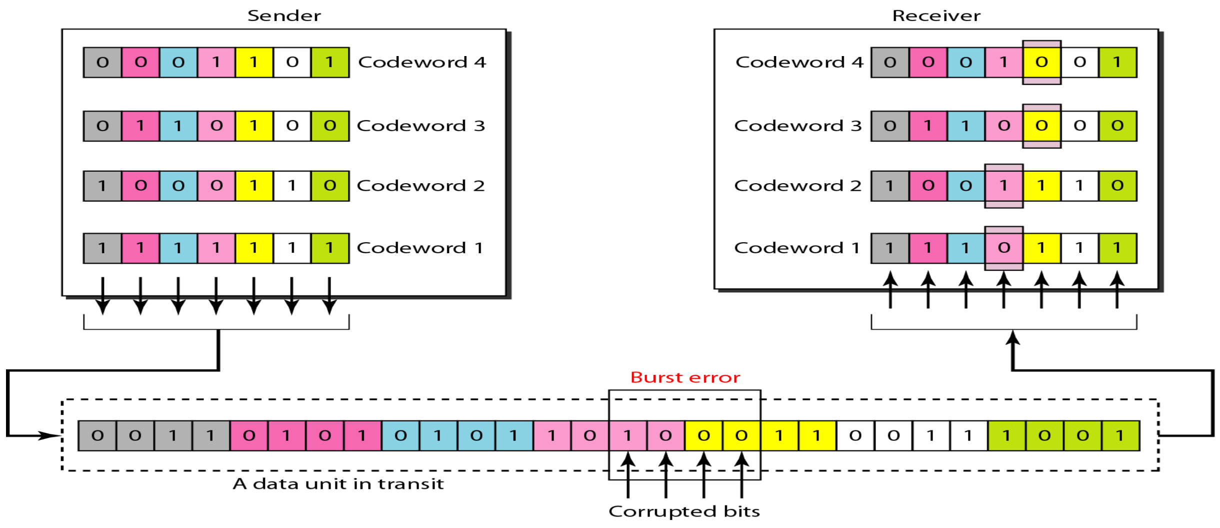
For example, if q_0 is in error, S_0 is the only bit affected; the syndrome, therefore, is 001. If b_2 is in error, S_0 and S_1 are the bits affected; the syndrome therefore is 01l. Similarly, if b_1 is in error, all 3 syndrome bits are affected and the syndrome is 111.

Example:

1. The dataword 0100 becomes the codeword 0100011. The codeword 0100011 is received. The syndrome is 000 (no error), the final dataword is 0100.
2. The dataword 0111 becomes the codeword 0111001. The codeword 0011001 is received. The syndrome is 011. Therefore b_2 is in error. After flipping b_2 (changing the 1 to 0), the final dataword is 0111.
3. The dataword 1101 becomes the codeword 1101000. The codeword 0001000 is received (two errors). The syndrome is 101, which means that b_0 is in error. After flipping b_0, we get 0000, the wrong dataword. This shows that our code cannot correct two errors.

Performance

A Hamming code can only correct a single error or detect a double error. However, there is a way to make it detect a burst error.



The key is to split a burst error between several codewords, one error for each codeword.

To make the Hamming code respond to a burst error of size *N*, we need to make *N* codewords out of our frame. Then, instead of sending one codeword at a time, we arrange the codewords in a table and send the bits in the table a column at a time.

In the above Figure, the bits are sent column by column (from the left). In each column, the bits are sent from the bottom to the top. In this way, a frame is made out of the four codewords and sent to the receiver. It is shown in the figure that when a burst error of size 4 corrupts the frame, only 1 bit from each codeword is corrupted. The corrupted bit in each codeword can then easily be corrected at the receiver.

3.4 Cyclic Codes

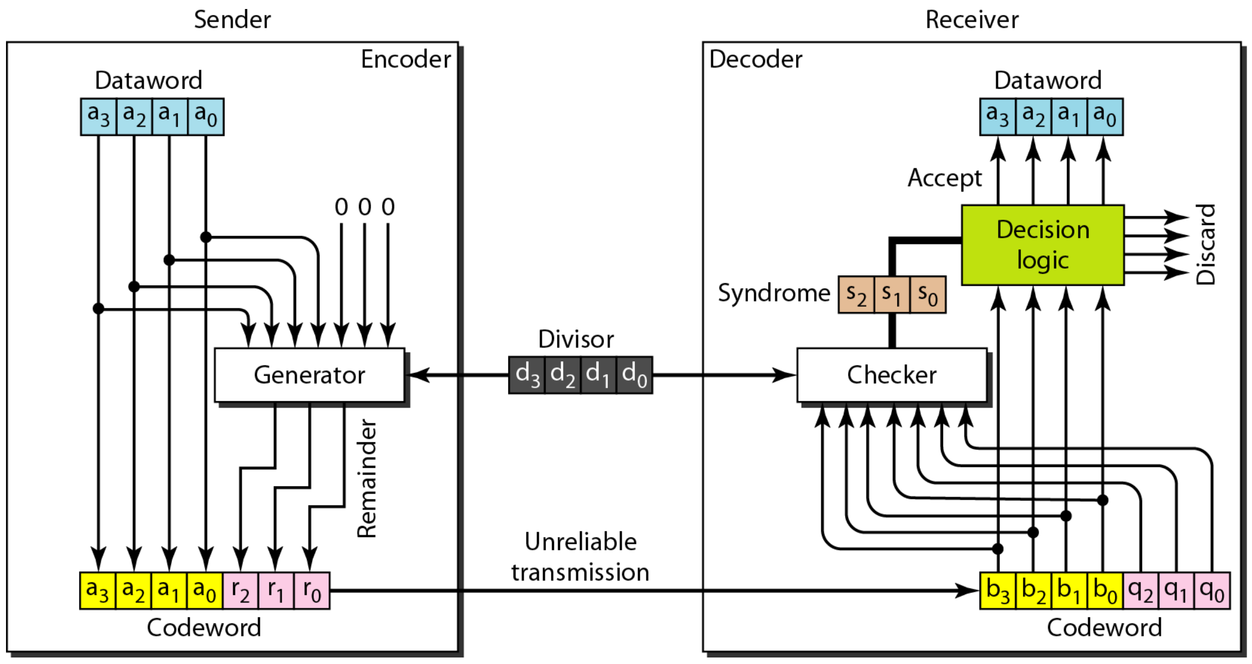
Cyclic codes are special linear block codes in which, if a codeword is cyclically shifted (rotated), the result is another codeword.

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_0 to b_6, we can shift the bits by using the following:

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

Cyclic Redundancy Check



Below Table shows an example of a CRC code.

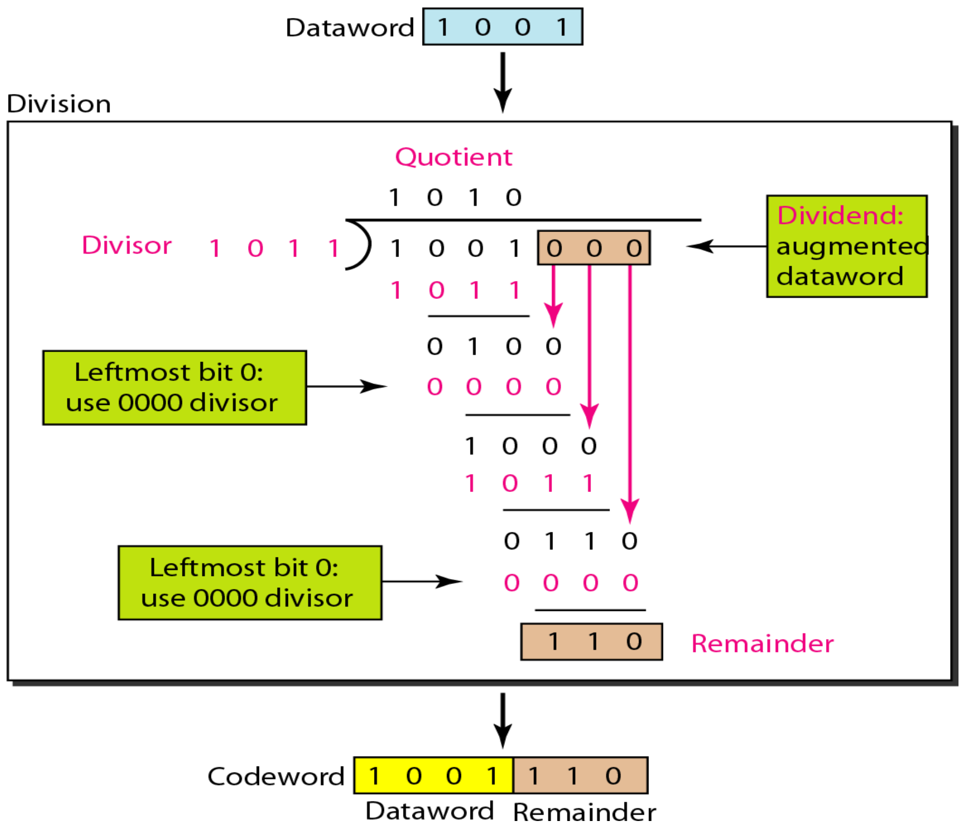
Dataword	*Codeword*	*Dataword*	*Codeword*
0000	0000000	1000	1000101
0001	0001011	1001	1001110
0010	0010110	1010	1010011
0011	0011101	1011	1011000
0100	0100111	1100	1100010
0101	0101100	1101	1101001
0110	0110001	1110	1110100
0111	0111010	1111	1111111

- 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 augmented dataword by the divisor (modulo-2 division).
- The quotient of the division is discarded; the remainder is appended to the dataword to create the codeword.
- The decoder receives the possibly corrupted codeword. A copy of all n bits is fed to the checker which is a replica of the generator.
- The remainder produced by the checker is a syndrome of *n* - *k* (3 here) bits, which is fed to the decision logic analyzer.
- The analyzer has a simple function. If the syndrome bits are all as, the 4 leftmost bits of the codeword are accepted as the dataword (interpreted as no error); otherwise, the 4 bits are discarded (error).

Encoder

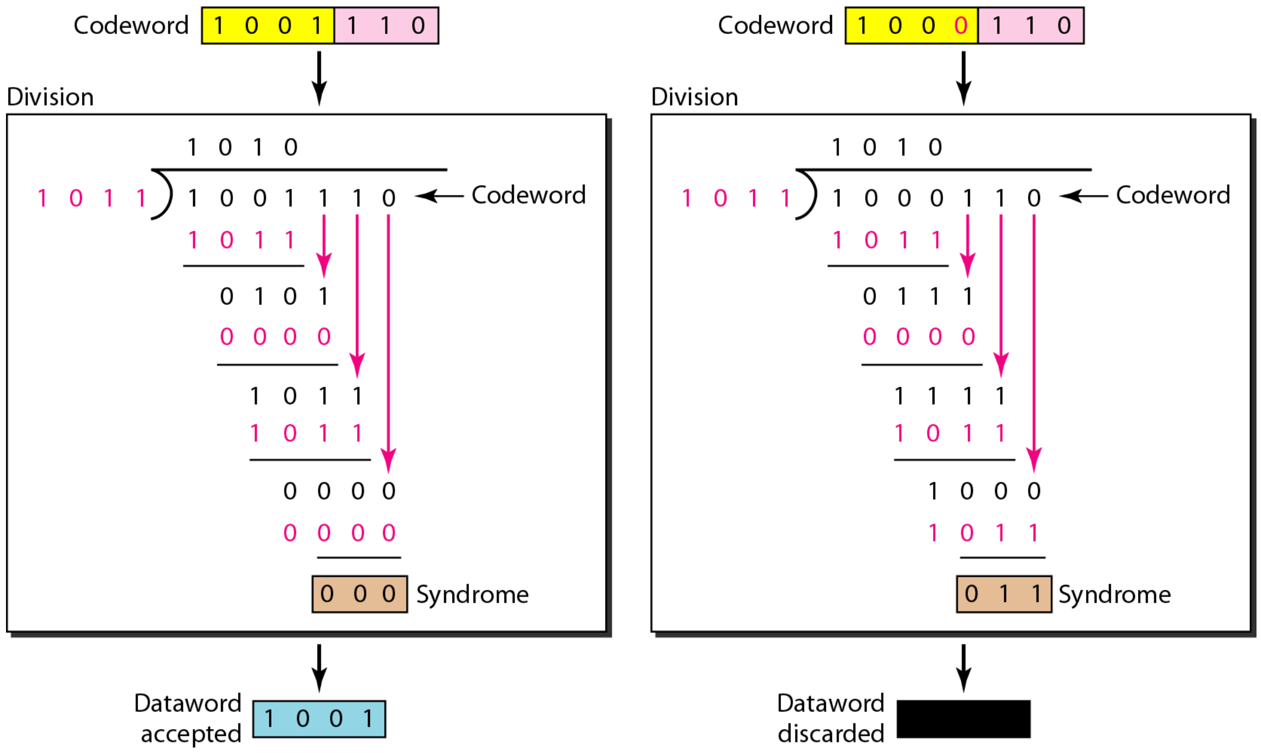
The encoder takes the dataword and augments it with n - k number of 0s. It then divides the augmented dataword by the divisor.



Decoder

The codeword can change during transmission. The decoder does the same division process as the encoder. The remainder of the division is the syndrome. If the syndrome is all 0s, there is no error; the dataword is separated from the received codeword and accepted. Otherwise, everything is discarded.

The left hand figure shows the value of syndrome when no error has occurred; the syndrome is 000. The right-hand part of the figure shows the case in which there is one single error. The syndrome is not all 0s



Hardware Implementation

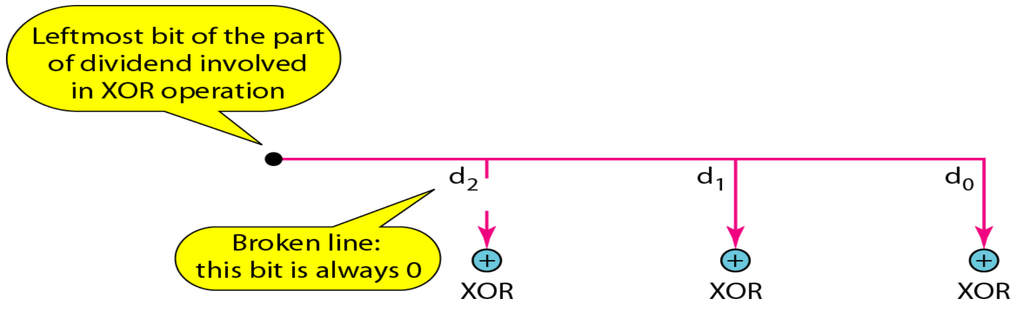
One of the advantages of a cyclic code is that the encoder and decoder can easily and cheaply be implemented in hardware by using a handful of electronic devices. Also, a hardware implementation increases the rate of check bit and syndrome bit calculation.

Divisor:

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 Os. In other words, the bits do not change from one dataword to another. In 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.

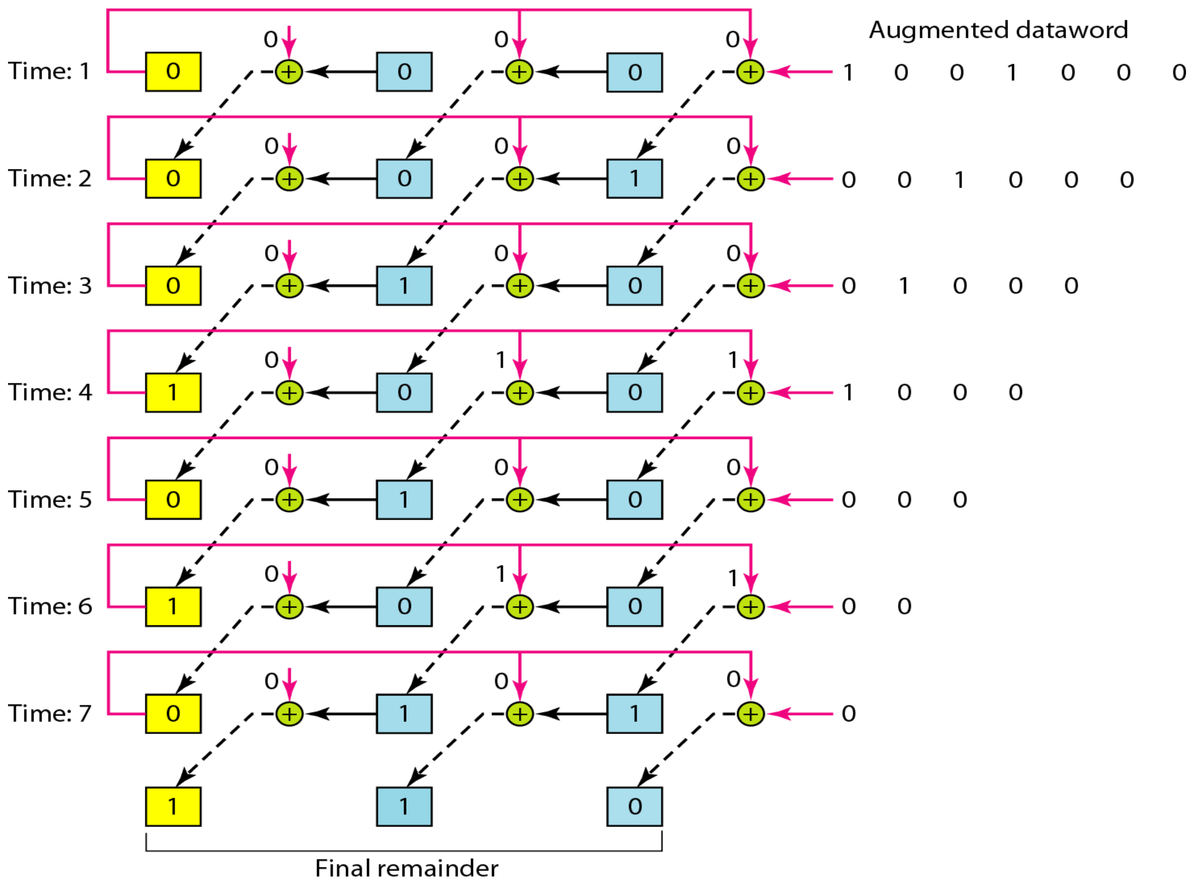
3. A close look shows that only *n* - *k* bits of the divisor is needed in the XOR operation. The leftmost bit is not needed because the result of the operation is always 0, no matter what the value of this bit. The reason is that the inputs to this XOR operation are either both 0s or both 1s.


Leftmost bit of the part of dividend involved in XOR operation
d_2
d_1
d_0
Broken line: this bit is always 0
XOR
XOR
XOR

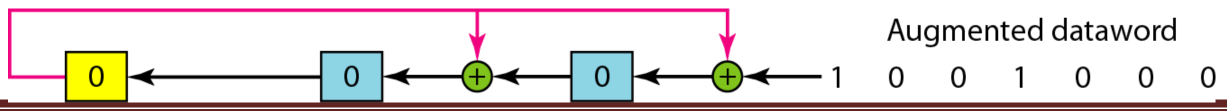
Steps:

1. Assume that the remainder is originally all Os (000 in our example).
2. At each time click (arrival of 1 bit from an augmented dataword), repeat the following two actions:
   a. 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.

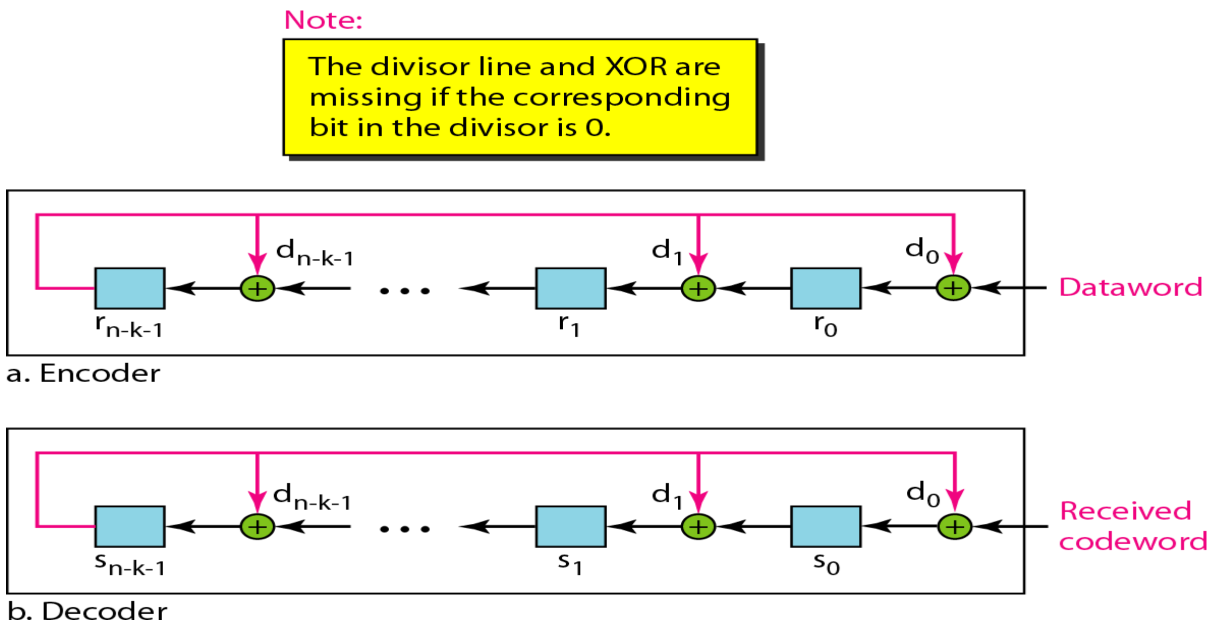
Below Figure shows this simulator, but note that this is not the final design; there will be more improvements.



- At each clock tick, shown as different times, one of the bits from the augmented dataword is used in the XOR process.
- The above design is for demonstration purposes only.
- It needs simplification to be practical.
- First, we do not need to keep the intermediate values of the remainder bits; we need only the final bits. We therefore need only 3 registers instead of 24.
- After the XOR operations, we do not need the bit values of the previous remainder.
- Also, we do not need 21 XOR devices; two are enough because the output of an XOR operation in which one of the bits is 0 is simply the value of the other bit.
- This other bit can be used as the output.
- With these two modifications, the design becomes tremendously simpler and less expensive, as shown below

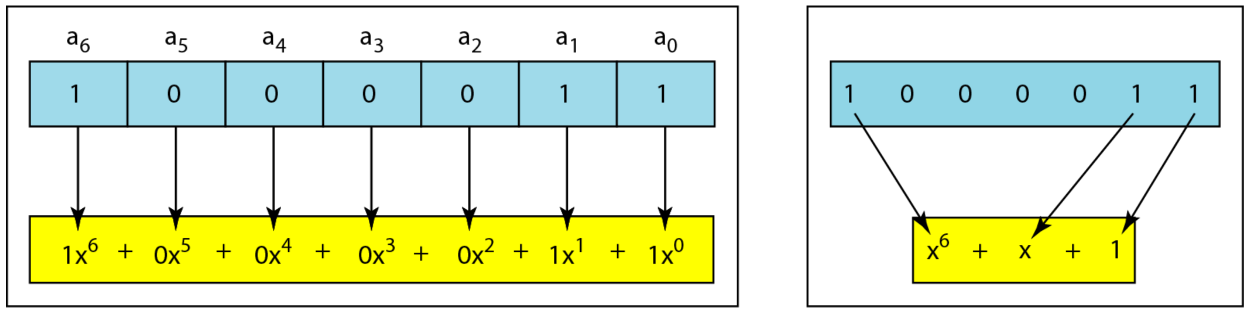


General Design



<u>Polynomials</u>

A pattern of Os and 1s can be represented as a **polynomial** with coefficients of 0 and 1. The power of each term shows the position of the bit; the coefficient shows the value of the bit. Figure shows a binary pattern and its polynomial representation.



a_6 a_5 a_4 a_3 a_2 a_1 a_0
1 0 0 0 0 1 1
$1x^6 + 0x^5 + 0x^4 + 0x^3 + 0x^2 + 1x^1 + 1x^0$
1 0 0 0 0 1 1
$x^6 + x + 1$

a. Binary pattern and polynomial b. Short form

Degree of a Polynomial

The degree of a polynomial is the highest power in the polynomial.

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

Adding and Subtracting Polynomials

- Adding and subtracting polynomials in mathematics are done by adding or subtracting the coefficients of terms with the same power. In our case, the coefficients are only 0 and 1, and adding is in modulo-2.
- This has two consequences.
- First, addition and subtraction are the same.
- Second, adding or subtracting is done by combining terms and deleting pairs of identical terms. For example, adding *x5* + *x4* + *x2* and *x6* + *x4* + *x2* gives just *x6* + *x5*.
- The terms *x4* and *x2* are deleted. However, note that if we add, for example, three polynomials and we get *x2* 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, x3 x x4 is x7 , For dividing, we just subtract the power of the second term from the power of the first.

Multiplying Two Polynomials

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

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

Dividing One Polynomial by Another

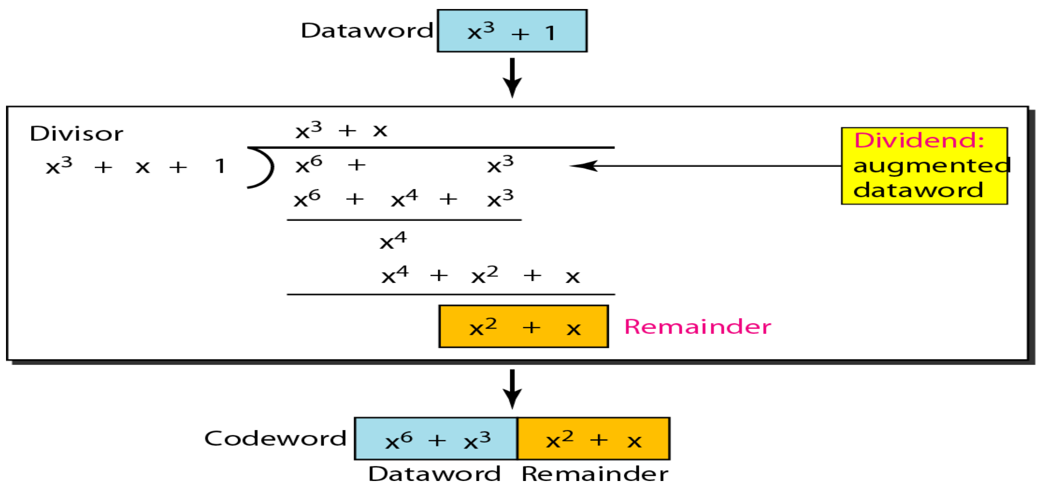
We divide the first term of the dividend by the first term of the divisor to get the first term of the quotient. We multiply the term in the quotient by the divisor and subtract the result from the dividend. We repeat the process until the dividend degree is less than the divisor degree.

Shifting

A binary pattern is often shifted a number of bits to the right or left. Shifting to the left means adding extra Os as rightmost bits; shifting to the right means deleting some rightmost bits. Shifting to the left is accomplished by multiplying each term of the polynomial by xn, where m

is the number of shifted bits; shifting to the right is accomplished by dividing each term of the polynomial by x^n. The following shows shifting to the left and to the right. Note that we do not have negative powers in the polynomial representation.

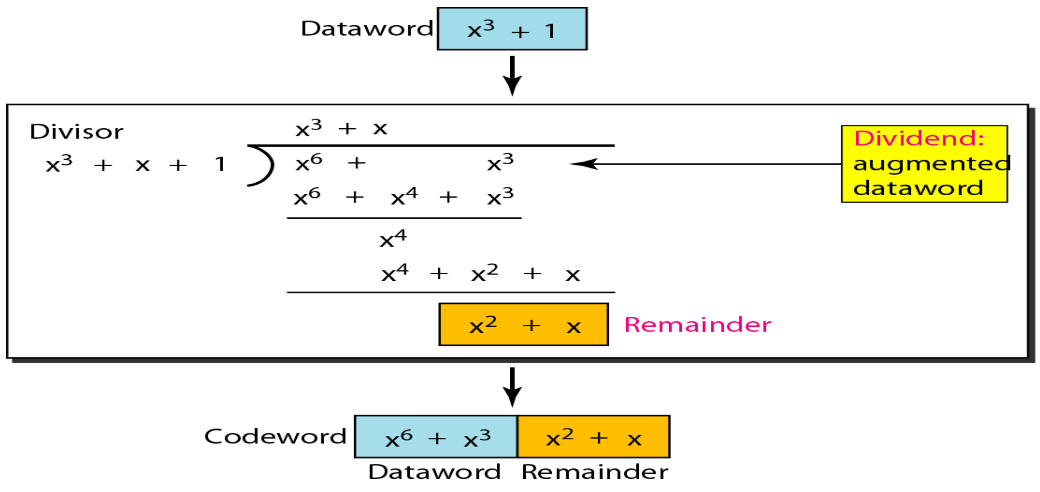
Shifting left 3 bits:	10011 becomes 10011000	$x^4 + x + 1$ becomes $x^7 + x^4 + x^3$
Shifting right 3 bits:	10011 becomes 10	$x^4 + x + 1$ becomes x



is the number of shifted bits; shifting to the right is accomplished by dividing each term of the polynomial by x^n. The followinCyclic Code Analysis

Following notations can be used in the cyclic codes:

Shifting left 3 bits:	10011 becomes 10011000	$x^4 + x + 1$ becomes $x^7 + x^4 + x^3$
Shifting right 3 bits:	10011 becomes 10	$x^4 + x + 1$ becomes x



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

Error: *e(x)* Generator: *g(x)*

In a cyclic code,

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

The received codeword is the sum of the sent codeword and the error.

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

The receiver divides the received codeword by g(x) to get the syndrg shows shifting to the left and to the right. Note that we do not have negative powers in the polynomial representation.

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

The Right hand side of above equation is called as syndrome.

If Syndrome does not have a remainder (syndrome =0), either *e(x)* is 0 or *e(x)* is divisible by *g(x).*

In a cyclic code, those *e(x)* errors that are divisible by *g(x)* are not caught.

Single-Bit Error

A single-bit error is $e(x) = x^i$, where *i* is the position of the bit. If a single-bit error is caught, then x^i is not divisible by *g(x).*

If the generator has more than one term and the coefficient of x^0 is 1, all single errors can be caught.

Example:

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

a. $x + 1$

b. $x3$

c. 1

Solution:

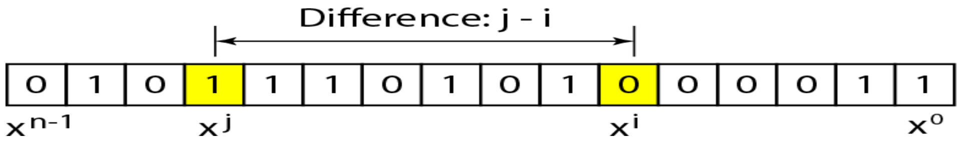
a. No x^i can be divisible by $x + 1$. In other words, $x^i/(x + 1)$ always has a remainder. So the syndrome is nonzero. Any single-bit error can be caught.

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

c. All values of *i* make *j* divisible by *g(x).* No single-bit error can be caught

Two Isolated Single-Bit Errors

Two isolated single bit errors can be represented as $e(x) = x^j + x^i$. The values of i and j define the positions of the errors, and the difference j - i defines the distance between the two errors.



We can write $e(x) = x^i (x^{j-i} + 1) = x^i (x^t + 1)$.

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

Example:

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

a. $x + 1$

b. $x^4 + I$

c. $x^7 + x^6 + 1$

d. $x^{15} + x^{14} + 1$

Solution:

a. This is a very poor choice for a generator. Any two errors next to each other cannot be detected.

b. This generator cannot detect two errors that are four positions apart. The two errors can be anywhere, but if their distance is 4, they remain undetected.

c. This is a good choice for this purpose.

d. This polynomial cannot divide any error of *type* $x^t + 1$ if t is less than 32,768. This means that a codeword with two isolated errors that are next to each other or up to 32,768 bits apart can be detected by this generator.

Odd Numbers of Errors

A generator that contains a factor of x + 1 can detect all odd-numbered errors.

Burst Errors

If L is the burst size and r is the degree of generator polynomial.

- All burst errors with L <= 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 $1- (1/2)^r$

Example:

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 I 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 $x + 1$.

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
CRC-16	$x^{16} + x^{12} + x^5 + 1$	HDLC
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	LANs

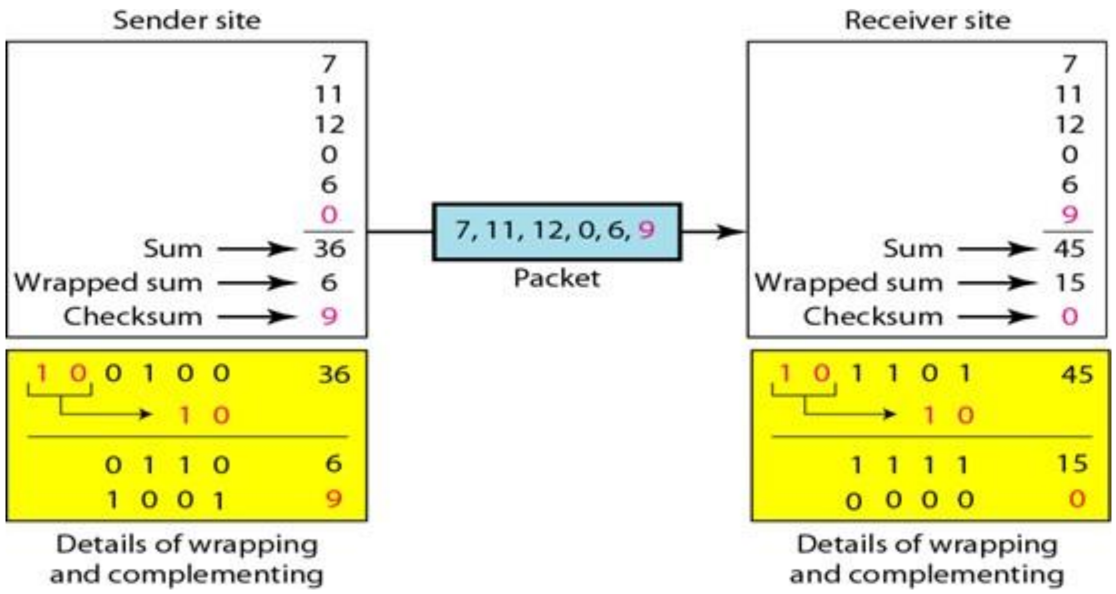
Advantages of Cyclic Codes

- Cyclic codes have a very good performance in detecting single-bit errors, double errors, an odd number of errors, and burst errors.

- They can easily be implemented in hardware and software.
- They are especially fast when implemented in hardware.

3.5 Checksum

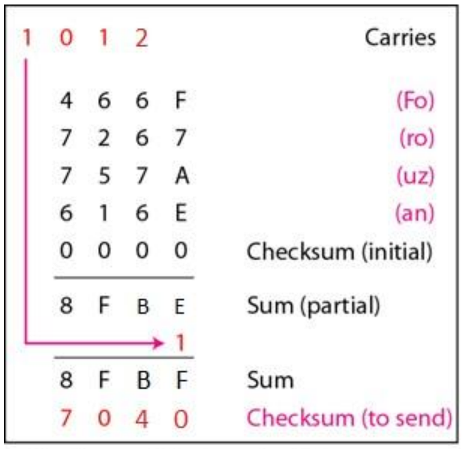
- The checksum is used in the Internet by several protocols. The checksum is based on the conceptof redundancy.
- Below Figure shows the process at the sender and at the receiver.
- The sender initializes the checksum to 0 and adds all data items and the checksum (the checksum is considered as one dataitem 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.
- The receiver follows the same procedure as the sender.
- It adds all data items (including the checksum); the result is 45.



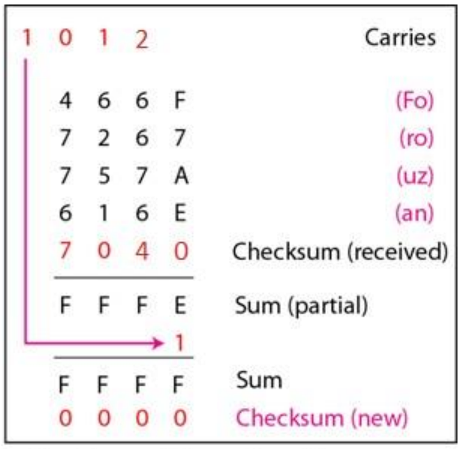
- The sum is wrapped and becomes 15. The wrapped sum is complemented and becomes 0.
- Since the value of the checksum is 0, this means that the data is not corrupted. The receiver drops the checksum and keeps the other data items.

- If the checksum is not zero, the entire packet is dropped.

Internet Checksum



a. Checksum at the sender site



a. Checksum at the receiver site

Traditionally, the Internet has been using a 16-bit checksum.

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.

Performance

- The traditional checksum uses a small number of bits (16) to detect errors in a message of any size (sometimes thousands of bits).
- However, it is not as strong as the CRC in error-checking capability.
- For example, if the value of one word is incremented and the value of another word is decremented by the same amount, the two errors cannot be detected because the sum and checksum remain the same.
- Also if the values of several words are incremented but the total change is a multiple of 65535, the sum and the checksum does not change, which means the errors are not detected.