13.3.8 Hex-to-Decimal Conversion

A hex number can be converted to its decimal equivalent by using the fact that each hex digit position has a weight that is a power of 16. The LSD has a weight of $16^0 = 1$, the next higher digit has a weight of $16^1 = 16$, the next higher digit has a weight of $16^2 = 256$, and so on. The conversion process is demonstrated in the examples below :

$$\begin{aligned} 356_{16} &= 3\times16^2 + 5\times16^1 + 6\times16^0 \\ &= 768 + 80 + 6 \\ &= 854_{10} \end{aligned}$$

$$\begin{aligned} 2AF_{16} &= 2\times16^2 + 10\times16^1 + 15\times16^0 \\ &= 512 + 160 + 15 \\ &= 687_{10} \end{aligned}$$

Note that in the second example the value 10 was substituted for A and the value 16 for F in the conversion to decimal. To convert a fractional number,

$$\begin{aligned} 56.08_{16} &= 5\times16^1 + 6\times16^0 + 0\times16^{-1} + 8\times16^{-2} \\ &= 80 + 6 + 0 + 8/256 = 86 + 0.03125 \\ &= 86.03125_{10} \end{aligned}$$

13.3.9 Binary-to-Hex Conversion

Binary numbers can be easily converted to hexadecimal by grouping in groups of four starting at the binary point.

EXAMPLE 13.6 *Convert 1010111010_2 to hexadecimal.*

Solution. Group in fours 10,1011,1010

Convert each number 2 B A

Thus, the solution is 2 BA.

EXAMPLE 13.7 *Convert 10101110.010111_2 to hexadecimal.*

Solution. Groups in fours (inserting zeros before MSB or in the end, wherever required)

1010, 1110 . 0101, 1100

A E . 5 C

$10101110.010111_2 = AE.5C_{16}$

13.3.10 Hex to Binary Conversion

Like the octal number system, the hexadecimal number system is used primarily as a "shorthand" method for representing binary numbers. It is a relatively simple matter to convert a hex number to binary. *Each* hex digit is converted to its 4-bit binary equivalent (table 13.1). This is illustrated below for $9F2_{16}$.

$$\begin{array}{lccc} 9F2_{16} = & 9 & F & 2 \\ & \downarrow & \downarrow & \downarrow \\ & 1001 & 1111 & 0010 \end{array}$$

$$= 100111110010_2$$

NOTE

Hex to binary conversion takes place using **4-bit substitution** for each hex character.

Similarly, $3A6_{16}$ =

3	A	6
↓	↓	↓
0011	1010	0110

$= 001110100110_2$

NOTE

The hexadecimal and octal codes are used as shorthand means of expressing large binary numbers.

Consider another example

$3BF.5C_{16}$ =

3	B	F	5	C
↓	↓	↓	↓	↓
0011	1011	1111	0101	1100

$= 001110111111.01011100_2$

Converting from Any Base to Any OTHER Base

As demonstrated in earlier examples and the table below, there is a direct correspondence between the number systems : with three binary digits corresponding to one octal digit ; four binary digits corresponding to one hexadecimal digit, which you can use to convert from one number system to another.

Table 13.4 *Correspondence of Binary and Octal Number Systems*

BIN	OCT	DEC
000	0	0
001	1	1
010	2	2
011	3	3
100	4	4
101	5	5
110	6	6
111	7	7

For conversion from ***base 2*** to ***base 8***, we **use groups of three bits** and vice-versa.

Table 13.5 *Correspondence of Binary, Octal and Hexadecimal Number Systems*

BIN	HEX	OCT	DEC
0000	0	00	0
0001	1	01	1
0010	2	02	2
0011	3	03	3
0100	4	04	4
0101	5	05	5
0110	6	06	6
0111	7	07	7
1000	8	10	8
1001	9	11	9
1010	A	12	10
1011	B	13	11
1100	C	14	12
1101	D	15	13
1110	E	16	14
1111	F	17	15

For conversion from ***base 2*** to ***base 16***, we **use groups of four bits** and vice-versa.

Let us consider some more examples regarding the same, *i.e.*, converting from any number system to another.

EXAMPLE 13.8 *Convert* $1948.B6_{16}$ *to Binary and Octal equivalents.*

Solution.

Hexadecimal	1	9	4	8	.	B	6
Binary	0001	1001	0100	1000	.	1011	0110

(By converting each individual Hex digit to equivalent 4 digit binary from above table 13.5)

$\therefore \quad 1948.B6_{16} = 0001100101001000.10110110_2$

New Octal number can be generated from above Binary equivalent *i.e.*, as follows :

Binary	0	001	100	101	001	000	.	101	101	100
Octal		1	4	5	1	0	.	5	5	4

(By creating groups of 3 binary digits and converting them into equivalent octal no.)

$\therefore \quad 1948.B6_{16} = 14510.554_8.$

EXAMPLE 13.9 *Convert* 75643.5704_8 *to hexadecimal and binary numbers.*

Solution. We shall convert it in following way :

(*i*) From octal to binary — by representing each octal digit to 3 digit binary number.

(*ii*) From the complete binary number, we shall create groups of 4 binary digits around the decimal point.

(*iii*) Convert each 4-digit-binary group to equivalent hex digit.

Octal	7	5	6	4	3	.	5	7	0	4
Binary	111	101	110	100	011	.	101	111	000	100

$\therefore \quad 75643.5704_8 = 111101110100011.101111000100_2$

Binary	0111	1011	1010	0011	.	1011	1100	0100
Hexadecimal	7	B	A	3	.	B	C	4

$\therefore \quad 75643.5704_8 = 7BA3.BC4_{16}$

After learning about various digital number systems, one must know how the data is represented in computers.

13.4 REPRESENTING UNSIGNED INTEGERS IN BINARY

An **unsigned integer** can be either a positive integer or zero but never a negative integer. Before we discuss how unsigned numbers are represented, consider a single digit decimal number :

- In a single decimal digit, you can write a number between 0 and 9 (maximum number is 9).
- In two decimal digits, you can write a number between 0 and 99 (maximum number is 99).
- In three decimal digits, you can write a number between 0 and 999 (maximum number is 999), and so on.

Since 9 is equivalent to $10^1 - 1$; 99 is equivalent to $10^2 - 1$; 999 is equivalent to $10^3 - 1$ etc., *in n decimal digits, you can write a number between 0 and* $10^n - 1$.

Analogously, in the binary number system,

an unsigned integer containing *n bits* can have a value between *0* and $2^n - 1$ *(i.e., out of* 2^n *different values).*

This fact is one of the most important and useful things to know about computers.

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 :

Table 13.6 *Ranges of Unsigned Integers Represented through n-bits*

n (bits)	Minimum value	Maximum Value
8	0	$2^8 - 1$ (=255)
16	0	$2^{16} - 1$ (=65,535)
32	0	$2^{32} - 1$ (= 4,294,967,295) (9+ digits)
64	0	$2^{64} - 1$ (=18,446,744,073,709,551,615) (19+ digits)

Thus the unsigned integers are represented by storing their equivalent binary code through ***n bits***, where *n* can be 8 or 16 or 32 or 64.

13.5 BINARY ADDITION

ALUs don't directly work upon decimal numbers ; rather they process binary numbers as a computer can understand only binary numbers. There are *five* basic cases for binary addition that must be understood before going on. These are :

Case 1 : 0 + 0 = 0

i.e., addition of two binary 0's (zero) results into a binary 0 (zero).

Case 2 : 0 + 1 = 1

i.e., addition of a binary 0 (zero) and a binary 1 (one) results into binary 1 (one).

Case 3 : 1 + 0 = 1

i.e., addition of a binary 1 (one) and a binary 0 (zero) results into binary 1 (one).

Case 4 : 1 + 1 = 10

i.e., Binary 1 + Binary 1 equals Binary 10. Or we can say that in binary numbers, one plus one equals **zero (0), carry one (1).**

Case 5 : 1 + 1 + 1 = 11

i.e., Binary 1 + Binary 1 + Binary 1 equals **Binary 1, carry Binary 1.**

Let us summarize these rules

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10 [0 with carry 1]
1 + 1 + 1 = 11 [1 with carry 1]

Let us now perform binary addition on bigger binary numbers. The binary numbers are also added column-by-column just like decimal numbers. Also, the way results larger than largest decimal digit are carried-over, in binary addition, results larger than 1 are also carried over.

For example,

Decimal Addition

```
  1   <-- This is carried over as 5 + 7 = 12 which is more than 9, hence carried over.
  15
  27
 ----
  42
```

Similarly, Binary Addition will be

Carries

```
 1111
  1111     (Equivalent of 15(10))
 11011     (Equivalent of 27(10))
------
101010     (Equivalent of 42(10))
------
```

Let us perform another addition, step-by-step for

```
 11100
+11010
```

Start with the least significant column *i.e.*, the right most column to get

```
 11100
+11010
------
     0
```

In all examples, 0 + 0 gives 0.

Next, add the bits of the second column (second from right) as follows :

```
 11100
+11010  11111
------
    10
```

This time, 0 + 1 gives

The addition in third column gives

```
 11100
+11010
------
   110
```

In this case, 1 + 0 results in

```
 11100
+11010   (carry 1)
------
 0110
```

As you see, 1 + 1 produces 10 *i.e.*, 0 with a carry of 1.

Finally, the last column gives

```
  1   <-- Carries
  11100
+ 11010
------
 110110
```

Here, 1 + 1 + 1 (previously generated carry) results in 11, recorded as 1 with a carry to the next higher column.

EXAMPLE 13.10 *Add the binary numbers* 01010111 *and* 00110101.

Solution. If you add the bits column by column as earlier explained you will get

```
  111  111   <-- Carries
  01010111
+ 00110101
----------
  10001100
```

EXAMPLE 13.11 *Add the binary numbers 1011 and 110.*

Solution.

```
   11   <-- Carries
   1011
+   110
------
  10001
```

EXAMPLE 13.12 *Add binary numbers* 11110 *and* 11.

Solution.

```
  111   <-- Carries
  11110
     11
------
 100001
```

EXAMPLE 13.13 *Add binary numbers* 11.01 *and* 101.11.

Solution.

```
 1111   <-- Carries
  11.01
 101.11
-------
 1001.00
```

Check Point 13.1

1. What are the bases of decimal, octal, binary and hexadecimal systems ?
2. What is the common property of decimal, octal, binary and hexadecimal number systems ?
3. Complete the sequence of following binary numbers : 100, 101, 110, _____ , _____ , _____ .
4. Complete the sequence of following octal numbers : 525, 526, 527, _____ , _____ , _____ .
5. Complete the sequence of following hexadecimal numbers : 17, 18, 19, _____ _____ , _____ .
6. Convert the following binary numbers to decimal and hexadecimal :
   (*a*) 1010 (*b*) 111010 (*c*) 101011111
   (*d*) 1100 (*e*) 10010101 (*f*) 11011100
7. Convert the following decimal numbers to binary and octal :
   (*a*) 23 (*b*) 100 (*c*) 145
   (*d*) 19 (*e*) 121 (*f*) 161
8. Convert the following hexadecimal numbers to binary :
   (*a*) A6 (*b*) A07 (*c*) 7AB4
   (*d*) BE (*e*) BC9 (*f*) 9BC8
9. Convert the following binary numbers to hexadecimal and octal :
   (*a*) 10011011101
   (*b*) 1111011101011011
   (*c*) 11010111010111
   (*d*) 1010110110111
   (*e*) 10110111011011
   (*f*) 1111101110101111

13.6 CHARACTER/STRING REPRESENTATION

In addition to numerical data, a computer must be able to handle numerical information. In other words, a computer should recognize codes that represent letters of the alphabet, punctuation marks, and other special characters as well as numbers. These codes are called *alphanumeric codes*. A complete alphanumeric code would include the 26 lowercase letters, 26 uppercase letters, 10 numeric digits, 7 punctuation marks, and anywhere from 20 to 40 other characters, such as +, /, #, %, *, and so on. We can say that an alphanumeric code represents all of the various characters and functions that are found on a standard typewriter (or computer) keyboard.

13.6.1 ASCII Code

The most widely used alphanumeric code, the *American Standard Code for Information Interchange* (ASCII), is used in most microcomputers and minicomputers, and in many mainframes. The ASCII code (pronounced "askee") is a 7-bit code, and so it has $2^7 = 128$ possible code groups. This is more than enough to represent all of the standard keyboard characters as well as control functions such as the (RETURN) and (LINEFEED) functions. Table 13.7 shows a partial listing of the ASCII code. In addition to the binary code group for each character, the table gives the octal and hexadecimal equivalents.

Table 13.7 *Partial Listing of ASCII Code*

Character	7-Bit ASCII	Decimal	Octal	Hex	Character	7-Bit ASCII	Decimal	Octal	Hex
A	100 0001	65	101	41	Y	101 1001	89	131	59
B	100 0010	66	102	42	Z	101 1010	90	132	5A
C	100 0011	67	103	43	0	011 0000	48	060	30
D	100 0100	68	104	44	1	011 0001	49	061	31
E	100 0101	69	105	45	2	011 0010	50	062	32
F	100 0110	70	106	46	3	011 0011	51	063	33
G	100 0111	71	107	47	4	011 0100	52	064	34
H	100 1000	72	110	48	5	011 0101	53	065	35
I	100 1001	73	111	49	6	011 0110	54	066	36
J	100 1010	74	112	4A	7	011 0111	55	067	37
K	100 1011	75	113	4B	8	011 1000	56	070	38
L	100 1100	76	114	4C	9	011 1001	57	071	39
M	100 1101	77	115	4D	blank	010 1000	32	040	20
N	100 1110	78	116	4E	.	010 1110	46	056	2E
O	100 1111	79	117	4F	(	110 1000	40	050	28
P	101 0000	80	120	50	+	010 1011	43	053	2B
Q	101 0001	81	121	51	$	010 0100	36	044	24
R	101 0010	82	122	52	*	010 1010	42	052	2A
					)	010 1001	41	051	29
S	101 0011	83	123	53	–	010 1101	45	055	2D
T	101 0100	84	124	54	/	010 1111	47	057	2F
U	101 0101	85	125	55	,	010 1100	39	054	2C
V	101 0110	86	126	56	=	011 1101	61	075	3D
W	101 0011	87	127	57	(RETURN)	000 1101	13	015	0D
X	101 1000	88	130	58	(LINEFEED)	000 1010	10	012	0A

EXAMPLE 13.14 *The following is a message encoded in ASCII code. What is the message ?*

1001000 1000101 1001100 1010000

Solution. Convert each 7-bit code to its hex equivalent. The results are

48 45 4C 50

Now locate these hex values in Table 13.7 and determine the character represented by each. The results are

H E L P

The ASCII code is used for the transfer of alphanumeric information between a computer and input/output devices such as video terminals or printers. A computer also uses it internally to store the information that an operator types in at the computer's keyboard.

The following example illustrates this.

EXAMPLE 13.15 *An operator is typing in a BASIC program at the keyboard of a certain micro-computer. The computer converts each keystroke into its ASCII code and stores the code in memory. Determine the codes that will be entered into memory when the operator types in the following BASIC statement :*

GOTO 25

Solution. Locate each character (including the space) in Table 13.7 and record its ASCII code.

G	1000111
O	1001111
T	1010100
O	1001111
(Space)	0100000
2	0110010
5	0110101

The main advantage of ASCII is its simplicity — it uses one byte to represent one character.

There is **extended ASCII** that uses 8 bits to represent various characters. It can represent 256 characters, as opposed to 128 characters of ASCII.

Apart from ASCII there are other systems that are also used to represent various symbols.

In the following lines, we are talking about some of these — *ISCII* and *Unicode*.

13.6.2 ISCII Code

With the advent of computerization considerable work has been undertaken to facilitate the use of Indian languages on computers. These activities were generally limited to specific languages and were independent exercises of various organizations, thus, making data-interchange impossible. In such a scenario, it was important to have a common standard for coding Indian scripts. In 1991, the Bureau of Indian Standards adopted the *Indian Standard Code for Information Interchange (ISCII), the ISCII standard* that was evolved by a standardization committee. This is an eight-bit code capable of coding 256 characters. ISCII code retains all ASCII characters and offers coding for Indian scripts also. Thus, it is also called *Indian Scripts Code for Information Interchange.*

All GIST products are based on ISCII. Also ISCII has been used by *IBM for PC-DOS*, *Apple for ILK*, and by several other companies that are developing products and solutions based on this representation. Also ISCII has been made mandatory for the data being collected by organizations like *The Election Commission*, and for projects as *Land Records Project* etc.

This standard does not only apply to the *Devanagari* script , but also to the *Gurmukhi, Gujarati, Oriya, Bengali, Assamese, Telugu, Kannada, Malayalam* and *Tamil script*. Because the structure of these scripts is so similar that a single coding can be applied to all of them, immediately providing *transliteration* between the scripts. Some of the ISCII versions for different scripts are being given below.

	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
240		ँ	ं	ः	अ	आ	इ	ई	उ	ऊ	ऋ		ए	ऐ	ऍ	
260	ओ	औ	ऑ	क	ख	ग	घ	ङ	च	छ	ज	झ	ञ	ट	ठ	ड
300	ढ	ण	त	थ	द	ध	न		प	फ	ब	भ	म	य		र
320	ऱ	ल	ळ		व	श	ष	स	ह	INV	ा	ि	ी	ु	ू	ृ
340		ॅ	ॆ	े		ै	ॉ	ॊ	ो	ौ	्					ATR
360	EXT	०	१	२	३	४	५	६	७	८	९					

Devnagari ISCII script © ISCII

	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
240		ঁ	ং	ঃ	অ	আ	ই	ঈ	উ	ঊ	ঋ		এ	ঐ		
260	ও	ঔ		ক	খ	গ	ঘ	ঙ	চ	ছ	জ	ঝ	ঞ	ট	ঠ	ড
300	ঢ	ণ	ত	থ	দ	ধ	ন		প	ফ	ব	ভ	ম	য	য়	র
320		ল				শ	ষ	স	হ	INV	া	ি	ী	ু	ূ	ৃ
340		ৎ	ৈ			ো	ৌ		্	়	ৗ					ATR
360	EXT	০	১	২	৩	৪	৫	৬	৭	৮	৯					

Bengali ISCII script © ISCII

13.6.3 Unicode

You have learnt about two different encoding schemes ASCII and ISCII in previous sections. These encoding schemes represent different sets of characters belonging to different languages by assigning a number to each of the character. Likewise, there are many encoding schemes available that represent different characters of different languages.

As world is becoming a global village thanks to modern technology, a need was being felt for an encoding scheme that could represent all the known languages characters through one encoding scheme. **Unicode** is the answer.

Unicode is developed as a **universal character set** with an aim :

- to define all the characters needed for writing the majority of known languages in use on computers in one place.
- to be a superset of all other character sets that have been encoded.

Before Unicode was invented, there were hundreds of different encoding systems for assigning these numbers. No single encoding could contain enough characters: for example, the European Union alone requires several different encodings to cover all its languages. Even for a single language like English, no single encoding was adequate for all the letters, punctuation, and technical symbols in common use.

These encoding systems also conflict with one another. That is, two encodings can use the same number for two different characters, or use different numbers for the same character. For example, the character **code 0×FF** represents **Ÿ** in *west European code page ISO8859-1*, while the same code represents **Я** in Russian *code page 1251*.

Check Point

13.2

1. Encode "INDIA" and "MARCHING" into ASCII codeform.
2. ASCII code is ____ bit code, extended ASCII is ____ bit code and Unicode is ____ bit code.
3. What is the use of ASCII code and Unicode ?
4. What is the full form of ASCII and Unicode ?
5. Name two Indian languages in Unicode along with their script names.

Any given computer (especially servers) needs to support many different encodings; yet whenever data is passed between different encodings or platforms, that data always runs the risk of corruption.

Encoding Terminology

Before we proceed further, it is important to discuss basic terms related to *encoding*.

Encoding Scheme It is a predefined way for converting information, character by character in a machine *intelligible* code. Same character set may be represented through different encoding schemes.

Code Space It refers to all the codes that an encoding scheme uses to represent characters. *e.g.*, ASCII encoding scheme has a code space from 0 to 127 (0×0 to 0×7F)

Code Point The *code point* refers to a code (from a code space) that represents a single character from the character set represented by an encoding scheme, *e.g.*, 0×41 is one code point of ASCII that represents character 'A'.

A code point value represents the position of a character in the coded character set. For example, the code point for the letter **á** in the Unicode coded character set is 225 in decimal, or E1 in hexadecimal notation (0×E1).

ASCII has 128 code points while unicode has 1,112,064 code points.

Code Unit It refers to unit of storage (number of bits used) used to represent one encoded *code point*. It is important to understand this *e.g.*, UTF-8 encoding scheme uses 8 bits' units to represent characters, but it is a variable length scheme. For some characters it uses just 8 bits (1 byte), for some it may use more number of 8 bit units, *e.g.*, to represent snowman character [☃], it requires 24 bits or 3 UTF-8 units. In this case we will say that *snowman code point* uses 3 *UTF-8 code units*.

Now we can proceed further with our discussion of *Unicode* and *Unicode* based *encoding* schemes.

Unicode Encoding Schemes

Unicode defines multiple encoding systems to represent characters. These are UTF-8, UTF-16 and UTF-32. Let us discuss about these Unicode encoding schemes.

The character encoding scheme reflects the way the coded character set is actually **mapped to bytes** in form of binary code (machine intelligible code) for manipulation in a computer.

13.6.3A UTF-8 (Unicode Transformation Format)-8

UTF-8 is a variable-width encoding that can represent every character in Unicode character set. In other words, it can encode each of the 1,114,112 *code points* in the Unicode character set.

The *code unit* of UTF-8 is 8 bits, called an **octet**. UTF-8 can use 1 to maximum 6 octets to represent *code points* depending on their size, although till now it has used upto 4 octets to represent any character.

UTF-8 is a type of multi-byte encoding. Sometimes you only use 8 bits to store the character, other times, 16 or 24 or more bits. The challenge with a multi-byte encoding is how do you know, for example, that these 16 bits represent a single two-byte character and not two one-byte characters. UTF–8 solves this character boundary problem!

NOTE

UTF-8 is another brilliant idea by Ken Thompson (one of the creators of UNIX).

Unicode codepoints are often written as U+<codepoint number> *e.g., U+0041* represents letter 'A'. We shall represent following Unicode codepoints with UTF-8 encoding for understanding purposes :

Unicode CodePoint	Symbol/character
U + 0024	'$'
U + 0041	'A'
U + 00A2	¢
U + 00F1	ñ
U + 2122	™

Before we proceed, let us recall that UTF-8 is a variable length encoding scheme. That is, it uses different number of bytes or **octets** (set of 8 bits) to represent different characters. Following table 13.8 gives an idea about how many **octets** will be used to represent a Unicode code point.

Table 13.8 *Unicode Code Points and No. of Octets used in UTF-8*

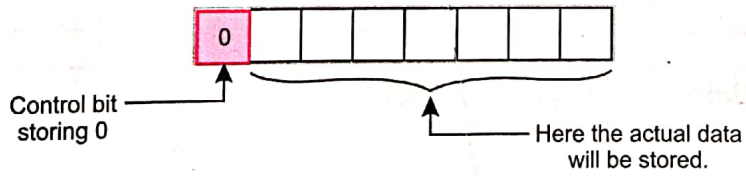
Unicode Code Points (in decimal)	Unicode Code Points (in Hexadecimal)	Number of Octets used
U-0 – U-127	(U+00 to U+07F)	1 octet (8 bits)
U-128 – U-2047	(U+80 to U+7FF)	2 octets (16 bits)
U-2048 – U-65535	(U+800 to U+FFFF)	3 octets (24 bits)
U-65536 – U-2097151	(U+10000 to U+1FFFFF)	4 octets (32 bits)

Let us now learn how UTF-8 represents different code points with varying lengths.

UTF-8 *1 Octet* (8-bits) Representation

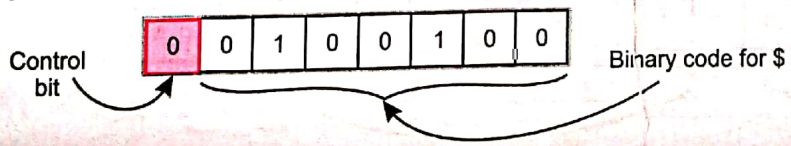
For 1 byte or 1 octet representation, UTF-8 uses the left-most bit as **control bit** which stores control code as **0**. **Control bits are special bits that store the control code and not the actual data.** The rest of the bits store the actual data's binary code.

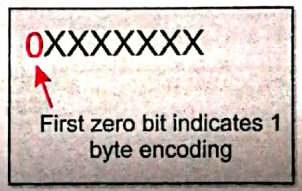
That is, for code points U + 0 to U + 7F (0 to 127 decimal), UTF-8 will store them in octets like :



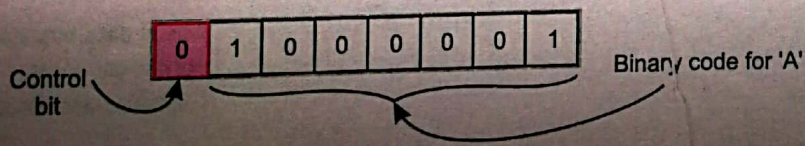
For example, consider these :

- Symbol $ [code point U + 0024] [Binary value for $: 0100100]





- Letter '*A*' [code point U + 0041] [Binary value for 'A' : 1000001]



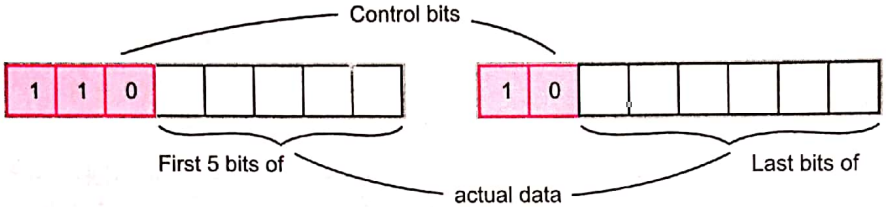
Control code for UTF-8 1-octet representation

UTF-8 2 *Octet* (16-bits) Representation

For 2 byte or 2 octet representation, UTF-8 uses these bits for storing control code **110, 10** :

- Left most 3 bits of first octet that store control code **110.**
- Left most 2 bits of next octet that store control code **10.**

That is

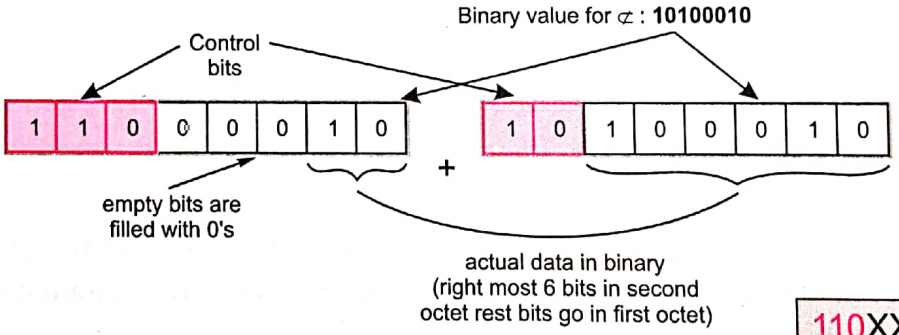


NOTE

UTF-8 encoding uses some control bits along with data bits to represent code points.

Consider these examples :

- Symbol ∝ [Unicode code point U + 00A2]

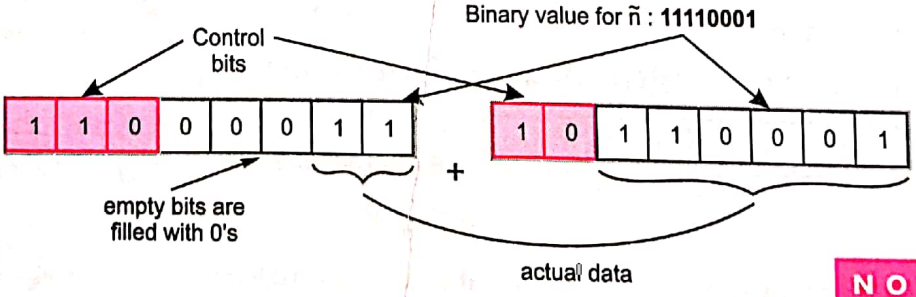


110XXXXX 10XXXXXX

110 in the first bit and 10 in the second bit indicate 2 byte encoding

- Symbol ñ [Unicode code point U + 00F1]

Control code for UTF-8 2-octet representation



NOTE

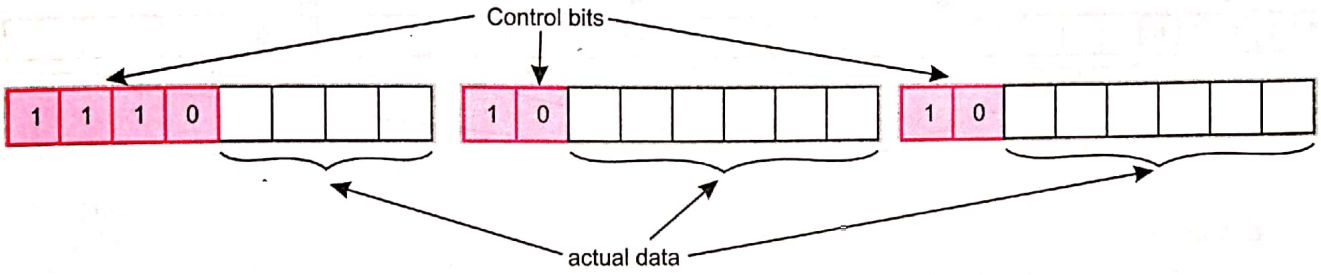
Actual data bits start filling from right most octet.

UTF-8 *3 Octet* (24-bits) Representation

For 3 byte or 3 octet representation, UTF-8 uses following bits for storing control code :

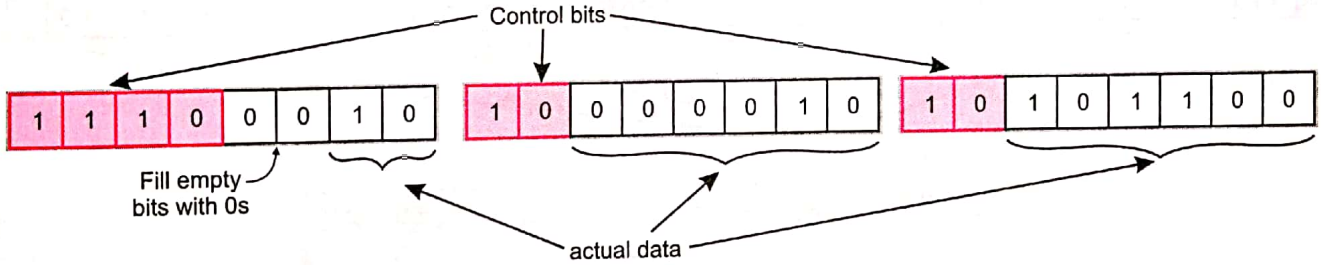
- Left most 4 bits of first octet to store control code **1110**.
- Left most 2 bits of middle octet to store control code **10**.
- Left most 2 bits of third octet to store control code **10**.

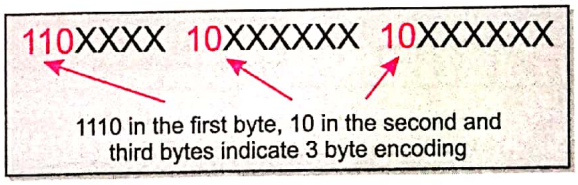
This is,



Consider these examples :

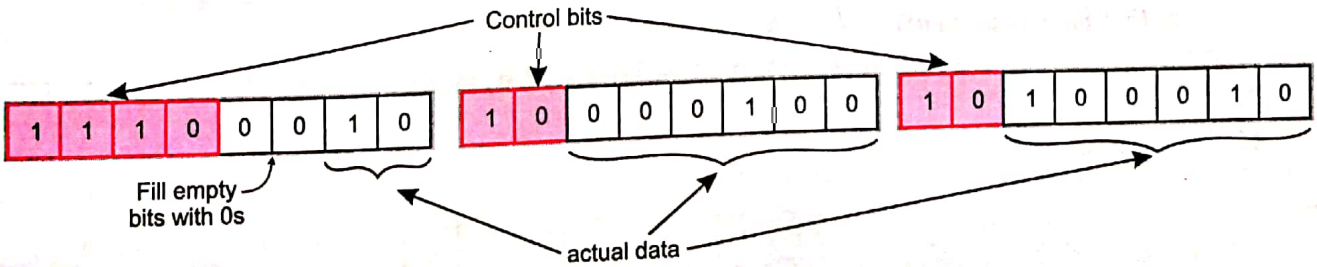
- Symbol € [Unicode code point U + 20AC] [Binary value : 10000010101100]





Control code for UTF-8 3-octet representation

- Symbol ™ (Trademark)
  [Unicode code point U + 2122] [Binary code : 10000100100010]

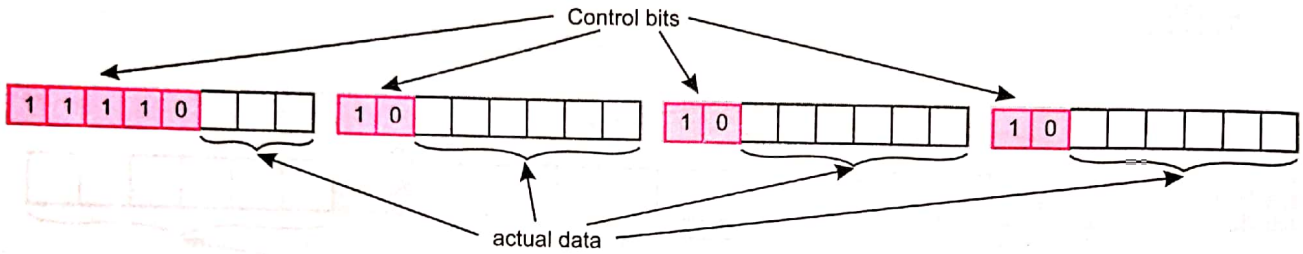


UTF-8 4 Octet (32-bits) Representation

For 4 byte or 4 octet (32-bits) representation, UTF-8 uses following bits as control bits

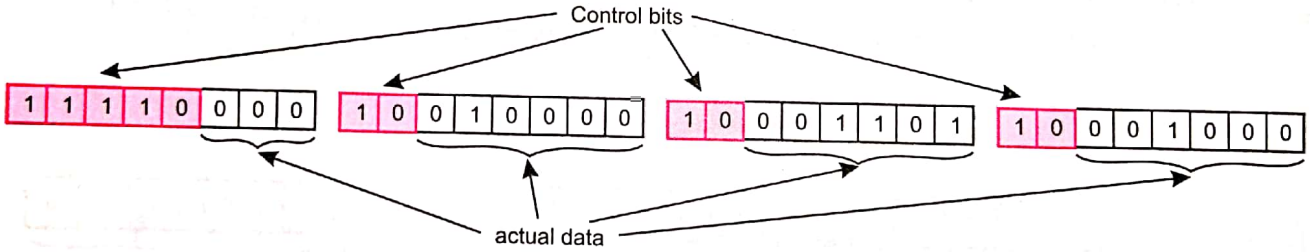
- Left most 5 bits in first octet to store control code 11110
- Left most 2 bits in each of the next three octets to store control **10**

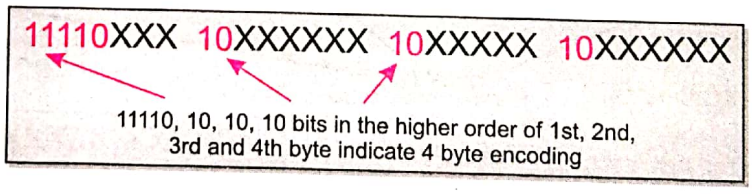
That is,



Examples :

- **Symbol .** [Unicode codepoint U + 10348] [Binary value : 00001000000011010001000]





Control code for UTF-8 4-octet representation

EXAMPLE 13.16 *Represent ~ symbol with Unicode code point as U+007E in hex and 126 in decimal, in UTF-8.*

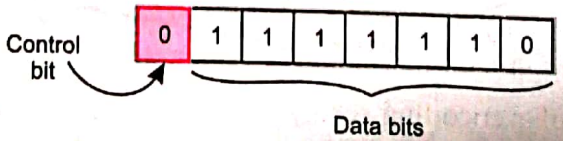
Solution. For Unicode code points in the decimal range **0-127, UTF-8 1-octet representation** is used. (Refer Table 13.8)

Given code point : U + 00 7E (in hex)

(Converting hex to binary with 4 bit replacement)

0111 1110 (in binary) = **01111110**

UTF-8 representation will be



EXAMPLE 13.17 *Represent © copyright symbol with Unicode code point U+00A9 in hex* **in UTF-8 encoding.**

Solution. Given code point 00A9 in hex is equivalent to $A \times 16^1 + 9 \times 16^0 = 169$ in decimal. 169 falls in range **128-2047.**

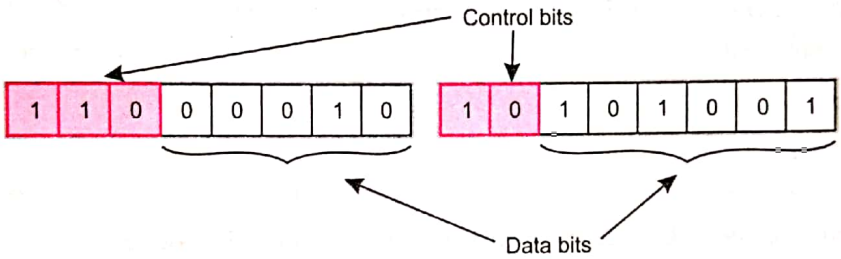
For Unicode code points in decimal range 128...2047, **UTF-8 2-octet** representation is used. (*Refer Table 13.8*)

Given code point is U + 0 0 A 9 (in hex)

(Converting hex to binary with 4-bit replacement)

1010 1001 = **10101001**

UTF-8 representation will be :

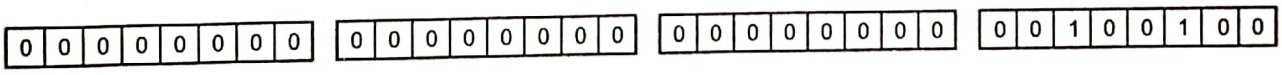


13.6.3B UTF-32

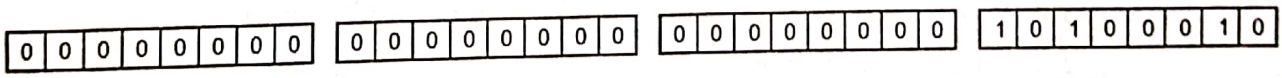
UTF-32 is a fixed length encoding scheme that uses exactly 4 bytes to represent all Unicode code points. That is, it directly stores the binary code of any Unicode code point in 4 bytes.

Consider these examples

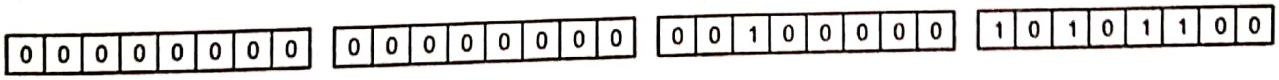
- Symbols $ [Unicode code point : U + 0024, Binary code : 00100100]



- Symbol ¢ [Unicode code point : U + 00A2, Binary code : 10100010]



- Symbol € [Unicode code point : U + 20AC, Binary code : 10000010101100]



There is one more Unicode encoding scheme UTF-16, which is also a variable length encoding scheme that either uses 2 bytes or 4 bytes to represent Unicode code points. But we are not going into details of this encoding scheme as it is beyond and scope of the syllabus.

UTF-8 is the most popular encoding scheme with more than 90% websites using it.