UNIT – II

Representation of Characters, Integers and Fractions-Octal and Hexadecimal number systems-Signed-Fixed and floating point number representations-BCD Code-Gray Code-ASCII Code.

Representation of Characters, Integers and Fractions:

Representation of Characters:

- Everything represented by a computer is represented by binary sequences.
- A common non-integer needed to be represented is characters.
- We use standard encodings (binary sequences) to represent characters.

A standard code ASCII (American Standard for Computer Information Interchange) defines what character is represented by each sequence. Characters must somehow be represented in the computer using the 1's and 0's that the hardware works with. How they are represented is arbitrary, but to make sure that characters used on one computer are understood correctly by another, there are *standards* defined.

The most common standard is called **ASCII** (American Standard Code for Information Interchange).

In ASCII, there are 128 characters to be represented, so each character is stored in one byte. Here is one representation of the ASCII character set

We can divide the character set into different classes:

- **Alphabetic** characters including:
  - lower case letters - 'a'..'z'
  - upper case letters - 'A'..'Z'
- **Digit** characters - '0'..'9'
- **Punctuation** characters - '.',';',',', etc.
- **White space** characters - ' ','\n','\t', etc.

- **Special** characters - '@',' < ', ' >', etc.
- **Control** characters - '^A','^B', etc.

Each character has a *numeric representation*. How can we find that out?

```
printf("The numeric representation of %c is %d\n",'a','a');
```

The characters whose ASCII values are between 32 and 126 are said to be **printable characters**. The others have special meanings, for example, '^J' is the newline ('\n') character in Unix, '^I' is the tab ('\t') character.

Examples:

0100 0001 is 41 (hex) or 65 (decimal). It represents `A'

0100 0010 is 42 (hex) or 66 (decimal). It represents `B'

Different bit patterns are used for each different character that needs to be represented. The code has some nice properties. If the bit patterns are compared, (pretending they represent integers), then `A' < `B'

This is good, because it helps with sorting things into alphabetical order.

Notes: `a' (61 hex) is different than `A' (41 hex)

`8' (38 hex) is different than the integer 8

the digits:

`0' is 48 (decimal) or 30 (hex)

`9' is 57 (decimal) or 39 (hex)

Because of this, code must deal with both characters and integers correctly.

Representation of Integers:

Integers are *whole numbers* or *fixed-point numbers* with the radix point *fixed* after the least-significant bit. They are contrast to *real numbers* or *floating-point numbers*, where the position of the radix point varies. It is important to take note that integers and floating-point numbers are treated differently in computers. They have different representation and are

processed differently (e.g., floating-point numbers are processed in a so-called floating-point processor).

Computers use *a fixed number of bits* to represent an integer. The commonly-used bit-lengths for integers are 8-bit, 16-bit, 32-bit or 64-bit. The term integer is used in computer engineering/science to refer to a data type which represents some finite subset of the mathematical integers. Integers may be unsigned or signed.

Common integer data types and ranges.

Bits	Name
8	Byte, Octal
16	Half word, Word
32	Word, Double word, Long word
64	Double word, longword, long long, quad, quadword
128	Octal word
n	*n*-bit integer (general case)

1. *Unsigned Integers*: can represent zero and positive integers.
2. *Signed Integers*: can represent zero, positive and negative integers. Three representation schemes had been proposed for signed integers:
    a. Sign-Magnitude representation
    b. 1's Complement representation
    c. 2's Complement representation

You, as the programmer, need to decide on the bit-length and representation scheme for your integers, depending on your application's requirements. Suppose that you need a counter for counting a small quantity from 0 up to 200, you might choose the 8-bit unsigned integer scheme as there is no negative numbers involved.

***I) n*-bit Unsigned Integers**

Unsigned integers can represent zero and positive integers, but not negative integers. The value of an unsigned integer is interpreted as "*the magnitude of its underlying binary pattern*".

Example 1: Suppose that *n*=8and the binary pattern is 0100 0001B, the value of this unsigned integer is 1×2^0 + 1×2^6 = 65D.

Example 2: Suppose that *n*=16and the binary pattern is 0001 0000 0000 1000B, the value of this unsigned integer is 1×2^3 + 1×2^12 = 4104D.

An *n*-bit pattern can represent 2^*n* distinct integers. An *n*-bit unsigned integer can represent integers from 0 to (2^*n*)-1, as tabulated below:

ii) Signed Integers

Signed integers can represent zero, positive integers, as well as negative integers. Three representation schemes are available for signed integers:

1. Sign-Magnitude representation
2. 1's Complement representation
3. 2's Complement representation

In all the above three schemes, the *most-significant bit* (msb) is called the *sign bit*. The sign bit is used to represent the *sign* of the integer - with 0 for positive integers and 1 for negative integers. The *magnitude* of the integer, however, is interpreted differently in different schemes.

iii) n-bit Sign Integers in Sign-Magnitude Representation

In sign-magnitude representation:

- The most-significant bit (msb) is the *sign bit*, with value of 0 representing positive integer and 1 representing negative integer.
- The remaining *n*-1 bits represents the magnitude (absolute value) of the integer. The absolute value of the integer is interpreted as "the magnitude of the (*n*-1)-bit binary pattern".

Example 1: Suppose that *n*=8 and the binary representation is 0 100 0001B.
Sign bit is 0 ⇒ positive
Absolute value is 100 0001B = 65D
Hence, the integer is +65D

The drawbacks of sign-magnitude representation are:

1. There are two representations (0000 0000B and 1000 0000B) for the number zero, which could lead to inefficiency and confusion.
2. Positive and negative integers need to be processed separately.

iv) Computers use 2's Complement Representation for Signed Integers

We have discussed three representations for signed integers: signed-magnitude, 1's complement and 2's complement. Computers use 2's complement in representing signed integers. This is because:

1. There is only one representation for the number zero in 2's complement, instead of two representations in sign-magnitude and 1's complement.
2. Positive and negative integers can be treated together in addition and subtraction. Subtraction can be carried out using the "addition logic".

REPRESENTATION OF FRACTIONS:

Use a “binary point” to separate positive from negative powers of two -- just like “decimal point.” ! 2’s comp addition and subtraction still work • only if binary points are aligned 00101000.101!"#$%&'(+ 11111110.110!)*$&'(00100111.011!+,$+-'(

Thus you can pick some part of a 32-bit computer word and decide where the binary point should be. Let's say we pick the middle of the word somewhere. This gives 1 bit of sign, 15 bits of integer, and 16 bits of fraction. Thus one can have numbers from about -32,767 to +32,767 with about 4½ decimal digits of fractional precision.

If you use this technique you must convert numbers from the bit representation to decimal representation correctly. Pascal and C do not provide this flexibility in conversion, but some other programming languages do. PL/I for example allows you to declare numeric variables with the number of bits and where the binary point should be located. PL/I then builds code to do the shifting and adjusting for you, and its input/output routines also work properly. However one can always write conversion subroutines in other languages that do work properly so that you can both input fractional quantities and output them properly.

Note that the rules for binary arithmetic on these fractional numbers are the same as it is for integers, so it requires no change in the hardware to deal with binary fractions. That is this fractional representation is isomorphic to the binary twos-complement integer representation that the machine uses. It only requires that you think the numbers are fractions and that you understand and treat properly the binary-point (the binary equivalent of the decimal point). For example you can only meaningfully add two numbers if the binary point is aligned. Of course, the computer will add them no matter where you think the binary point is. This is just like decimal arithmetic where you must add two numbers with the decimal points aligned if you want the right answer.

One can shift the binary point of a number by multiplying or dividing by the proper power of two, just as one shifts the decimal point by multiplying or dividing by a power of ten. Most machines also provide arithmetic shift operations that shift the bit representation of the number right or left more quickly than a multiply instruction would. An arithmetic right shift of 3 is a binary division by 2^3 or 8; a left shift is a binary multiplication. Arithmetic right shifts usually copy the sign bit so that negative numbers stay negative. Arithmetic left shifts introduce zeros from the right side of the number. If the sign bit changes in an arithmetic left shift then the number has overflowed.

One is not restricted to scaling in the binary representation, one can also scale in the decimal representation. The rules for decimal scaling are the same as in the table above. For example if you read in an integer but you know that it contains a number with two `implied' decimal digits, i.e. the number 500 represents 5 with a q=2. You treat it that way in the machine, multiply it by 2.1 (21 with a q=1), the result is (10500, with q=3). Now you can correctly output the result with the decimal point properly placed because you know the q=3, 10.500. Again you must be careful to only add and subtract numbers with the same decimal scale, and carefully scale the numbers around multiplication and division to preserve both the high-order digits while maintaining the required precision after the decimal point.

Octal and Hexadecimal number systems

OCTAL

An older computer-based number system is "octal", or base eight. The digits in octal math are 0, 1, 2, 3, 4, 5, 6, and 7. The value "eight" is written as "1 eight and 0 ones", or 10_8.

- Base or radix 8 number system.
- 1 octal digit is equivalent to 3 bits.
- Octal numbers are 0 to7. (see the chart down below)
- Numbers are expressed as powers of 8.

Conversion of octal to decimal

(Base 8 to base 10)

Example: convert (632)8 to decimal

= (6 x 82) + (3 x 81) + (2 x 80)

= (6 x 64) + (3 x 8) + (2 x 1)

= 384 + 24 + 2

= (410)10

Conversion of decimal to octal (base 10 to base 8)

Example: convert (177)10 to octal

177 / 8 = 22 remainder is 1

22 / 8 = 2 remainder is 6

2 / 8 = 0 remainder is 2

Answer = 2 6 1

The answer is read from bottom to top as (261)8, the same as with the binary case.

Conversion of decimal fraction to octal fraction is carried out in the same manner as decimal to binary except that now the multiplication is carried out by 8.

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

HEXADECIMAL

In base ten, we had digits 0 through 9. In base eight, we had digits 0 through 7. In base 4, we had digits 0 through 3. In any base system, you will have digits 0 through one-less-than-your-base. This means that, in hexadecimal, we need to have "digits" 0 through 15. To do this, we would need single solitary digits that stand for the values of "ten", "eleven", "twelve", "thirteen", "fourteen", and "fifteen". But we don't. So, instead, we use letters. That is, counting in hexadecimal, the sixteen "numerals" are:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

In other words, A is "ten" in "regular" numbers, B is "eleven", C is "twelve", D is "thirteen", E is "fourteen", and "F" is fifteen. It is this use of letters for digits that makes hexadecimal numbers look so odd at first. But the conversions work in the usual manner.

- Base or radix 16 number system.
- 1 hex digit is equivalent to 4 bits.
- Numbers are 0,1,2…..8,9, A, B, C, D, E, F.
- B is 11, E is 14
- Numbers are expressed as powers of 16.
- 160 = 1, 161 = 16, 162 = 256, 163 = 4096, 164 = 65536, …

Conversion of hex to decimal (base 16 to base 10)

Example: convert (F4C)16 to decimal

= (F x 162) + (4 x 161) + (C x 160)

= (15 x 256) + (4 x 16) + (12 x 1)

Conversion of decimal to hex (base 10 to base 16)

Example: convert (4768)10 to hex.

= 4768 / 16 = 298 remainder 0

= 298 / 16 = 18 remainder 10 (A)

= 18 / 16 = 1 remainder 2

= 1 / 16 = 0 remainder 1

Answer: 1 2 A 0

The answer is read from bottom to top , same as with the binary case.

= 3840 + 64 + 12 + 0

= (3916)10

Conversion of binary to octal and hex

•Conversion of binary numbers to octal and hex simply requires grouping bits in the binary numbers into groups of three bits for conversion to octal and into groups of four bits for conversion to hex.

•Groups are formed beginning with the LSB and progressing to the MSB.

•Thus, 11 100 1112 = 3478

•11 100 010 101 010 010 0012 = 30252218

•1110 01112 = E716

•1 1000 1010 1000 01112 = 18A8716

Signed Number

Our standard way of writing a decimal numeral representing a negative number is to place a minus sign in front of its digits. For example, we read −53 as "negative fifty-three". We typically write positive fifty-three as 53, with no sign, but if we want to emphasize that it is positive, we could write it as +53. The point is that every decimal numeral begins, either implicitly or explicitly, with a symbol indicating its sign, which is followed by a sequence of digits that represent its magnitude (i.e., a "distance" from zero). We could reasonably call this the sign-magnitude representation scheme.

As there are two signs, + and −, a very natural way to incorporate the notion of a sign in a binary numeral is to use a single bit to encode it. For example, we could encode + by 0 and − by 1. If we further decide to place the sign first (i.e., use the first bit to encode the sign), then, for example, the binary numeral 110110 would represent -22. (The 1 in the first bit indicates that the number is negative; the other three 1's are in the 16, 4, and 2 columns, and so yield a magnitude of 22.)

The sign-magnitude approach may be the most natural for humans, but it turns out that an alternative scheme, called two's complement, is what most computers use. Under this scheme, the weight (or place value) of the most significant bit is negative.

For example, suppose we have an 8-bit numeral. Then the column weights areas usual, except that the weight associated to the leftmost column is -(27) rather than +(27). Hence, the binary numeral 11001001 represents

$(1 \times -128) + (1 \times 64) + (1 \times 8) + (1 \times 1) = -55$

- **Unsigned Versus Signed Numbers**

1 If the 4-bit binary value 1101 is unsigned, then it represents the decimal value 13, but as a signed two's complement number, it represents -3.

2 C programming language has **int** and **unsigned int** as possible types for integer variables.

3 If we are using 4-bit unsigned binary numbers and we add 1 to 1111, we get 0000 ("return to zero").

4 If we add 1 to the largest positive 4-bit two's complement number 0111 (+7), we get 1000 (-8).

FIXED AND FLOATING POINT REPRESENTATION

A real number or floating point number has integer part and fractional part separated by a decimal. It is either positive or negative. e.g. 0.345, -121.37 etc

One method of representing real numbers would be to assume a fixed position for the decimal point. e.g. in a 8-bit fixed point representation, where 1 bit is used for sign (+ve or – ve) and 5 bits are used for integral part and two bits are used for fractional part.

- Positive integers and zero can be represented by unsigned numbers
- Negative numbers must be represented by signed numbers since + and – signs are not available, only 1's and 0's are
- Signed numbers have msb as 0 for positive and 1 for negative – msb is the sign bit

Fixed point position usually uses one of the two following positions

- A binary point in the extreme left of the register to make it a fraction
- A binary point in the extreme right of the register to make it an integer
- In both cases, a binary point is not actually present

Representation of fixed point in memory

- *Represents binary number +11100.11*
- *Largest positive number which can be stored 11111.11*
- *Smallest positive number which can be stored 00000.01*
- *This range is quite inadequate even for simple arithmetic calculations. To increase the range we use floating point representation.*
- *When an integer is positive, the msb, or sign bit, is 0 and the remaining bitsrepresent the magnitude*
- *When an integer is negative, the msb, or sign bit, is 1, but the rest of the number can be represented in one of three ways*
- ***Signed-magnitude representation***
- ***Signed-1's complement representation***
- ***Signed-2's complement representation***
- *Consider an 8-bit register and the number +14*

The only way to represent it is 00001110

- *Consider an 8-bit register and the number –14*
- *Signed magnitude: 1 0001110*
- *Signed 1's complement: 1 1110001*
- *Signed 2's complement: 1 1110010*

Floating Point Representation:

A floating point number consists of two parts:

- An EXPONENT (also called a CHARACTERISTIC)
- A FRACTION (also called a SIGNIFICAND or MANTISSA)

In floating point representation, the number is represented as a combination of a mantissa, m, and an exponent e.

In such a representation it is possible to float a decimal point within number towards left or right side.

For example:

53436.256 = 5343.6256 x 101

534.36256 x 102

and so on

In general floating representation of a number of any base may be written as: N = ±Mantissa x (Base) ±exponent

Normalized Floating Representation

It has been noted that a number may have more than one floating point representations. In order to have unique representation of non-zero numbers a normalized floating point representation is used.

A floating point representation in decimal number system is normalized floating point iff mantissa is less than 1 and greater than equal to .1 or 1/10(base of decimal number system).

i.e. .1≤ |mantissa| <1

A floating point representation in binary number system is normalized floating point iff mantissa is less than 1 and greater than equal to .5 or 1/2(base of binary number system).

i.e. .5≤ |mantissa| <1

A floating point representation is called normalized floating point representation iff mantissa lies in the range:

1/base≤ |mantissa| <1

Disadvantages of floating point representation

- All the eight digits cannot be stored, since two digits are required for exponent.
- Some specific rules are to be followed when arithmetic operations are performed with such numbers.
- Data loss due to truncation of digits.

BCD CODE:

- BCD stands for Binary Coded Decimal
- It is one of the early computer codes
- It uses 6 bits to represent a symbol
- It can represent 64 (26) different characters
- Binary Coded Decimal is a 4-bit code used to represent numeric data only. For example, a number like 9 can be represented using Binary Coded Decimal as 1001_2.
- Binary Coded Decimal is mostly used in simple electronic devices like calculators and microwaves. This is because it makes it easier to process and display individual numbers on their Liquid Crystal Display (LCD) screens.
- **A standard Binary Coded Decimal**, an enhanced format of Binary Coded Decimal, is a 6-bit representation scheme which can represent non-numeric characters. This allows 64 characters to be represented. For letter A can be represented as 110001_2 using standard Binary Coded Decimal

In computing and electronic systems, binary-coded decimal (BCD) is an encoding for decimal numbers in which each digit is represented by its own binary sequence. Its main virtue is that it allows easy conversion to decimal digits for printing or display and faster decimal calculations. Its drawbacks are the increased complexity of circuits needed to implement mathematical operations and a relatively inefficient encoding – it occupies more space than a pure binary representation. Even though the importance of BCD has diminished, it is still widely used in financial, commercial, and industrial applications.

In BCD, a digit is usually represented by four bits which, in general, represent the values/digits/characters 0-9. Other bit combinations are sometimes used for sign or other indications.

To BCD-encode a decimal number using the common encoding, each decimal digit is stored in a four-bit nibble.

DECIMAL	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Thus, the BCD encoding for the number 127 would be:

0001 0010 0111

Since most computers store data in eight-bit bytes, there are two common ways of storing four-bit BCD digits in those bytes:

- each digit is stored in one byte, and the other four bits are then set to all zeros, all ones (as in the EBCDIC code), or to 0011 (as in the ASCII code)
- two digits are stored in each byte.

The best way to introduce the Packed Decimal Data format is to first present an earlier format for encoding decimal digits. This format is called **BCD**, for "Binary Coded-decimal".

As may be inferred from its name, it is a precursor to EBCDIC (Extended BCD Interchange Code) in addition to heavily influencing the Packed Decimal Data format.

We shall introduce BCD and compare it to the 8–bit unsigned binary previously discussed for storing unsigned integers in the range 0 through 255 inclusive. While BCD doubtless had encodings for negative numbers, we shall postpone signed notation to Packed Decimal.

The essential difference between BCD and 8–bit binary is that BCD encodes each decimal in a separate 4–bit field (sometimes called "nibble" for half–byte). This contrasts with the usual binary notation in which it is the magnitude of the number, and not the number of digits, that determines whether or not it can be represented in the format.

To emphasize the difference between 8–bit unsigned binary and BCD, we shall examine a selection of two–digit numbers and their encodings in each system.

Decimal Number	**8–bit binary**	**BCD (Represented in binary)**
05	0000 0101	0000 0101
13	0000 1101	0001 0011
17	0001 0001	0001 0111
23	0001 0111	0010 0011
31	0001 1111	0011 0001

64	0100 0000	0110 0100
89	0101 1001	1000 1001
96	0110 0000	1001 0110

As a hypothetical aside, consider the storage of BCD numbers on a byte–addressable computer. The smallest addressable unit would be an 8–bit byte. As a result of this, all BCD numbers would need to have an even number of digits, as to fill up an integral number of bytes. Our solution to the storage of integers with an odd number of digits is to recall that a leading zero does not change the value of the integer.

Advantages of BCD Codes

- It is very similar to decimal system.
- We need to remember binary equivalent of decimal numbers 0 to 9 only.

Disadvantages of BCD Codes

- The addition and subtraction of BCD have different rules.
- The BCD arithmetic is little more complicated.
- BCD needs more number of bits than binary to represent the decimal number. So BCD is less
- Efficient than binary.

GRAY CODE:

A Gray Code represents numbers using a binary encoding scheme that groups a sequence of bits so that only one bit in the group changes from the number before and after. It is named for Bell Labs researcher Frank Gray, who described it in his 1947 patent submittal on Pulse Code Communication. He did not call it a Gray Code, but noted there was no name associated with the novel code and referred to it as a Binary Reflected Code for the way he determined the groupings and number representations. When the patent was granted in 1953 others began to refer to the encoding scheme as the Gray Code. The encoding was used in some applications prior to Gray's patent, but Frank Gray was the first to document the code and how to develop it using the 'reflection' method in a patent application.

It is the non-weighted code and it is not arithmetic codes. That means there are no specific weights assigned to the bit position. It has a very special feature that, only one bit will change each time the decimal number is incremented as shown in fig. As only one bit changes at a time, the gray code is called as a unit distance code. The gray code is a cyclic code. Gray code cannot be used for arithmetic operation.

A Gray Code is not weighted, the columns of bits do not reflect an implicit base weight as the Binary number system does. In the Binary number system, the least significant bit (right most) column is weighted as 2^0 (1); the second column, 2^1 (2); the third 2^2 (4), and so on, each column representing the base raised to a power. The final value is determined by multiplying the bit by the column weight and adding the column results, so in Binary the 4 bit number '0011' represents 1*2 + 1*1 = 3. Columns in the Gray Code are positional but not weighted and since a Gray Code is a numeric representation of a cyclic encoding scheme, where it will roll over and repeat, it isn't suited for mathematical operations. Gray Code sequences have to be converted to Binary or Binary Coded Decimal (BCD) if they are used in mathematical computations or for displays.

A member of unit-distant, minimal-change codes, where only one bit of a sequence changes as the number count progresses, Gray Codes provide more flexibility with respect to misalignment and synchronization because they limit the maximum read error to one unit. This property also makes them useful in error detection schemes. Better than a parity check, communication systems use Gray Codes to detect unexpected changes in data. If the bits in a number are summed, the sum of the next number should only change by one with the sum alternating even and odd.

A comparison of the first ten numbers in Decimal, Binary and Gray Code is shown in Table

Table 1. Decimal, Binary, Gray Code Numbers

Decimal(base 10)	Binary(base2)	Binary-Reflected(no base)
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010

4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111

USES

Gray Codes have gone beyond the encoding mask documented in the patent; Gray Codes are now incorporated into systems where one-bit detection is useful.

In aircraft, where altimeters are normally mechanical, an encoding disk synced to the dials may produce a type of Gray Code output (Gillham Code) to send to the transponder for processing. This specialized code reflects a one bit change for each 100 foot increment allowing the altitude to be tracked.

Application of Gray code

- Gray code is popularly used in the encoders.
- A encoder produces a code word which represents the angular position

ASCII CODE

The American Standard Code for Information Interchange or ASCII assigns values between 0 and 255 for upper and lower case letters, numeric digits, punctuation marks and other symbols. ASCII characters can be split into the following sections:

❑ 0 – 31 Control codes

❑ 32 – 127 Standard, implementation-independent characters

❑ 128 – 255 Special symbols, international character sets – generally, non-standard characters.

- American standard code for information interchange (ASCII) is a 7-bit code, which means that only 128 characters i.e. 2^7 can be represented. However, manufactures have added an eight bit to this coding scheme, which can now provide for 256 characters.
- This 8-bit coding scheme is referred to as an 8-bit American standard code for information interchange. The symbolic representation of letter A using this scheme is 1000001_2..

Not only does the computer **not** understand the (decimal) numbers you use, it doesn't even understand letters like "ABCDEFG...". The fact is, it doesn't care. Whatever letters you input into the computer, the computer just saves it there and delivers to you when you instruct it so. It saves these letters in the same Binary format as digits, in accordance to a pattern. In PC (including DOS, Windows 95/98/NT, and UNIX), the pattern is called **ASCII** (pronounced *ask-ee*) which stands for ***A**merican **S**tandard **C**ode for **I**nformation **I**nterchange*.

In this format, the letter "A" is represented by "0100 0001" ,or most often, referred to decimal 65 in the ASCII Table. The standard coding under **ASCII** is here. When performing comparison of characters, the computer actually looks up the associated ASCII codes and compare the ASCII values instead of the characters. Therefore the letter "B" which has ASCII value of 66 is greater than the letter "A" with ASCII value of 65.

Only the first 128 characters (Codes 00 – 7F in hexadecimal) are standard.

To allow compatibility with telecommunications equipment, computer manufacture is able gravitated toward the ASCII code. As computer hardware became more reliable, however, the need for a parity bit began to fade. In the early 1980s, microcomputer and microcomputer-peripheral makers began to use the parity bit to provide an “extended” character set for values between 12810 and 25510.

Depending on the manufacturer, the higher-valued characters could be any-thing from mathematical symbols to characters that form the sides of boxes to foreign-language characters such as ñ. Unfortunately, no amount of clever tricks can make ASCII a truly international interchange code.