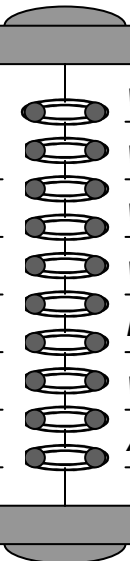


CHAPTER 1: BITS AND BYTES

*“Tall Oaks from little acorns grow”
David Everett (1769-1813)*

CH 1



‘Find Out’ List:

<input type="checkbox"/> <u>What is a BIT ?</u> <input type="checkbox"/> <u>How Many BITS Do I need?</u> <input type="checkbox"/> <u>How do I calculate my needs?</u> <input type="checkbox"/> <u>BITS and computers? How?</u> <input type="checkbox"/> <u>What is a BYTE ?</u> <input type="checkbox"/> <u>Why is it called a BYTE ?</u>	<input type="checkbox"/> <u>What is Parity ?</u> <input type="checkbox"/> <u>What are Abstract Data Types?</u> <input type="checkbox"/> <u>What is ASCII?</u> <input type="checkbox"/> <u>What is an ASCII file?</u> <input type="checkbox"/> <u>Do All Computers use ASCII?</u> <input type="checkbox"/> <u>What is EBCDIC?</u> <input type="checkbox"/> <u>Any other questions?</u>
--	---

Introduction

This chapter starts with the very basics: the machine level representation of how data is stored in the computer. It is not possible to truly understand data structures without a general understanding of basic data types (Chapter 2). Correspondingly, it is not possible to truly understand basic data types without first understanding the components (bits and bytes) which make up these basic data types. It’s basically that simple; in fact, it *is* that simple (no adjectives needed). For an electronic engineer, the material covered in this chapter would be considered profoundly rudimentary. It would be as necessary as memorization of multiplication tables are to normal day functioning; we would not even be aware that we were once forced to memorize them unless it were brought to our attention.

With this in mind, we undertake our discussion of bits and bytes. The material discussed is not meant to qualify you to build your own computer, nor even to give you a complete understanding of how computers operate. The intent is to provide you with an *overview* of how computers function and, basically, how they process data. At times, the chapter might be oversimplified, and certain areas will be omitted. However, as noted above, failure to grasp the concepts presented here probably means that you will experience similar results when it comes to understanding data structures.

Bits

The term bit is an acronym for the expression **B**inary **D**igit. By definition, it is “a single digit in a binary numbering scheme”, meaning it can take on one of two values: 0 and 1 (a binary condition). It is a mutually exclusive state: Something either isn’t (‘0’), or it is (‘1’). It is also the basic unit of information storage.

Def →

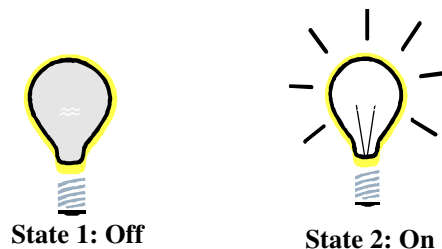
What is a BIT?

? ←

I still don't get it. What does that have to do with computer functioning??

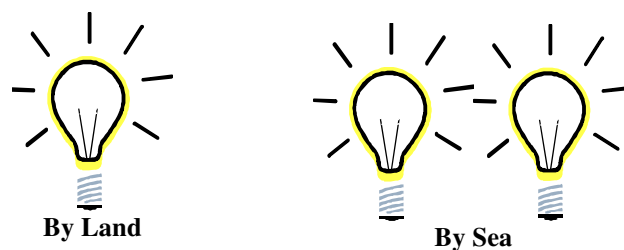
Well, nothing really, except that it provides with a convenient analogy for describing the way in which a computer processes data. The computer has no idea what a 1 or a 0 is. It is, after all, nothing more than an electronic device with voltage running through it. As with all devices requiring electricity, such as a light bulb, we could describe them as either being on or off. In other words, they exist in a binary state.

Figure 1.1.



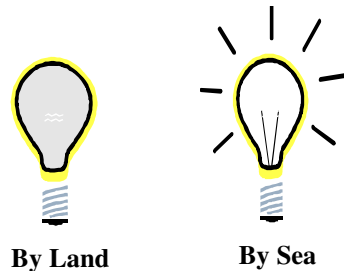
Binary schemes are perhaps the oldest coding systems known to man and the easiest to understand. Take, for example, the code used by Paul Revere to signal how the British were attacking: 1 lantern if by land, 2 lanterns if by sea (or, using more modern technology, light bulbs). It was intended as a simple binary code since there were only two possible states.

Figure 1.2



the code could have also been simplified (and one lantern saved), if the scheme had been *off* if by land, *on* if by sea.¹

Figure 1.3.



Suppose that the British were once again invading, and we had to send a new message to Paul. The problem is that they also might be coming via air, so now there are three possibilities. We could of course use three lanterns/light bulbs, but we really don't need to. On the other hand, we can't use just a single lantern, since we know that there are only two combinations of messages (On or off). Fortunately, since we know that there were already two lanterns available, we could use the following scheme:

Figure 1.4.

<u>Lantern 1.</u>	<u>Lantern 2.</u>	<u>Message</u>	<u>Lanterns/Lightbulbs</u>
Off	Off	By Land	
Off	On	By Sea	
On	Off	By Air	
On	On	(Unused)	

By this simple scheme, we can provide all three messages, with one additional possibility to spare. Since a lantern can either be on or off, we can represent it as a binary digit (bit). Hence, the code can be restated as:

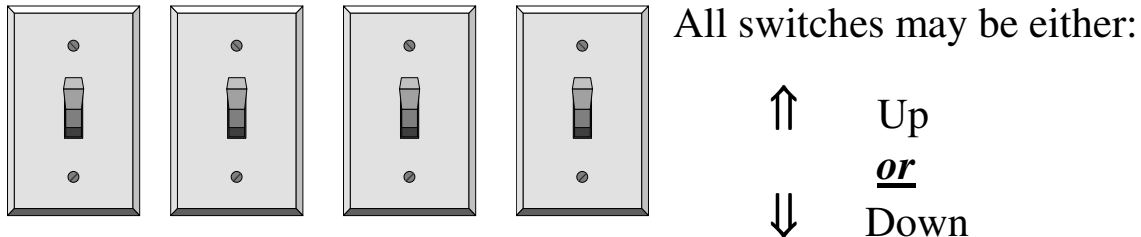
Figure 1.5

00 → By Land
01 → By Sea
10 → By Air
11 → Unused

¹ Of course, if one of the options were that the British might not attack, then there would be three states, and one light bulb would be insufficient.

Let's take another example. Suppose that a student were to visit me in my office. After each visit, the faculty member in the next office always comes in and asks me to rate the student (from excellent to absolutely wretched). We decide to set up a signal system. On the wall between our offices are four light switches, each one capable of sending two messages (since each light switch can be either up or down).

Figure 1.6.



We agree on the following coding scheme:

Figure 1.7.

<u>Switch Settings</u>	<u>Student Rating</u>	<u>Binary Representation</u>
↓↓↓↓	Appears Dead	0000
↓↓↓↑	Don't ask	0001
↓↓↑↓	Poor	0010
↓↓↑↑	Not Good	0011
↓↑↓↓	Average	0100
↓↑↓↑	Fair	0101
↓↑↑↑	Good	0111
↑↓↓↓	Very Good	1000
↑↓↓↑	Excellent	1001

Since our rating scheme only consisted of nine messages, we didn't use up all combinations of light switches. In fact, we could have developed 7 additional ratings (1010, 1011, 1100, 1101, 1110, and 1111). If we now consider the pattern we have developed, we notice a simple progression:

Table 1.1.

No. Bits	No. Messages	Combinations
1	2	0 1
2	4	00 01 10 11
3	8	000 001 010 011 100 101 110 111
4	16	0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

? *Can we predict how many bits we need in advance ??*

Given that each bit has two possible states, for any number (n) of bits, the amount of information conveyed (I) is:

$$2^n = I \quad \text{Formula 1.1}$$

Continuing our table above, we note:

Table 1.2.

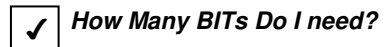
No. Bits	No. Messages	No. Bits	No. Messages	No. Bits	No. Messages	No. Bits	No. Messages
0	$2^0 = 1$	6	$2^6 = 64$	12	$2^{12} = 4,096$	18	$2^{18} = 262,144$
1	$2^1 = 2$	7	$2^7 = 128$	13	$2^{13} = 8,192$	19	$2^{19} = 524,288$
2	$2^2 = 4$	8	$2^8 = 256$	14	$2^{14} = 16,384$	20	$2^{20} = 1,048,596$
3	$2^3 = 8$	9	$2^9 = 512$	15	$2^{15} = 32,768$	21	$2^{21} = 2,097,152$
4	$2^4 = 16$	10	$2^{10} = 1024$	16	$2^{16} = 65,536$	22	$2^{22} = 4,194,304$
5	$2^5 = 32$	11	$2^{11} = 2048$	17	$2^{17} = 131,072$	23	$2^{23} = 8,388,608$

There are a few things to note from this table:

Notes about Adding bits

1. Even though a bit has only 2 states, adding 1 bit *doubles* the amount of information conveyed (i.e., $2^6 = 64$ is twice as much as 2^5 (32))
2. It doesn't take too many bits to convey an enormous amount of information (given 32 bits, we could convey 4,294,967,296 pieces of information; given 64 bits, we could convey $2^{64} = 9,223,372,036,854,780,000$ pieces of information).

Therefore, given any number of bits, we can readily calculate how much information we can convey.



? *That's a completely different way of doing it than in decimal !!!*

No, the concept is the same. Suppose that I asked how many values given a single decimal digit (0, 1, 2, 3, 4, 5, 6, 7, 8 or 9)? Obviously, the answer 10 (or $10^1 = 10$). Now, assume that I asked you how many values you could get given any combination of 2 decimal digits? Once again, the answer is 10^2 , or 100 different combinations (from 0 to 99). For any number of decimal digits, the total number of representations would be:

Table 1.3.

No. Digits	No. Messages	No. Digits	No. Messages	No. Digits	No. Messages
0	$10^0 = 1$	3	$10^3 = 1,000$	6	$10^6 = 1,000,000$
1	$10^1 = 10$	4	$10^4 = 10,000$	7	$10^7 = 10,000,000$
2	$10^2 = 100$	5	$10^5 = 100,000$	8	$10^8 = 100,000,000$

In fact, the basic formula is the same regardless of the numeric base (binary, decimal, octal (base 8) or hexadecimal (base 16)). (We will consider base 8 and base 16 later)

Notes about different bases

The real formula should be:

$$I = B^n$$

Where:

I is the amount of information (combinations) attained

B is the numeric base (binary, octal, decimal, hexadecimal, or any other)

n is the number of digits (in that base) available



But, what if I know how much information I wish to convey. How do I determine how many bits I need ??

To determine the number of bits required, given that we know the number of messages (information) we need, the procedure is reversed:

$$\begin{aligned}
 I &= 2^n && \text{Formula 1.2.} \\
 \log(I) &= n * \log(2) \\
 &= n * 0.30103 \\
 \log(I)/.30103 &= n
 \end{aligned}$$

For Example:

Table 1.4.

Number Messages	Number of Bits Needed	Number Messages	Number of Bits Needed
2	$\log(2)/.30103 = .30103/.30103 = 1.00$	20	$\log(20) /.301 = 1.30/.301 = 4.32$
3	$\log(3)/.30103 = .4771/.30103 = 1.39$	60	$\log(60) /.301 = 1.78/.301 = 5.91$
5	$\log(5)/.30103 = .69897/.30103 = 2.32$	100	$\log(100) /.301 = 2.00/.301 = 6.64$
8	$\log(8)/.30103 = .90309/.30103 = 3.00$	500	$\log(500) /.301 = 2.70/.301 = 8.96$
12	$\log(12)/.30103 = 1.0791/.30103 = 3.58$	1,000	$\log(1,000) /.301 = 3.00/.301 = 9.96$

? *How can we have partial bits ?? How could we have 1.39 Light Switches ??*

We can't. As we saw with our examples about lanterns/light-bulbs and light switches, if we have more capacity than we need, the remaining combinations go unused. Therefore, the correct formula is:

Formula 1.3.

$$\lceil \log(I)/.30103 \rceil = n \quad \text{where: } \lceil \quad \rceil \text{ means ceiling; the value rounded to the next integer}$$

Therefore, for a given number of pieces of information:

Table 1.5.

Number Messages	Number of Bits Needed	Number Messages	Number of Bits Needed	Number Messages	Number of Bits Needed
2	1	20	1	5,000	13
3	2	60	2	10,000	14
5	3	100	3	100,000	17
8	3	500	3	1,000,000	20
12	4	1,000	4	100,000,000	27

Of course we could have gotten the same information from the table where we converted the number of bits into the amount of information which could be conveyed.

How do I calculate my needs?

Bits and Computers²

At this point, it should be obvious that we have gone through this explanation of Bits because computers rely on these concepts to store and manipulate data. We might best explain this by illustrating how data was stored in older (pre-integrated circuit) technologies.

The first and second generation of computers was extremely expensive. One reason (among many others) was because of how the data was stored in either the CPU (Central Processing Unit) or RAM (Random Access Memory), both of which basically stored data the same way. We have already seen how we could store, and pass along, information using lanterns or light switches. The computer



² A number of statements and examples are simplified for the sake of illustration

Figure 1.8.

manipulates data in basically the same manner. Early computers, rather than using switches, relied on small rings known as donuts (they were about the size of a printed ‘o’ on this paper), which had two wires passing through them³. These donuts had wires passing through them, and the wires either had voltage passing along them or not. Hence, as shown in Figure 1.8., they were either in an *on*⁴ (shaded) or *off* (unshaded) state (a binary condition).

If we were to look inside one of these early machines, we might see architecture that resembles the illustration below. Note that some of the first machines had only about 1,000 ‘donuts’ (a row of 16 (all in an *off* position) are illustrated in Figure 1.9.)

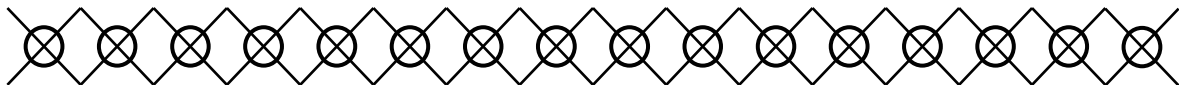


Figure 1.9.

If there *was* voltage running through the wires, the donut was in an *on* (or high voltage) state; if not, it was in an *off* (or low voltage) state. For example:

Voltage:

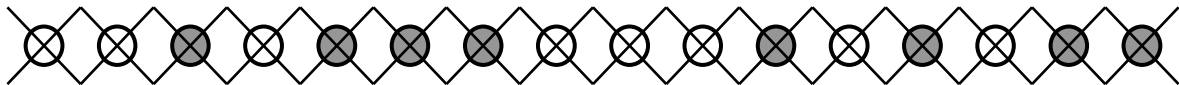


Figure 1.10.

In this case, if we assume that the shaded donuts are ‘on’, and the unshaded donuts are ‘off’, then we would have a bit pattern which would be: **0010111000101011**

The power of computers lies in their ability to quickly change each donut/bit from one state (e.g., on) to another (e.g., off). *How quickly?* The original IBM-PCs ran at 4.77 **MHz** (Megahertz, or *millions of cycles per second*; often referred to as *clock speed*, or the speed at which the CPU operates). The new (at the time of this writing) Processors are running at about 2.8 GHz (GigaHertz, or Billions of cycles per secone). That means that 2,800,000,000 different messages can (theoretically) be transmitted every second.



The real question which needs to be addressed is *How many bits, or donuts’, do we have to group together to be able to represent what we need?*

Bytes

In response to the above question, the answer is: *How much information do you want?*

³ Contrary to popular belief, bits are really not ‘on’ or ‘off’, but rather carry either high voltage or low voltage through them. High voltage implies that both wires have voltage running through them; low voltage implies that voltage is running through only one of the wires.

⁴ In point of fact, there was either high or low voltage passing through them, but the concept remains the same.

That really was the underlying question for the first computer engineers. Obviously, it was necessary to represent the digits (0..9), the alphabet (a..z, A..Z) and some special symbols (! @ # \$ % ^ * ()). At very minimum, we might like to represent 94 symbols:

- 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- 52 alphabetic characters (26 lower case, 26 upper case)
- 32 special characters (! @ # \$ % ^ & * () _ - + | \ = - ` ? < > . : ; “ ‘ { } [] \)
- 94 characters

which means that we would need to group together 7 bits since

$$n = \lceil \log(I)/\log(2) \rceil = \lceil \log(94)/.30103 \rceil = \lceil 1.97313/.30103 \rceil = \lceil 6.55 \rceil = 7$$

With 34 combinations unused since $2^7 = 128$ ($128 - 94 = 34$). There are also a number of 'hidden' symbols (Carriage returns, line-feeds, backspaces, etc.) which we also need to represent, so that unused combinations will not go to waste.

Finally, one more bit was deemed necessary. This bit can be used to either represent more characters (e.g., other alphabets, symbols, etc.) or to detect errors in data. When used for this latter purpose, the additional bit is referred to as a *parity* bit, which brought the total number of bits needed to 8 (We will discuss parity bits shortly).

Def → This total number (8) of bits is universally known as a *byte*, and it is considered the minimum number of bits required to represent the total character set described above AND guard against storage errors⁵. *What is a byte ?*

? ← **Why is it called a byte?**

No particular reason, other than that it what IBM decided to call it. Since IBM was the primary producer of computers when these issues were being settled, they could basically do whatever they wished, and they wished to call it a byte. *Why is it called a byte ?*

? ← **NOW, What was this about parity? I've heard of that before.**

Parity

? ← **Why do we need parity, and how does it work ??**

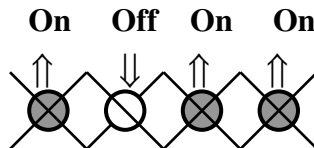
⁵ There is no rule that bits should be grouped together in 8-bits to form a character set. Some early computers, such as CDC's Cyber, used a 6-bit character set. This was achieved by using only capital letters (Cyber was a 'number-cruncher' and wasn't concerned with word-processing).

Well, first of all, hardware mistakes can happen, especially in the earlier computers⁶. A parity bit is intended as a method for catching obvious errors, although it can not catch them all. It works in a very simple manner:

Suppose that we decide that we need to represent only 16 pieces of information. We know that we would need a total of four bits (since $2^4 = 16$). Suppose that we wished to transmit the message:

Figure 1.11.

Voltage:



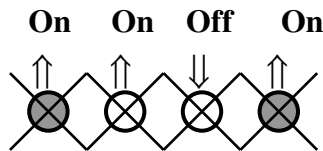
Bit Pattern:

1 0 1 1

Suppose also that the data is transferred, *but* because of a hardware problem, not as **1011**, but as **1001**. That is, as:

Figure 1.12.

Voltage:



Bit Pattern:

1 0 0 1

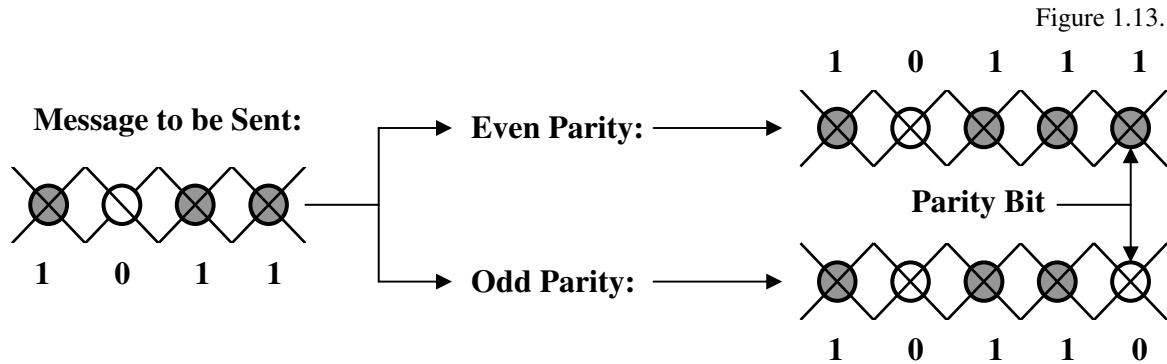
A **parity bit** is an additional bit added to a sequence of bits in order to catch some of these blatant errors. If we look at the 2 sequences of bits, we notice that the first one (the message we intended) has three '1' bits while the other (the mis-sent message) has two. The parity bit, in essence, is intended to catch obvious errors such as this one.



Well, the number of '1' bits can either be either odd or even number; there are no other choices. In the first case, there is an odd number (three) of '1's; in the second, there is an even number. The first issue, then, is *which one do you expect?* If you expect all data transmitted to have *even* parity (an even number of '1' bits), you would add a '1' to the four-bit sequence and transmit it as the five-bit sequence 10111 (containing a total of 4 '1' bits). If you expect all data transmitted to have *odd* parity (an odd number of '1' bits), you

⁶ Early computers, especially first generation machines, were prone to a number of errors. They relied on Vacuum Tubes which gave off an enormous amount of light and heat, which in turn attracted a number of bugs. When these bugs landed on the circuitry, they would be 'fried' on the wiring and in turn cause the data transmitted along the wires to be sent incorrectly. Early programmers spent much of their time scrapping the bugs off the wires. This is where the term *debugging* came from.

would add a '0' to the four-bit sequence and transmit it as the five-bit sequence 10110 (containing a total of 3 '1' bits).



In our case, if we assume even parity and send our message (1011) as 10111, and we in fact end up sending the message 10011 (which contains an odd number of '1' bits), the receiver will know that something is wrong.

What is Parity ?



? Which is better? Is parity supposed to be odd or even ??

Neither one is better than the other⁷. The only important issue is that BOTH the sender and receiver MUST agree on one or the other. If the sender assumes even parity, and the receiver assumes odd parity, well, the messages received will obviously be interpreted as errors.

? Will this approach catch all errors in transmission ??

Obviously, the method is not fail-proof. If, for example, we wish to send the message 1011 and we want to use odd parity. We will send the sequence 10110. If, however, there is an error and we actually send the message 1100 (again adding a '0' as a parity bit to make it odd), the receiver will interpret the message as correct (since the number of 'on' bits is odd). However, if there are enough transmissions (it doesn't take too many or too long, especially if we are operating at 450 MHz), we will pick up the error. The scheme is similar to the concept of *check-digits*, which are used by businesses to help pick up errors in data entry⁸.

⁷ Odd Parity actually appears to be somewhat more common than even parity

⁸ Although we won't go into a detailed discussion here, almost every business which relies on data to be entered at the keyboard uses check-digits, which work much like parity bits. Check your telephone bill. Since you are the only person in the United States with your telephone number (a three-digit area code, and a seven-digit telephone number), you would expect your account number to be ten-digits long. More than likely, it's 13 digits long. The additional digits are referred to as check-digits.

The Abstract Data Type Character

Although we will save our discussion of basic data types until Chapter 2, in fact, we have just introduced our first basic data type: The character (sort of; we will discuss the character data type in more detail in chapter 2). If we think about it, characters are really abstractions. While we are accustomed to the Roman alphabet, it is not the only one in use. There is the Greek Alphabet, Sanskrit, the Cyrillic Alphabet, the Chinese Alphabet, the Hebrew Alphabet, the Arabic Alphabet, and so forth. All of them accomplish the goal (communication), but all do so differently. What sequence of bits should represent any symbol is likewise arbitrary.



But how can data types be abstract? High and low voltage doesn't seem abstract.

We Already Know:

A byte is called a byte and contains 8-bits because IBM decided so.

High and low voltage may not necessarily be abstract, but the manner in which we interpret it can be. We know that bits need not be separated into groups of eight because of some natural phenomenon. The bits, or donuts, are all laid-out in rows, as illustrated in Figures 1.9. and 1.10. We *logically* group them, and retrieve them, in bunches of eight, or as an *octet*. We could group them together in bunches of six (as some manufacturers once did), or 7, or 12, or 25, or 105, or even 1,000. It doesn't really matter. It only matters that we know how they are grouped.

The manner in which we interpret the grouping is equally as arbitrary, although with some foresight as to how they could best be manipulated. For our purposes, we will use the term abstract to mean that our usage of the data stored in the computer is based on our conceptual models of the data can be manipulated, not necessarily on how it is physically stored.

What are Abstract Data Types?

This does bring up an interesting point, however: *How abstract should we be?* The manner in which data is stored may be open to some interpretation, but there needs to be some common ground.

ASCII

Having (more-or-less) agreed upon the use of a 8-bits as a byte to represent a complete character set, another problem developed. It was similar to the *you say 'toe-maa-toe' and I say 'ta-may-tow'* situation. Take, for example, the letter 'c', as in the word *cop* (slang for policeman). No problem. That is, unless you are speaking Russian, where the letter 'c' is pronounced as an 's' and the word '*cop*' (a '*p*' is an '*r*' and '*cop*' is pronounced as *sor*) means rubbish.

While humans can usually recognize differences in pronunciation and meaning, computers can not. Once a coding scheme is established, that machine will recognize only that scheme. If, using a seven bit character set, we instruct the computer to interpret the sequence of on-off voltaged donuts/bits '1000011' as the character 'C', that is all that it will ever represent. It will not be viewed as an 'S'.

The problem was that each manufacturer began building machines based on their own coding schemes. For example, if there were three manufacturers of computers, their coding schemes might appear as:

Table 1.6.

Binary Sequence	Manufacturer #1 Interpretation	Manufacturer #2 Interpretation	Manufacturer #3 Interpretation
00000000	A	0	EOF (End of File)
00000001	B	1	TAB (Tab)
00000010	C	2	SP (Space)
•	•	•	•
00001001	0	a	A
00001010	1	b	B
00001011	2	c	C
•	•	•	•
01111110	CR (Return)	X	0
01111111	BS (BackSpace)	Y	1
11111111	NULL	Z	2

Of course, this meant that data stored on one computer could not be transferred to another, at least not without interpreting it first. If Manufacturer #1 wished to send the message "Hello", Manufacturer #2 might interpret it as "\$^/+-" and Manufacturer #3 might view not get any message, since the characters transmitted were all "hidden", as far as they were concerned. In the early days of computers, intercommunication was not a valued asset.

Consequently, a committee was formed to try and standardize the character set and the manner in which the symbols would be represented in binary. The resultant scheme became known as the American Standard Code for Information Interchange (ASCII). Contrary to popular belief, ASCII is not some unique **Def** language or specialized set of instructions (as we have heard it described before). It is nothing more than an agreement on interpretation, an agreement which most (more on that later) software developers accept.

The complete ASCII character set is given at the end of this Chapter, in Addendum 1.1. Lest we encourage you skip to the end of the chapter now, a partial listing is given in Table 1.6.

Table 1.7.

Bit Sequence	ASCII Value	Character	Description	Bit Sequence	ASCII Value	Character	Description
00000000	0	NULL	Null	01000001	65	A	Cap. A
00000001	1	SOH	Start Head	01000010	66	B	Cap. B
00000010	2	STX	Start Text	01000011	67	C	Cap. C
•	•	•	•	•	•	•	•
00100000	32	SP	Space	01100001	97	a	Lower a
00100001	33	!	Exclamation	01100010	98	b	Lower b
00100010	34	“	Quote.Mark	01100011	99	c	Lower c
•	•	•	•	•	•	•	•
00110000	48	0	Zero	01111101	125	}	Rt. Ellipses
00110001	49	1	One	01111110	126	~	Tilde
00110010	50	2	Two	01111111	127	DEL	Delete

The table is generally self-explanatory. *Binary sequence* refers to how the value is actually stored in the computer. *Character* refers to the agreed upon symbol which the binary sequence represents. The column ASCII Value may however, puzzle some of you. You will notice that there is a definite order to the way in which the ASCII binary sequences are laid out. Because they are ordered, they can be numbered sequentially (starting with 0). Thus, if someone makes a reference to *ASCII 65*, they are not talking about the binary representation for 65 (there is none), but rather to the 65th (actually, 66th) entry on the ASCII list (i.e., the upper case character ‘A’).

The Standard ASCII Character set contains 128 characters (ASCII Values 0 through 127).



Why 128 characters?

Let’s not forget that these tables were developed in an era where there were frequent hardware breakdowns and transmission problems. Although we decided to group bits together in collections of eight (8), of those, one bit was needed for the parity bit. Hence, we had $2^7 = 128$ (which meant the values 0 through 127) combinations to use.



But Computers are much more reliable now. Why don’t we use all of the eight bits in a byte (or $2^8 = 256$) to represent a character set ??

This is all true, and there is an *Extended ASCII Character Set*, which does allow us to use all 8-bits, or 256 characters⁹. Since the major characters needed (at least for the American English character set) were included in the standard ASCII character set, the additional 128 characters are combinations of non-English letters (e.g., é, ä, å), common, but previously omitted symbols (e.g., ¢, £), graphics characters, and frequently used Greek letters and Mathematical symbols (e.g., α, β, γ, Σ, ±, ≥, ≤). The addendum

Def

⁹ The extended ASCII character set is not standardized.

(Addendum 1.2.) at the end of the chapter includes the complete (extended) ASCII listing.

? *What does this have to do with the expression I often hear: An ASCII file?*

ASCII Files

After reading the above discussion, there might be some implication to the term ASCII File. An ASCII file is one which stores data, in 8-bit formats, according to the ASCII scheme we have just described. It's really that simple. If you try and display the contents of a file from an operating system such as DOS¹⁰, the assumption is that it is an ASCII file. That means that when that when it goes to display the file, it takes 8-bits at a time, matches up the 8-bit sequence against the ASCII table, and prints out the character associated with the sequence. **Def** *What is an ASCII file?*

Notice, however, that not all files are in ASCII format (in fact, most are not). For example, this document was created in MS-Word. It contains a number of different formats, such as *italics*, **boldface**, underlining, ^{superscripts}, _{subscripts}, indentations, and commands which set the margins, pagination, footnotes, and others. Look at your ASCII tables. *Are there any sequences of bits which indicate that any of these operations are to be applied?* If you were to try and display this file through the operating system, or reading it directly into a 'Text Editor', you would see a number of strange symbols (e.g., □, ♠, ⌘), and the screen might jump down, or might make beeping sounds. Very simply, that is because the system takes the bits 8 at a time, when in fact, MS-Word might be grouping them as 6, or 16, or 32 bits. Most application packages store data in different ways; the only way to manipulate the data contained in them is through the package. The same holds true for executable, or binary files, which are put together in a non-ASCII format which the operating system can understand.

? *But, that makes no sense. Aren't all ASCII files also binary files (i.e., stored as groupings of bits)?*

Yes and no. All files do indeed store data as bits, and in a binary sequence, but not all files are ASCII Files. ASCII files assume that 1-byte (8 bits) are sufficient to completely represent a piece of datum. As we will soon see, that is not always the case. As we have also seen, sometimes we don't really need all 8-bits. In ASCII files, we use 8-bits, whether we need that much or not. Binary files might use more than 8-bits (say 16, or 32, or 64) to store a particular piece of data.

¹⁰ Some of the newer operating systems, such as Windows 98, analyze the file before attempting to display them, and can display non-ASCII files.

⊙ ? ← ????????????????????

This will hopefully all start making more sense in the next chapter, when we start discussing the manner in which we store different types of data.

⊙ ? ← *Alright. At least we know now all that character data is stored in ASCII. Right ???*

Well *Not quite*

EBCDIC

The idea behind ASCII was certainly relevant. All machines would same coding scheme, and thus data could be readily transferred between them. Programs could be written to run on any machine, and certainly data stored in ASCII format could be used, regardless of platform.

Do all computers use ASCII? IBM decided not to adopt this scheme for all of its machines, and uses its own standard, the Extended Binary Coded Decimal Interchange Code (EBCDIC)¹¹. ⊙ Def

The coding scheme works just like ASCII, but the sequence of symbols represented varies. *What is EBCDIC?* For the sake of contrast, the EBCDIC (vs. ASCII) scheme is also given at the end of this chapter, in Addendum 1.3.

⊙ ? ← *That can't be true! I have an IBM-PC at home, and it uses ASCII, doesn't it?*

Yes, ASCII is used by IBM-PCs (and all the IBM clones). However, that is because IBM did not develop the original operating system for the PC¹². Microsoft developed DOS for their PCs, and they decided to use ASCII. It is the primary reason Bill Gates is the richest man in the world (IBM would not be amused at this explanation).

⊙ ? ← *Is EBCDIC still in use? I haven't heard much about it.*

Yes, it's still in use. All IBM machines, except their PCs use it. It is also frequently used for storing data on large tape drives. Not surprisingly, many of these tape drives are produced by IBM. The standard character sets are essentially the same (only two characters

¹¹ Basically, if you are as big as IBM was, you get to call all of the shots. Since you own the football, if you don't like the way the game is being played, you take your football and go home.

¹² There are two rationales frequently given for this. One is that IBM wanted to get their PCs out in a hurry and felt that if they developed their own operating system the PC would be delayed. The other is that they felt that the PC was a 'fad' and that they should not expend the resources required for development.

differ), although the order (how the characters are related to the bit patterns) are quite different. Compare Addendum 1.1 with Addendum 1.3.

Unicode

In later years, it became apparent that even 256 characters were insufficient to represent all of the symbols needed. The scientific community, book publishers, and bibliographic information services needed considerably more symbols. Additionally, as noted earlier, there was a lack of international character set standards, and the onset of the Internet was forcing demand for software to be simultaneously internationalized and localized.

In response, the Unicode Project was undertaken in 1988. Project members sought a uniform method of encoding which would be more efficient and flexible than existing systems. Today, the *Unicode* Standard encompasses the principal scripts of the world, and provides the foundation for the internationalization and localization of software, including Java, Windows NT, AIX, Visual C++, NetWare 4.0, and QuickDraw GX. Languages that can be encoded include Russian, Arabic, Anglo-Saxon, Greek, Hebrew, Thai, and Sanskrit. The unified Han (China, Japan, Korea) subset contains 20,902 ideographic characters defined by the national and industry standards of each country, as well as Taiwan. In addition, the Unicode Standard includes mathematical operators, technical symbols, geometric shapes, and dingbats.



But, how is that possible using only 8-bits ???

It isn't. Unicode requires a 2-byte (16-bit) character set. As we will see in Chapter 2, if we have 16-bits, we have a total of 65,536 combinations.

Summary

This chapter was intended as a very simple introduction to what a bit is, what a byte is, and why it is important to understand, and how the computer manipulates bits and bytes. Along the way, we introduced our first basic abstract data type, the character, although additional discussion will be provided in Chapter 2. The main intent of this chapter, however, is to lay the groundwork for the understanding of the other two chapters in section I.

If, during the reading of this chapter, you felt bored, *Congratulations!* The material covered might indeed be simplistic. However, realize that that doesn't mean it is not important. As we stated when we first began this chapter, the material covered forms the genetic underpinnings (if you will) of computer functioning.

Chapter Terminology: Be able to fully describe these terms

Abstract Data Type	Donut
ASCII	EBCDIC
ASCII Files	Even Parity
Binary	IBM
Bit	MHz
Byte	Odd Parity
Character Data Types	Parity
Character Set	Parity Bit
Clock Speed	RAM
CPU	

Review Questions

1. Describe three binary conditions which people experience on a daily basis.
2. Remember the student evaluation schemes which the professor in the next-door office and I established? Good News. Five (5) new light switches have been added, bringing the total number of light switches to 9. How many student ratings could we now assign?
3. I am starting my own language. It will consist of only the consonants (B, C, D, F, G, H, J, K, L, M, N, P, Q, R, S, T, V, W, X, Y, Z) which can only be represented in Uppercase. Additionally, I will only allow octal (8 and 9 are not allowed). To further simplify matters, I will only have 5 special characters (+ - space . ,) and 4 hidden characters (CR, LF, EOL, and ESC). I don't care about Parity (my machines will be *perfect*). How many bits do I need to fully represent the entire character set?
4. I have built a new computer that operates at 32.76 MHz. Theoretically, how many times could I change the signal it contains in 1 hour, 15 minutes, and 13 seconds?
5. How many bits would I need to:
 - a. represent all lower case characters?
 - b. represent all states in the United States?
 - c. represent all individuals in the United States (assume 250 Million)?
 - d. represent the National Deficit (1998: Approximately \$4.5 Trillion)?
6. How much Information could I represent given
 - a. 5 bits?
 - b. 11 bits?
 - c. 24 bits?
 - d. 138 bits?

7. Explain what a byte is, why it is called a byte, and how many characters can be represented.
8. How many bytes are in a kilobyte (exactly)? Why?
9. Explain why parity bits were created. Explain the difference between odd and even parity. Given the following bit patterns, show how they would be represented (for the given parity):
 - a. 100 (odd parity)
 - b. 100 (even parity)
 - c. 110101 (odd parity)
 - d. 110101 (even parity)
10. Explain why the data type character is referred to as an *abstract* data type.
11. Explain what ASCII and EBCDIC are. Why are they different?
12. Describe what an ASCII file is. Are there such things as EBCDIC Files?
13. What is a binary file?
14. Memorize the ASCII and EBCDIC tables (this is not a joke; if you were taking a computer science course at a respectable university, you would probably be required to memorize the entire table, in decimal, octal and hexadecimal). If that seems excessive, you might try memorizing the following selections:

Decimal	Char	Description
=====	====	=====
0	NUL	Null
•	•	•
7	BEL	Rings Bell
8	BS	Backspace
•	•	•
13	CR	Carriage Return
•	•	•
27	ESC	Escape
•	•	•
32	SP	Space
•	•	•
48	0	Zero
.....	To	
57	9	Nine
•	•	•
65	A	Uppercase A
.....	To	
90	Z	Uppercase Z
•	•	•
97	a	Lowercase a
.....	To	
122	z	Lowercase z

There are actually only nine (9) decimal values that you need to memorize. If you know that that decimal representation of the character '0' is 48, then you know that '1' is 50, '2' is 51, and so forth. If you know that the representation of the character 'a' is 97, then you know that 'b' is 98, 'c' is 99, and so forth. You end up knowing 67 ASCII values.

What questions should I know?

Answers to Review Questions (NOTE: checking the answers before you have tried to answer the questions doesn't help you at all)

1. On/Off, Male/Female, AM/PM, Married/Unmarried, Dead/Alive, etc.
(Note: Each of these are truly binary states: One is, for example, either married or unmarried; contrary to some beliefs, one cannot be partially married or partially unmarried. Similarly, High/Low is NOT a binary condition, since there are an infinite number of levels between high and low).
2. $I = 2^9 = 512$.
3. Given: 21 Letters + 8 Digits + 5 Special characters + 4 Hidden Characters = 38 Characters

$$\left\lceil \frac{\log(38)}{\log(2)} \right\rceil = \left\lceil \frac{1.5798}{0.30103} \right\rceil = \left\lceil 5.248 \right\rceil = 6 \text{ Bits}$$
4. 1 hour = 60 minutes

$$\begin{array}{l} + 15 \text{ minutes} \\ \hline 75 \text{ minutes} * 60 = 4,500 \text{ seconds} \\ + 15 \text{ seconds} \\ \hline 4,515 \text{ seconds} * 32,760,000 = 147,911,400,000 \text{ times} \end{array}$$
5. a. 26 characters => $n = \left\lceil \frac{\log(26)}{\log(2)} \right\rceil = \left\lceil \frac{1.41497}{0.301} \right\rceil = \left\lceil 4.7341 \right\rceil = 5$
 b. 50 States => $n = \left\lceil \frac{\log(50)}{\log(2)} \right\rceil = \left\lceil \frac{1.69897}{0.301} \right\rceil = \left\lceil 5.6444 \right\rceil = 6$
 c. 250 Million => $n = \left\lceil \frac{\log(250M)}{\log(2)} \right\rceil = \left\lceil \frac{8.39794}{0.301} \right\rceil = \left\lceil 27.9 \right\rceil = 28$
 d. 4.5 Trillion => $n = \left\lceil \frac{\log(4.5T)}{\log(2)} \right\rceil = \left\lceil \frac{12.653}{0.301} \right\rceil = \left\lceil 42.18 \right\rceil = 43$
6. a. 5 bits = $2^5 = 32$ pieces of information
 b. 11 bits = $2^{11} = 512$ pieces of information
 c. 24 bits = $2^{24} = 16,777,216$ pieces of information
 d. 138 bits = $2^{138} = 34,844,914,372,700,000,000,000,000,000,000,000,000,000,000,000$ pieces of information

7. One byte equals eight bits and can contain up to $2^8 = 256$ pieces of information, although the standard character set contains only $2^7 = 128$ pieces of information, with one bit allocated as a parity bit. The reason it is called a byte is because IBM decided that is what it would call it.
8. One kilobyte = 1,024 bytes because $2^{10} = 1,024$, and that is as close as we can get to 1,000 using the binary numbering system.
9.

a. 100 (odd parity):	1000
b. 100 (even parity):	1001
c. 110101 (odd parity):	1101011
d. 110101 (even parity)	1101010
10. An abstract data type is one which we logically perceive. It does not necessarily mean that it is (physically) stored in that fashion.
11. ASCII and EBCDIC are both character coding schemes, albeit different. ASCII is used by most computer manufacturers. EBCDIC is used exclusively by IBM, which does so (essentially) because they feel like it, and can get away with it.
12. An ASCII file is one which saves each consecutive 8-bits according to the ASCII coding scheme. EBCDIC files do exist; they are files which save each consecutive 8-bits according to the ASCII coding scheme.
13. Binary files are files which do NOT save data according to ASCII (or EBCDIC) coding schemes (more discussion in later chapters).

C/C++ Programming Assignments

1. Type in, compile and run the following C/C++ program.

```

#include <stdio.h>           // Include the Standard Input-Output (IO) C Header File
#include <iostream.h>       // Include the Standard Input-Output (IO) C++ Header File
int main(void)             // main is a function name which returns an integer
{                           // This is similar to a BEGIN statement in Pascal
    char ch;               // ch is the variable where we will store an ASCII character
    ch = 'T';              // Assign the ASCII Character T to the variable ch
// Let's first print out the values using C
    printf("The values of character %c are %d decimal, %o octal and %X Hexadecimal\n",
           ch,ch,ch,ch);

/* The output from the above statement will appear as:
The values of character T are 84 decimal, 54 octal and 54 Hexadecimal */

    ch = 38;               // Assign ASCII 38 (decimal) to the variable
// Now let's print out the values using C++
    cout << "The values of character " << ch <<"are" << dec << (int) ch <<"decimal" << oct
    << (int) ch << "octal and " << hex << (int) ch << "Hexadecimal" << endl;

/* The output from the above statement will appear as:
The values of character & are 38 decimal, 56 octal and 26 Hexadecimal */

    ch = 0115;            // putting 0 (zero) in front assigns the Octal value
    printf("The values of character %c are %d decimal, %o octal and %X Hexadecimal\n",
           ch,ch,ch,ch);

/* The output from the above statement will appear as:
The values of character M are 77 decimal, 115 octal and 4D Hexadecimal */

    ch = 0X6b;            // putting 0X (Zero X) in front assigns the hexadecimal val
    cout << "The values of character " << ch <<"are" << dec << (int) ch <<"decimal" << oct
    << (int) ch << "octal and " << hex << (int) ch << "Hexadecimal" << endl;

/* The output from the above statement will appear as:
The values of character k are 107 decimal, 153 octal and 6B Hexadecimal */

    return(0);           // Return a 0 (Since the function main is of type int)
}                         // End of function main

```

Suggestion: Since all of the *printf* and *cout* statements are identical, cutting and pasting would be appropriate

2. **Modify the above program as follows: Spell out each letter of your name, alternating each letter in your name as an ASCII symbol, a decimal value, an octal value, and a hexadecimal value. Place the code needed just before the return statement.**

In other words, depending on the number of letters in your name, you would enter:

letters: 1, 5, 9, 13, Enter the input as characters
 letters: 2, 6, 10, 14, Enter the input as integers
 letters: 3, 7, 11, 15, Enter the input as octal numbers
 letters: 4, 8, 12, 16, Enter the input as hexadecimal numbers

For example, my name (Peeter Kirs) would be entered as:

```
ch = 'P'; printf("%c",ch); // The ASCII character 'P'
ch = 101; cout << ch; // The decimal representation for the character 'e'
ch = 0145; printf("%c",ch); // The Octal representation for the character 'e'
ch = 0x54; cout << ch; // The Hexadecimal representation for the character 't'
ch = 'e'; printf("%c",ch); // The ASCII Character 'e'
ch = 101; cout << ch; // The decimal representation for the character 'r'
ch = 040; printf("%c",ch); // The octal representation for the character ' ' (Space)
ch = 0X4b; cout << ch; // The Hexadecimal representation for the character
'K'
ch = 'i'; printf("%c",ch); // The ASCII character 'i'
ch = 114; cout << ch; // The decimal representation for the character 'r'
ch = 0163; printf("%c",ch); // The Octal representation for the character 's'
```

Once again, since the *printf* and *cout* statements are identical, and the assignment statements are almost identical, a cut-and-paste is suggested.

Addendum 1.1: The Standard ASCII Character Set

<u>Binary</u>	<u>Dec.</u>	<u>Oct.</u>	<u>Hex.</u>	<u>Char.</u>	<u>Description</u>	<u>Binary</u>	<u>Dec.</u>	<u>Oct.</u>	<u>Hex.</u>	<u>Char.</u>	
0000000	0	0	0	NUL	Null, Tape Feed	1000000	64	100	40	@	At Sign
0000001	1	1	1	SOH	Start of Heading	1000001	65	101	41	A	Upper Case A
0000010	2	2	2	STX	Start of Text	1000010	66	102	42	B	Upper Case B
0000011	3	3	3	ETX	End of Text	1000011	67	103	43	C	Upper Case C
0000100	4	4	4	EOT	End of Transmission	1000100	68	104	44	D	Upper Case D
0000101	5	5	5	ENQ	Enquiry	1000101	69	105	45	E	Upper Case E
0000110	6	6	6	ACK	Acknowledge	1000110	70	106	46	F	Upper Case F
0000111	7	7	7	BEL	Ring Bell	1000111	71	107	47	G	Upper Case G
0001000	8	10	8	BS	Backspace	1001000	72	110	48	H	Upper Case H
0001001	9	11	9	HT	Horizontal Tab	1001001	73	111	49	I	Upper Case I
0001010	10	12	A	LF	Line Feed	1001010	74	112	4A	J	Upper Case J
0001011	11	13	B	VT	Vertical Tab	1001011	75	113	4B	K	Upper Case K
0001100	12	14	C	FF	Form Feed	1001100	76	114	4C	L	Upper Case L
0001101	13	15	D	CR	Carriage Return	1001101	77	115	4D	M	Upper Case M
0001110	14	16	E	SO	Shift Out	1001110	78	116	4E	N	Upper Case N
0001111	15	17	F	SI	Shift In	1001111	79	117	4F	O	Upper Case O
0010000	16	20	10	DLE	Data Link Escape	1010000	80	120	50	P	Upper Case P
0010001	17	21	11	DC1	Device Control 1	1010001	81	121	51	Q	Upper Case Q
0010010	18	22	12	DC2	Device Control 2	1010010	82	122	52	R	Upper Case R
0010011	19	23	13	DC3	Device Control 3	1010011	83	123	53	S	Upper Case S
0010100	20	24	14	DC4	Device Control 4	1010100	84	124	54	T	Upper Case T
0010101	21	25	15	NAK	Negative Acknowledge	1010101	85	125	55	U	Upper Case U
0010110	22	26	16	SYN	Synchronous Idle	1010110	86	126	56	V	Upper Case V
0010111	23	27	17	ETB	End Transmission Block	1010111	87	127	57	W	Upper Case W
0011000	24	30	18	CAN	Cancel	1011000	88	130	58	X	Upper Case X
0011001	25	31	19	EM	End of Medium	1011001	89	131	59	Y	Upper Case Y
0011010	26	32	1A	SUB	Substitute	1011010	90	132	5A	Z	Upper Case Z
0011011	27	33	1B	ESC	Escape	1011011	91	133	5B	[Left Bracket
0011100	28	34	1C	FS	File Separator	1011100	92	134	5C	\	Back Slash
0011101	29	35	1D	GS	Group Separator	1011101	93	135	5D]	Right Bracket
0011110	30	36	1E	RS	Record Separator	1011110	94	136	5E	^	Carat
0011111	31	37	1F	US	Unit Separator	1011111	95	137	5F	_	Underscore
0100000	32	40	20	SP	Blank Space	1100000	96	140	60	`	Grave Accent
0100001	33	41	21	!	Exclamation	1100001	97	141	61	a	Lower Case a
0100010	34	42	22	"	Quotation Mark	1100010	98	142	62	b	Lower Case b
0100011	35	43	23	#	Pound/Number Sign	1100011	99	143	63	c	Lower Case c
0100100	36	44	24	\$	Dollar Sign	1100100	100	144	64	d	Lower Case d
0100101	37	45	25	%	Percent Sign	1100101	101	145	65	e	Lower Case e
0100110	38	46	26	&	Ampersand	1100110	102	146	66	f	Lower Case f
0100111	39	47	27	'	Single Quote	1100111	103	147	67	g	Lower Case g
0101000	40	50	28	(Left Parentheses	1101000	104	150	68	h	Lower Case h
0101001	41	51	29)	Right Parentheses	1101001	105	151	69	i	Lower Case i
0101010	42	52	2A	*	Star/Multi. Sign	1101010	106	152	6A	j	Lower Case j
0101011	43	53	2B	+	Plus Sign	1101011	107	153	6B	k	Lower Case k
0101100	44	54	2C	,	Comma	1011000	108	154	6C	l	Lower Case l
0101101	45	55	2D	-	Minus/hyphen	1101101	109	155	6D	m	Lower Case m
0101110	46	56	2E	.	Period	1101110	110	156	6E	n	Lower Case n
0101111	47	57	2F	/	Slash	1101111	111	157	6F	o	Lower Case o
0110000	48	60	30	0	Zero	1110000	112	160	70	p	Lower Case p
0110001	49	61	31	1	One	1110001	113	161	71	q	Lower Case q
0110010	50	62	32	2	Two	1110010	114	162	72	r	Lower Case r
0110011	51	63	33	3	Three	1110011	115	163	73	s	Lower Case s
0110100	52	64	34	4	Four	1110100	116	164	74	t	Lower Case t
0110101	53	65	35	5	Five	1110101	117	165	75	u	Lower Case u
0110110	54	66	36	6	Six	1110110	118	166	76	v	Lower Case v
0110111	55	67	37	7	Seven	1110111	119	167	77	w	Lower Case w
0111000	56	70	38	8	Eight	1111000	120	170	78	x	Lower Case x
0111001	57	71	39	9	Nine	1111001	121	171	79	y	Lower Case y
0111010	58	72	3A	:	Colon	1111010	122	172	7A	z	Lower Case z
0111011	59	73	3B	;	Semi-colon	1111011	123	173	7B	{	Left Elipses
0111100	60	74	3C	<	Less Than Sign	1111100	124	174	7C		Vertical Line
0111101	61	75	3D	=	Equality Symbol	1111101	125	175	7D	}	Right Elipses
0111110	62	76	3E	>	Greater Than Sign	1111110	126	176	7E	~	Tilde
0111111	63	77	3F	?	Question Mark	1111111	127	177	7F	DEL	Delete

Addendum 1.2: The Extended ASCII Character Set

Binary	Dec.	Oct.	Hex.	Char.	Binary	Dec.	Oct.	Hex.	Char.
10000000	128	200	80	Ç	10111010	192	300	C0	┐
10000001	129	201	81	ü	11000001	193	301	C1	└
10000010	130	202	82	é	11000010	194	302	C2	┌
10000011	131	203	83	â	11000011	195	303	C3	┐
10000100	132	204	84	ä	11000100	196	304	C4	└
10000101	133	205	85	à	11000101	197	305	C5	┌
10000110	134	206	86	ã	11000110	198	306	C6	┐
10000111	135	207	87	ç	11000111	199	307	C7	└
10001000	136	210	88	ê	11001000	200	310	C8	┌
10001001	137	211	89	è	11001001	201	311	C9	┐
10001010	138	212	8A	è	11001010	202	312	CA	└
10001011	139	213	8B	ÿ	11001011	203	313	CB	┌
10001100	140	214	8C	î	11001100	204	314	CC	┐
10001101	141	215	8D	ì	11001101	205	315	CD	└
10001110	142	216	8E	Ä	11001110	206	316	CE	┌
10001111	143	217	8F	Å	11001111	207	317	CF	┐
10010000	144	220	90	É	11010000	208	320	CE	└
10010001	145	221	91	æ	11010001	209	321	CF	┌
10010010	146	222	92	Æ	11010010	210	322	D0	┐
10010011	147	223	93	ø	11010011	211	323	D1	└
10010100	148	224	94	ö	11010100	212	324	D2	┌
10010101	149	225	95	ò	11010101	213	325	D3	┐
10010110	150	226	96	û	11010110	214	326	D4	└
10010111	151	227	97	ù	11010111	215	327	D5	┌
10011000	152	230	98	ÿ	11011000	216	330	D6	┐
10011001	153	231	99	Ö	11011001	217	331	D7	└
10011010	154	232	9A	Ü	11011010	218	332	D8	┌
10011011	155	233	9B	ç	11011011	219	333	D9	┐
10011100	156	234	9C	£	11011100	220	334	DA	└
10011101	157	235	9D	¥	11011101	221	335	DB	┌
10011110	158	236	9E	₣	11011110	222	336	DE	┐
10011111	159	237	9F	f	11011111	223	337	DF	└
10100000	160	240	A0	á	11100000	224	340	E0	α
10100001	161	241	A1	í	11100001	225	341	E1	β
10100010	162	242	A2	ó	11100010	226	342	E2	Γ
10100011	163	243	A3	ú	11100011	227	343	E3	Π
10100100	164	244	A4	ñ	11100100	228	344	E4	Σ
10100101	165	245	A5	Ñ	11100101	229	345	E5	σ
10100110	166	246	A6	ª	11100110	230	346	E6	μ
10100111	167	247	A7	º	11100111	231	347	E7	τ
10101000	168	250	A8	¿	11101000	232	350	E8	φ
10101001	169	251	A9	┐	11101001	233	351	E9	θ
10101010	170	252	AA	└	11101010	234	352	EA	Ω
10101011	171	253	AB	½	11101011	235	353	EB	δ
10101100	172	254	AC	¼	11101100	236	354	EC	∞
10101101	173	255	AD	;	11101101	237	355	ED	φ
10101110	174	256	AE	«	11101110	238	356	EE	ε
10101111	175	257	AF	»	11101111	239	357	EF	∩
10110000	176	260	B0	█	11110000	240	360	F0	≡
10110001	177	261	B1	█	11110001	241	361	F1	±
10110010	178	262	B2	█	11110010	242	362	F2	≥
10110011	179	263	B3	┐	11110011	243	363	F3	≤
10110100	180	264	B4	└	11110100	244	364	F4	┌
10110101	181	265	B5	┌	11110101	245	365	F5	┐
10110110	182	266	B6	┐	11110110	246	366	F6	÷
10110111	183	267	B7	└	11110111	247	367	F7	≈
10111000	184	270	B8	┐	11111000	248	370	F8	°
10111001	185	271	B9	└	11111001	249	371	F9	·
10111010	186	272	BA	┐	11111010	250	372	FA	·
10111011	187	273	BB	└	11111011	251	373	FB	√
10111100	188	274	BC	┐	11111100	252	374	FC	n
10111101	189	275	BD	└	11111101	253	375	FD	²
10111110	190	276	BE	┐	11111110	254	376	FE	█
10111111	191	277	BF	└	11111111	255	377	FF	█

Addendum 1.3: ASCII v. EBCDIC: Differences in BOLD; Unique items also Underlined

<u>Dec.</u>	<u>ASCII</u>	<u>Description</u>	<u>EBCDIC</u>	<u>Description</u>	<u>Dec.</u>	<u>ASCII</u>	<u>Description</u>	<u>EBCDIC</u>	<u>Description</u>
0	NUL	Null, Tape Feed	NUL	Null	64	@	At Sign	SP	Blank Space
1	SOH	Start of Heading	SOH	Start of Heading	65	A	Upper Case A		
2	STX	Start of Text	STX	Start of Text	66	B	Upper Case B		
3	ETX	End of Text	ETX	End of Text	67	C	Upper Case C		
4	EOT	End of Trans.	PF	Punch Off	68	D	Upper Case D		
5	ENQ	Enquiry	HT	Horizontal Tab	69	E	Upper Case E		
6	ACK	Acknowledge	LC	Lower Case	70	F	Upper Case F		
7	BEL	Ring Bell	DEL	Delete	71	G	Upper Case G		
8	BS	Backspace			72	H	Upper Case H		
9	HT	Horizontal Tab			73	I	Upper Case I		
10	LF	Line Feed			74	J	Upper Case J	¢	Cents Sign
11	VT	Vertical Tab	SMM	Start Man. Mess	75	K	Upper Case K	.	Period/Decimal
12	FF	Form Feed	FF	Form Feed	76	L	Upper Case L		
13	CR	Carriage Return	CR	Carriage Return	77	M	Upper Case M	<	Less Than Sign
14	SO	Shift Out		SO Shift Out	78	N	Upper Case N	(Left Parenthesis
15	SI	Shift In		SI Shift In	79	O	Upper Case O	+	Plus Sign
16	DLE	Data Link Escape	DLE	Data Link Escape	80	P	Upper Case P	&	Ampersand
17	DC1	Device Control 1	DC1	Device Control 1	81	Q	Upper Case Q		
18	DC2	Device Control 2	DC2	Device Control 2	82	R	Upper Case R		
19	DC3	Device Control 3	TM	Tape Mark	83	S	Upper Case S		
20	DC4	Device Control 4	RES	Restore	84	T	Upper Case T		
21	NAK	Negative Acknowl.	NL	New Line	85	U	Upper Case U		
22	SYN	Synchronous Idle	BS	BackSpace	86	V	Upper Case V		
23	ETB	End Trans Block	IL	Idle	87	W	Upper Case W		
24	CAN	Cancel	CAN	Cancel	88	X	Upper Case X		
25	EM	End of Medium		EM End of Medium	89	Y	Upper Case Y		
26	SUB	Substitute	CC	Cursor Control	90	Z	Upper Case Z	!	Exclamation
27	ESC	Escape	CU1	Customer Use 1	91	[Left Bracket	\$	Dollar Sign
28	FS	File Separator	IFS	Interchg. File Sep.	92	\	Back Slash	*	Asterisk/Star
29	GS	Group Separator	IGS	Interchg. Group Sep.	93]	Right Bracket)	Right Parentheses
30	RS	Record Separator	IRS	Interchg. Rec. Sep.	94	^	Carat	;	Semicolon
31	US	Unit Separator	IUS	Interchg. Unit Sep.	95	_	Underscore	¬	Logical NOT
32	SP	Blank Space	DS	Digit Select	96	`	Grave Accent	-	Hyphen/Minus Sign
33	!	Exclamation	SOS	Start of Significance	97	a	Lower Case a		
34	"	Quotation Mark		FS Field Separator	98	b	Lower Case b		
35	#	Pound/No. Sign			99	c	Lower Case c		
36	\$	Dollar Sign	BYP	By-Pass	100	d	Lower Case d		
37	%	Percent Sign	LF	Line Feed	101	e	Lower Case e		
38	&	Ampersand	ETB	End Trans. Block	102	f	Lower Case f		
39	'	Single Quote	ESC	Escape	103	g	Lower Case g		
40	(Left Parentheses			104	h	Lower Case h		
41)	Right Parentheses			105	i	Lower Case i		
42	*	Star/Multi. Sign	SM	Set Mode	106	j	Lower Case j		
43	+	Plus Sign	CU2	Customer Use 2	107	k	Lower Case k	,	Comma
44	,	Comma			108	l	Lower Case l	%	Percent
45	-	Minus/hyphen	ENQ	Enquiry	109	m	Lower Case m	_	Underline/Underscore
46	.	Period	ACK	Acknowledge	110	n	Lower Case n	>	Greater Than Sign
47	/	Slash	BEL	Ring Bell	111	o	Lower Case o	?	Question Mark
48	0	Zero			112	p	Lower Case p		
49	1	One			113	q	Lower Case q		
50	2	Two	SYN	Synchronous Idle	114	r	Lower Case r		
51	3	Three			115	s	Lower Case s		
52	4	Four	PN	Punch On	116	t	Lower Case t		
53	5	Five	RS	Read Stop	117	u	Lower Case u		
54	6	Six	UC	Upper Case	118	v	Lower Case v		
55	7	Seven	EOT	End of Transmission	119	w	Lower Case w		
56	8	Eight			120	x	Lower Case x		
57	9	Nine			121	y	Lower Case y		
58	:	Colon			122	z	Lower Case z	:	Colon
59	;	Semi-colon	CU3	Customer Use 3	123	{	Left Elipses	#	Pound/No. Sign
60	<	Less Than Sign	DC4	Device Control 4	124		Vertical Line	@	At Sign
61	=	Equality Symbol	NAK	Negative Acknowl.	125	}	Right Elipses	'	Apostrophe
62	>	Greater Than Sign			126	~	Tilde	=	Equal Sign
63	?	Question Mark	SUB	Substitute	127	DEL	Delete	"	Quotation Mark

Addendum 1.3:ASCII v. EBCDIC (Cont.): Differences in BOLD; Unique items also Underlined

<u>Dec.</u>	<u>ASCII</u>	<u>EBCDIC</u>	<u>Description</u>	<u>Dec.</u>	<u>ASCII</u>	<u>EBCDIC</u>	<u>Description</u>
128	Ç			192	L		
129	ü	a	Lower Case a	193	⊥	A	Upper Case A
130	é	b	Lower Case b	194	⊥	B	Upper Case A
131	â	c	Lower Case c	195	⊥	C	Upper Case A
132	ä	d	Lower Case d	196	⊥	D	Upper Case A
133	à	e	Lower Case e	197	⊥	E	Upper Case A
134	â	f	Lower Case f	198	⊥	F	Upper Case A
135	ç	g	Lower Case g	199	⊥	G	Upper Case A
136	è	h	Lower Case h	200	⊥	H	Upper Case A
137	ë	i	Lower Case i	201	⊥	I	Upper Case A
138	è			202	⊥		
139	ï			203	⊥		
140	î			204	⊥		
141	ï			205	⊥		
142	Ä			206	⊥		
143	Å			207	⊥		
144	É			208	⊥		
145	æ	j	Lower Case j	209	⊥	J	Upper Case A
146	Æ	k	Lower Case k	210	⊥	K	Upper Case A
147	ô	l	Lower Case l	211	⊥	L	Upper Case A
148	ö	m	Lower Case m	212	⊥	M	Upper Case A
149	ò	n	Lower Case n	213	⊥	N	Upper Case A
150	û	o	Lower Case o	214	⊥	O	Upper Case A
151	ù	p	Lower Case p	215	⊥	P	Upper Case A
152	ÿ	q	Lower Case q	216	⊥	Q	Upper Case A
153	Ö	r	Lower Case r	217	⊥	R	Upper Case A
154	Û			218	⊥		
155	ç			219	■		
156	£			220	■		
157	¥			221	■		
158	€			222	■		
159	f			223	■		
160	á			224	α		
161	í			225	β		
162	ó	s	Lower Case s	226	Γ	S	Upper Case A
163	ú	t	Lower Case t	227	π	T	Upper Case A
164	ñ	u	Lower Case u	228	Σ	U	Upper Case A
165	Ñ	v	Lower Case v	229	σ	V	Upper Case A
166	ª	w	Lower Case w	230	μ	W	Upper Case A
167	º	x	Lower Case x	231	τ	X	Upper Case A
168	¿	y	Lower Case y	232	φ	Y	Upper Case A
169	ƒ	z	Lower Case z	233	⊖	Z	Upper Case A
170	¬			234	Ω		
171	½			235	δ		
172	¼			236	∞		
173	ı			237	φ		
174	«			238	ε		
175	»			239	∩		
176	⋮			240	≡	0	Zero
177	⋮			241	±	1	One
178	⋮			242	≥	2	Two
179	⋮			243	≤	3	Three
180	⋮			244	⌈	4	Four
181	⋮			245	⌋	5	Five
182	⋮			246	÷	6	Six
183	⋮			247	≈	7	Seven
184	⋮			248	°	8	Eight
185	⋮			249	·	9	Nine
186	⋮			250	·		
187	⋮			251	√		
188	⋮			252	ⁿ		
189	⋮			253	²		
190	⋮			254	■		
191	⋮			255	FF		