

# Information Representation and Storage

Norman Matloff  
University of California at Davis  
©2001-2007, N. Matloff

March 22, 2007

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Bits and Bytes</b>	<b>3</b>
2.1	“Binary Digits” . . . . .	3
2.2	Hex Notation . . . . .	3
2.3	There Is No Such Thing As “Hex” Storage at the Machine Level! . . . . .	5
<b>3</b>	<b>Main Memory Organization</b>	<b>5</b>
3.1	Bytes, Words and Addresses . . . . .	5
3.1.1	The Basics . . . . .	5
3.1.2	Word Addresses . . . . .	6
3.1.3	“Endian-ness” . . . . .	6
3.1.4	Other Issues . . . . .	7
<b>4</b>	<b>Representing Information as Bit Strings</b>	<b>9</b>
4.1	Representing Integer Data . . . . .	9
4.2	Representing Real Number Data . . . . .	12
4.2.1	“Toy” Example . . . . .	12
4.2.2	IEEE Standard . . . . .	12
4.3	Representing Character Data . . . . .	14
4.4	Representing Machine Instructions . . . . .	15
4.5	What Type of Information is Stored Here? . . . . .	15
<b>5</b>	<b>Examples of the Theme, “There Are No Types at the Hardware Level”</b>	<b>16</b>
5.1	Example . . . . .	16
5.2	Example . . . . .	17
5.3	Example . . . . .	18
5.4	Example . . . . .	18

5.5	Example . . . . .	19
5.6	Example . . . . .	19
<b>6</b>	<b>Visual Display</b>	<b>20</b>
6.1	The Basics . . . . .	20
6.2	Non-English Text . . . . .	21
6.3	It's the Software, Not the Hardware . . . . .	21
6.4	Text Cursor Movement . . . . .	21
6.5	Mouse Actions . . . . .	22
6.6	Display of Images . . . . .	22
<b>7</b>	<b>There's Really No Such Thing As "Type" for Disk Files Either</b>	<b>22</b>
7.1	Disk Geometry . . . . .	22
7.2	Definitions of "Text File" and "Binary File" . . . . .	23
7.3	Programs That Access of Text Files . . . . .	24
7.4	Programs That Access Binary Files . . . . .	24
<b>8</b>	<b>Storage of Variables in HLL Programs</b>	<b>25</b>
8.1	What Are HLL Variables, Anyway? . . . . .	25
8.2	Order of Storage . . . . .	25
8.2.1	Scalar Types . . . . .	26
8.2.2	Complex Data Structures . . . . .	26
8.2.3	Pointer Variables . . . . .	27
8.3	Local Variables . . . . .	28
8.4	Variable Names and Types Are Imaginary . . . . .	29
8.5	Segmentation Faults and Bus Errors . . . . .	30
<b>A</b>	<b>ASCII Table</b>	<b>31</b>
<b>B</b>	<b>An Example of How One Can Exploit Big-Endian Machines for Fast Character String Sorting</b>	<b>33</b>
<b>C</b>	<b>How to Inspect the Bits of a Floating-Point Variable</b>	<b>33</b>

## 1 Introduction

A computer can store many types of information. A high-level language (HLL) will typically have several data types, such as the C/C++ language's **int**, **float**, and **char**. Yet a computer can not directly store any of these data types. Instead, a computer only stores 0s and 1s. Thus the question arises as to how one can represent the abstract data types of C/C++ or other HLLs in terms of 0s and 1s. What, for example, does a **char** variable look like when viewed from “under the hood”?

A related question is how we can use 0s and 1s to represent our program itself, meaning the machine language instructions that are generated when our C/C++ or other HLL program is compiled. In this chapter, we will discuss how to represent various types of information in terms of 0s and 1s. And, in addition to this question of *how* items are stored, we will also begin to address the question of *where* they are stored, i.e. where they are placed within the structure of a computer's main memory.

## 2 Bits and Bytes

### 2.1 “Binary Digits”

The 0s and 1s used to store information in a computer are called **bits**. The term comes from **binary digit**, i.e. a digit in the base-2 form of a number (though once again, keep in mind that not all kinds of items that a computer stores are numeric). The physical nature of bit storage, such as using a high voltage to represent a 1 and a low voltage to represent a 0, is beyond the scope of this book, but the point is that every piece of information must be expressed as a string of bits.

For most computers, it is customary to label individual bits within a bit string from right to left, starting with 0. For example, in the bit string 1101, we say Bit 0 = 1, Bit 1 = 0, Bit 2 = 1 and Bit 3 = 1.

If we happen to be using an n-bit string to represent a nonnegative integer, we say that Bit n-1, i.e. the leftmost bit, is the most significant bit (MSB). To see why this terminology makes sense, think of the base-10 case. Suppose the price of an item is \$237. A mistake by a sales clerk in the digit 2 would be much more serious than a mistake in the digit 7, i.e. the 2 is the most significant of the three digits in this price. Similarly, in an n-bit string, Bit 0, the rightmost bit, is called the least significant bit (LSB).

A bit is said to be **set** if it is 1, and **cleared** if it is 0.

A string of eight bits is usually called a **byte**. Bit strings of eight bits are important for two reasons. First, in storing characters, we typically store each character as an 8-bit string. Second, computer storage cells are typically composed of an integral number of bytes, i.e. an even multiple of eight bits, with 16 bits and 32 bits being the most commonly encountered cell sizes.

The whimsical pioneers of the computer world extended the pun, “byte” to the term **nibble**, meaning a 4-bit string. So, each hex digit (see below) is called a nibble.

### 2.2 Hex Notation

We will need to define a “shorthand” notation to use for writing long bit strings. For example, imagine how cumbersome it would be for us humans to keep reading and writing a string such as 1001110010101110. So, let us agree to use **hexadecimal** notation, which consists of grouping a bit string into 4-bit substrings, and then giving a single-character name to each substring.

For example, for the string 1001110010101110, the grouping would be

1001 1100 1010 1110

Next, we give a name to each 4-bit substring. To do this, we treat each 4-bit substring as if it were a base-2 number. For example, the leftmost substring above, 1001, is the base-2 representation for the number 9, since

$$1 \cdot (2^3) + 0 \cdot (2^2) + 0 \cdot (2^1) + 1 \cdot (2^0) = 9,$$

so, for convenience we will call that substring “9.” The second substring, 1100, is the base-2 form for the number 12, so we will call it “12.” However, we want to use a single-character name, so we will call it “c,” because we will call 10 “a,” 11 “b,” 12 “c,” and so on, until 15, which we will call “f.”

In other words, we will refer to the string 1001110010101110 as 0x9cae. This is certainly much more convenient, since it involves writing only 4 characters, instead of 16 0s and 1s. However, keep in mind that we are doing this only as a quick shorthand form, for use by us humans. The computer is storing the string in its original form, 1001110010101110, *not* as 0x9cae.

We say 0x9cae is the hexadecimal, or “hex,” form of the the bit string 1001110010101110. Often we will use the C-language notation, prepending “0x” to signify hex, in this case 0x9cae.

Recall that we use bit strings to represent many different types of information, with some types being numeric and others being nonnumeric. If we happen to be using a bit string as a nonnegative number, then the hex form of that bit string has an additional meaning, namely the base-16 representation of that number. For example, the above string 1001110010101110, if representing a nonnegative base-2 number, is equal to

$$\begin{aligned} 1(2^{15}) &+ 0(2^{14}) + 0(2^{13}) + 1(2^{12}) + 1(2^{11}) + 1(2^{10}) + 0(2^9) + 0(2^8) \\ &+ 1(2^7) + 0(2^6) + 1(2^5) + 0(2^4) + 1(2^3) + 1(2^2) + 1(2^1) + 0(2^0) = 40,110. \end{aligned}$$

If the hex form of this bit string, 0x9cae, is treated as a base-16 number, its value is

$$9(16^3) + 12(16^2) + 10(16^1) + 14(16^0) = 40,110,$$

verifying that indeed the hex form is the base-16 version of the number. That is in fact the origin of the term “hexadecimal,” which means “pertaining to 16.” [But there is no relation of this name to the fact that in this particular example our bit string is 16 bits long; we will use hexadecimal notation for strings of any length.] The fact that the hex version of a number is also the base-16 representation of that number comes in handy in converting a binary number to its base-10 form. We *could* do such conversion by expanding the powers of 2 as above, but it is much faster to group the binary form into hex, and then expand the powers of 16, as we did in the second equation.

The opposite conversion—from base-10 to binary—can be expedited in the same way, by first converting from base-10 to base-16, and then degrouping the hex into binary form. The conversion of decimal to base-16 is done by repeatedly dividing by 16 until we get a quotient less than 16; the hex digits then are obtained as the remainders and the very last quotient. To make this concrete, let’s convert the decimal number 21602 to binary:

```
Divide 21602 by 16, yielding 1350, remainder 2.
Divide 1350 by 16, yielding 84, remainder 6.
Divide 84 by 16, yielding 5, remainder 4.
The hex form of 21602 is thus 5462.
The binary form is thus 0101 0100 0110 0010, i.e. 0101010001100010.
```

### 3 MAIN MEMORY ORGANIZATION There Is No Such Thing As “Hex” Storage at the Machine Level!

The main ingredient here is the repeated division by 16. By dividing by 16 again and again, we are building up powers of 16. For example, in the line

```
Divide 1350 by 16, yielding 84, remainder 6.
```

above, that is our second division by 16, so it is a *cumulative* division by  $16^2$ . [Note that this is why we are dividing by 16, not because the number has 16 bits.]

#### 2.3 There Is No Such Thing As “Hex” Storage at the Machine Level!

Remember, hex is merely a convenient notation **for us humans**. It is wrong to say something like “The machine stores the number in hex,” “The compiler converts the number to hex,” and so on. **It is crucial that you avoid this kind of thinking, as it will lead to major misunderstandings later on.**

## 3 Main Memory Organization

During the time a program is executing, both the program’s data and the program itself, i.e. the machine instructions, are stored in main memory. In this section, we will introduce main memory structure. (We will usually refer to main memory as simply “memory.”)

### 3.1 Bytes, Words and Addresses

#### 3.1.1 The Basics

Memory (this means RAM/ROM) can be viewed as a long string of consecutive bytes. Each byte has an identification number, called an **address**. Again, an address is just an “i.d. number,” like a Social Security Number identifies a person, a license number identifies a car, and an account number identifies a bank account. Byte addresses are consecutive integers, so that the memory consists of Byte 0, Byte 1, Byte 2, and so on.

On each machine, a certain number of consecutive bytes is called a **word**. The number of bytes or bits (there are eight times as many bits as bytes, since a byte consists of eight bits) in a word in a given machine is called the machine’s **word size**. This is usually defined in terms of the size of number which the CPU addition circuitry can handle, which in recent years has typically been 32 bits. In other words, the CPU’s adder inputs two 32-bit numbers, and outputs a 32-bit sum, so we say the word size is 32 bits.

Most CPUs popular today have 32-bit or 64-bit words. As of December 2006, the trend is definitely toward the latter, with many desktop PCs having 64-bit words.

Early members of the Intel CPU family had 16-bit words, while the later ones were extended to 32-bit and then 64-bit size. In order to ensure that programs written for the early chips would run on the later ones, Intel designed the later CPUs to be capable of running in several modes, one for each bit size.

Note carefully that most machines do not allow overlapping words. That means, for example, that on a 32-bit machine, Bytes 0-3 will form a word and Bytes 4-7 will form a word, but Bytes 1-4 do NOT form a word. If your program tries to access the “word” consisting of Bytes 1-4, it may cause an execution error. On UNIX systems, for instance, you may get the error message “bus error.” However, an exception to this is Intel chips, which do not require alignment on word boundaries like this.



As we will discuss in detail later, compilers usually choose to store **int** variables one per word, and **char** variables one per byte. So, in this little program, **X** will occupy four bytes on a 32-bit machine, which we assume here; **PC** will point to one byte.

Suppose for example that **X** is in memory word 4000, so **&X** is 4000. Then **PC** will be 4000 too. Word 4000 consists of bytes 4000, 4001, 4002 and 4003. Since **X** is 1, i.e.  $\underbrace{000\dots00}_31\ 0s\ 1$ , one of those four bytes will contain 1 and the others will contain 0. The 1 will be in byte 4000 if and only if the machine is little-endian. In the `return` line, **PC** is pointing to byte 4000, so the return value will be either 1 or 0, depending on whether the machine is little- or big-endian, just what we wanted.

Note that within a byte there is no endian-ness issue. Remember, the endian-ness issue is defined at the word level, in terms of addresses of bytes within words; the question at hand is, “Which byte within a word is treated as most significant—the lowest-numbered-address byte, or the highest one?” This is because there is no addressing of bits within a byte, thus no such issue at the byte level.

Note that a C compiler treats hex as base-16, and thus the endian-ness of the machine will be an issue. For example, suppose we have the code

```
int Z = 0x12345678;
```

and **&Z** is 240.

As we will see later, compilers usually store an **int** variable in one word. So, **Z** will occupy Bytes 240, 241, 242 and 243. So far, all of this holds independent of whether the machine is big- or little-endian. But the endian-ness will affect which bytes of **Z** are stored in those four addresses.

On a little-endian machine, the byte 0x78, for instance, will be stored in location 240, while on a big-endian machine it would be in location 243.

Similarly, a call to **printf()** with **%x** format will report the highest-address byte first on a little-endian machine, but on a big-endian machine the call would report the lowest-address byte first. The reason for this is that the C standard was written with the assumption that one would want to use **%** format only in situations in which the programmer intends the quantity to be an integer. Thus the endian-ness will be a factor. This is a very important point to remember if you are using a call to **printf()** with **%x** format to determine the actual bit contents of a word which might not be intended as an integer.

There are some situations in which one can exploit the endian-ness of a machine. An example is given in Appendix B.

### 3.1.4 Other Issues

Many machines insist that words be **aligned**. In a 32-bit machine, this would mean that words only begin at addresses which are multiples of 4 (32 bits is 4 bytes). Thus Bytes 568-571 would form Word 568, but Bytes 569-572 would not be considered a word.

As we saw above, the address of a word is defined to be the address of its lowest-numbered byte. This presents a problem: How can we specify that we want to access, say, Byte 52 instead of Word 52? The answer is that for machine instruction types which allow both byte and word access (some instructions do, others do not), the instruction itself will indicate whether we want to access Byte *x* or Word *x*.

For example, we mentioned earlier that the Intel instruction 0xc7070100 in 16-bit mode puts the value 1 into a certain “cell” of memory. Since we now have the terms *word* and *byte* to work with, we can be more specific than simply using the word *cell*: The instruction 0xc7070100 puts the value 1 into a certain *word*

of memory; by contrast, the instruction 0xc60701 puts the value 1 into a certain *byte* of memory. You will see the details in later chapters, but for now you can see that differentiating between byte access and word access *is* possible, and is indicated in the bit pattern of the instruction itself.

Note that the word size determines capacity, depending on what type of information we wish to store. For example:

- (a) Suppose we are using an  $n$ -bit word to store a nonnegative integer. Then the range of numbers that we can store will be 0 to  $2^n - 1$ , which for  $n = 16$  will be 0 to 65,535, and for  $n = 32$  will be 0 to 4,294,967,295.
- (b) If we are storing a signed integer in an  $n$ -bit word, then the information presented in Section 4.1 will show that the range will be  $-2^{n-1}$  to  $2^{n-1} - 1$ , which will be -32,768 to +32,767 for 16-bit words, and -2,147,483,648 to +2,147,483,647 for 32-bit words.
- (c) Suppose we wish to store characters. Recall that an ASCII character will take up seven bits, not eight. But it is typical that the seven is “rounded off” to eight, with 1 bit being left unused (or used for some other purpose, such as a technique called **parity**, which is used to help detect errors). In that case, machines with 16-bit words can store two characters per word, while 32-bit machines can store four characters per word.
- (d) Suppose we are storing machine instructions. Some machines use a fixed instruction length, equal to the word size. These are the so-called RISC machines, to be discussed in a future unit. On the other hand, most older machines have instructions are of variable lengths.

On earlier Intel machines, for instance, instructions were of lengths one to six bytes (and the range has grown much further since then). Since the word size on those machines was 16 bits, i.e. two bytes, we see that a memory word might contain two instructions in some cases, while in some other cases an instruction would be spread out over several words. The instruction 0xc7070100 mentioned earlier, for example, takes up four bytes (count them!), thus two words of memory.<sup>1</sup>

It is helpful to make an analogy of memory cells (bytes or words) to bank accounts, as mentioned above. Each individual bank account has an account number and a balance. Similarly, each memory has its address and its contents.

As with anything else in a computer, an address is given in terms of 0s and 1s, i.e. as a base-2 representation of an unsigned integer. The number of bits in an address is called the **address size**. Among earlier Intel machines, the address size grew from 20 bits on the models based on the 8086 CPU, to 24 bits on the 80286 model, and then to 32 bits for the 80386, 80486 and Pentium. The current trend is to 64 bits.

The address size is crucial, since it puts an upper bound on how many memory bytes our system can have. If the address size is  $n$ , then addresses will range from 0 to  $2^n - 1$ , so we can have at most  $2^n$  bytes of memory in our system. It is similar to the case of automobile license plates. If for example, license plates in a certain state consist of three letters and three digits, then there will be only  $26^3 10^3 = 17,560,000$  possible plates. That would mean we could have only 17,560,000 cars and trucks in the state.

Keep in mind that an address is considered as unsigned integer. For example, suppose our address size is, to keep the example simple, four bits. Then the address 1111 is considered to be +15, not -1.

#### IMPORTANT NOTATION:

<sup>1</sup>Intel machines today are still of the CISC type.

## 4 REPRESENTING INFORMATION AS BIT STRINGS

We will use the notation  $c()$  to mean “contents of,” e.g.  $c(0x2b410)$  means the contents of memory word  $0x2b410$ . Keep this in mind, as we will use it throughout the course

Today it is customary to design machines with address size equal to word size. To see why this makes sense, consider this code:

```
int X, *P;  
...  
P = &X;
```

The variable **P** is a pointer and thus contains an address. But **P** is a variable in its own right, and thus will be stored in some word. For example, we may have **&X** and **P** equal to 200 and 344, respectively. Then we will have  $c(344) = 200$ , i.e. an address will be stored in a word. So it makes sense to have address size equal to word size. ✓

## 4 Representing Information as Bit Strings

We may now address the questions raised at the beginning of the chapter. How can the various abstract data types used in HLLs, and also the computer’s machine instructions, be represented using strings of 0s and 1s?<sup>2</sup>

### 4.1 Representing Integer Data

Representing nonnegative integer values is straightforward: We just use the base-2 representation, such as 010 for the number +2. For example, the C/C++ language data type **unsigned int** (also called simply **unsigned**) interprets bit strings using this representation.

But what about integers which can be either positive or negative, i.e. which are signed? For example, what about the data type **int** in C?

Suppose for simplicity that we will be using 3-bit strings to store integer variables. [Note: We will assume this size for bit strings in the next few paragraphs.] Since each bit can take on either of two values, 0 or 1, there are  $2^3 = 8$  possible 3-bit strings. So, we can represent eight different integer values. In other words, we could, for example, represent any integer from -4 to +3, or -2 to +5, or whatever. Most systems opt for a range in which about half the representable numbers are positive and about half are negative. The range -2 to +5, for example, has many more representable positive numbers than negative numbers. This might be useful in some applications, but since most computers are designed as general-purpose machines, they use integer representation schemes which are as symmetric around 0 as possible. The two major systems below use ranges of -3 to +3 and -4 to +3.

But this still leaves open the question as to which bit strings represent which numbers. The two major systems, **signed-magnitude** and **2s complement**, answer this question in different ways. Both systems store the nonnegative numbers in the same way, by storing the base-2 form of the number: 000 represents 0, 001 represents +1, 010 represents +2, and 011 represents +3. However, the two systems differ in the way they store the negative numbers, in the following way.

The signed-magnitude system stores a 3-bit negative number first as a 1 bit, followed by the base-2 representation of the magnitude, i.e. absolute value, of that number. For example, consider how the number -3 would be stored. The magnitude of this number is 3, whose base-2 representation is 11. So, the 3-bit,

---

<sup>2</sup>The word *string* here does not refer to a character string. It simply means a group of bits.

signed-magnitude representation of -3 is 1 followed by 11, i.e. 111. The number -2 would be stored as 1 followed by 10, i.e. 110, and so on. The reader should verify that the resulting range of numbers representable in three bits under this system would then be -3 to +3. The reader should also note that the number 0 actually has *two* representations, 000 and 100. The latter could be considered “-0,” which of course has no meaning, and 000 and 100 should be considered to be identical. Note too that we see that 100, which in an unsigned system would represent +4, does *not* do so here; indeed, +4 is not representable at all, since our range is -3 to +3.

The 2s complement system handles the negative numbers differently. To explain how, first think of strings of three decimal digits, instead of three bits. For concreteness, think of a 3-digit odometer or trip meter in an automobile. Think about how we could store positive and negative numbers on this trip meter, if we had the desire to do so. Since there are 10 choices for each digit (0,1,...,9), and there are three digits, there are  $10^3 = 1000$  possible patterns. So, we would be able to store numbers which are approximately in the range -500 to +500.

Suppose we can wind the odometer forward or backward with some manual control. Let us initially set the odometer to 000, i.e. set all three digits to 0. If we were to wind *forward* from 000 once, we would get 001; if we were to wind forward from 000 twice, we would get 002; and so on. So we would use the odometer pattern 000 to represent 0, 001 to represent +1, 002 to represent +2, ..., and 499 to represent +499. If we were to wind *backward* from 000 once, we would get 999; if we were to wind backward twice, we would get 998; and so on. So we would use the odometer pattern 999 to represent -1, use 998 to represent -2, ..., and use 500 to represent -500 (since the odometer would read 500 if we were to wind backward 500 times). This would give us a range -500 to +499 of representable numbers.

Getting back to strings of three binary digits instead of three decimal digits, we apply the same principle. If we wind backward once from 000, we get 111, so we use 111 to represent -1. If we wind backward twice from 000, we get 110, so 110 will be used to represent -2. Similarly, 101 will mean -3, and 100 will mean -4. If we wind backward one more time, we get 011, which we already reserved to represent +3, so -4 will be our most negative representable number. So, under the 2s complement system, 3-bit strings can represent any integer in the range -4 to +3.

This may at first seem to the reader like a strange system, but it has a very powerful advantage: We can do addition of two numbers without worrying about their signs; whether the two addends are both positive, both negative or of mixed signs, we will do addition in the same manner. For example, look at the base-10 case above, and suppose we wish to add +23 and -6. These have the “trip meter” representations 023 and 994. Adding 023 and 994 yields 1017, but since we are working with 3-digit quantities, the leading 1 in 1017 is lost, and we get 017. 017 is the “trip meter” representation of +17, so our answer is +17—exactly as it should be, since we wanted to add +23 and -6. The reason this works is that we have first wound forward 23 times (to 023) but then wound backward 6 times (the 994), for a net winding forward 17 times.

The importance of this is that in building a computer, the hardware to do addition is greatly simplified. The same hardware will work for all cases of signs of addends. For this reason, most modern computers are designed to use the 2s-complement system.

For instance, consider the example above, in which we want to find the representation of -29 in an 8-bit string. We first find the representation of +29, which is 00011101 [note that we remembered to include the three leading 0s, as specified in (a) above]. Applying Step (b) to this, we get 11100010. Adding 1, we get 11100011. So, the 8-bit 2s complement representation of -29 is 11100011. We would get this same string if we wound back from 000000 29 times, but the method here is much quicker.

This transformation is **idempotent**, meaning that it is its own inverse: If you take the 2s complement rep-

representation of a negative number  $-x$  and apply Steps (b) and (c) above, you will get  $+x$ . The reader should verify this in the example in the last paragraph: Apply Steps (b) and (c) to the bit string 11100011 representing  $-29$ , and verify that the resulting bit string does represent  $+29$ . In this way, one can find the base-10 representation of a negative number for which you have the 2s complement form.

By the way, the  $n$ -bit representation of a negative integer  $-x$  is equal to the base-2 representation of  $2^n - x$ . You can see this by noting first that the base-2 representation of  $2^n$  is  $1\underbrace{000\dots00}_{n \text{ 0s}}$ . That means that the  $n$ -bit

2s complement “representation” of  $2^n$ —it is out of range for  $n$  bits, but we can talk about its truncation to  $n$  bits—is  $\underbrace{000\dots00}_{n \text{ 0s}}$ . Since the 2s complement representation of  $-x$  is the result of winding backwards  $x$  times from  $\underbrace{000\dots00}_{n \text{ 0s}}$ , that is the result of winding backwards  $x$  times from  $2^n$ , which is  $2^n - x$ .

For example, consider 4-bit 2s complement storage. Winding backward 3 times from 0000, we get 1101 for the representation of  $-3$ . But taken as an unsigned number, 1101 is 13, which sure enough is  $2^4 - 3$ .

Although we have used the “winding backward” concept as our informal definition of 2s complement representation of negative integers, it should be noted that in actual computation—both by us humans and by the hardware—it is inconvenient to find representations this way. For example, suppose we are working with 8-bit strings, which allow numbers in the range  $-128$  to  $+127$ . Suppose we wish to find the representation of  $-29$ . We *could* wind backward from 00000000 29 times, but this would be very tedious.

Fortunately, a “shortcut” method exists: To find the  $n$ -bit 2s complement representation of a negative number  $-x$ , do the following.

- (a) Find the  $n$ -bit base-2 representation of  $+x$ , making sure to include any leading 0s.
- (b) In the result of (a), replace 0s by 1s and 1s by 0s. (This is called the **1s complement** of  $x$ .)
- (c) Add 1 to the result of (b), ignoring any carry coming out of the Most Significant Bit.

Here’s why the shortcut works: Say we have a number  $x$  whose 2s complement form we know, and we want to find the 2s complement form for  $-x$ . Let  $x'$  be the 1s complement of  $x$ , i.e. the result of interchanging 1s and 0s in  $x$ . Then  $x+x'$  is equal to  $\underbrace{111\dots11}_{n \text{ 1s}}$ , so  $x+x' = -1$ . That means that  $-x = x'+1$ , which is exactly the shortcut above.

Here very important properties of 2s-complement storage:

- (i) The range of integers which is supported by the  $n$ -bit, 2s complement representation is  $-2^{n-1}$  to  $2^{n-1} - 1$ .
- (ii) The values  $-2^{n-1}$  and  $2^{n-1} - 1$  are represented by 10000...000 and 01111...111, respectively.
- (iii) All nonnegative numbers have a 0 in Bit  $n-1$ , and all negative numbers have a 1 in that bit position.

The reader should verify these properties with a couple of example in the case of 4-bit strings.

By the way, due to the slight asymmetry in the range in (i) above, you can see that we can not use the “shortcut” method if we need to find the 2s complement representation of the number  $-2^n - 1$ ; Step (a) of that method would be impossible, since the number  $2^n - 1$  is not representable. Instead, we just use (ii).

Now consider the C/C++ statement

Sum = X + Y;

The value of Sum might become negative, even if both the values X and Y are positive. Here is why, say for 16-bit word size: With X = 28,502 and Y = 12,344, the resulting value of Sum will be -24,690. Most machines have special bits which can be used to detect such situations, so that we do not use misleading information.

Again, most modern machines use the 2s complement system for storing signed integers. We will assume this system from this point on, except where stated otherwise.

## 4.2 Representing Real Number Data

The main idea here is to use **scientific notation**, familiar from physics or chemistry, say  $3.2 \times 10^{-4}$  for the number 0.00032. In this example, 3.2 is called the **mantissa** and -4 is called the **exponent**.

The same idea is used to store real numbers, i.e. numbers which are not necessarily integers (also called **floating-point** numbers), in a computer. The representation is essentially of the form

$$m \times 2^n \tag{1}$$

with m and n being stored as individual bit strings.

### 4.2.1 “Toy” Example

Say for example we were to store real numbers as 16-bit strings, we might devote 10 bits, say Bits 15-6, to the mantissa m, and 6 bits, say Bits 5-0, to the exponent n. Then the number 1.25 might be represented as

$$5 \times 2^{-2} \tag{2}$$

that is, with m = 5 and n = -2. As a 10-bit 2s complement number, 5 is represented by the bit string 000000101, while as a 6-bit 2s complement number, -2 is represented by 111110. Thus we would store the number 1.25 as the 16-bit string 000000101 111110 i.e.

000000101111110 = 0x017e

Note the design tradeoff here: The more bits I devote to the exponent, the wider the range of numbers I can store. But the more bits I devote to the mantissa, the less roundoff error I will have during computations. Once I have decided on the string size for my machine, in this example 16 bits, the question of partitioning these bits into mantissa and exponent sections then becomes one of balancing accuracy and range.

### 4.2.2 IEEE Standard

The floating-point representation commonly used on today’s machines is a standard of the Institute of Electrical and Electronic Engineers (IEEE). For the C type **float**, the standard uses 32-bit storage, with 64 bits used for **double**. The 32-bit case, which we will study here, follows the same basic principles as with our simple example above, but it has a couple of refinements to the simplest mantissa/exponent format. It consists of a Sign Bit, an 8-bit Exponent Field, and 23-bit Mantissa Field. These fields will now be explained. Keep in mind that there will be a distinction made between the terms *mantissa* and *Mantissa Field*, and between *exponent* and *Exponent Field*.

Recall that in base-10, digits to the right of the decimal point are associated with negative powers of 10. For example, 4.38 means

$$4(10^0) + 3(10^{-1}) + 8(10^{-2}) \quad (3)$$

It is the same principle in base-2, of course, with the base-2 number 1.101 meaning

$$1(2^0) + 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3}) \quad (4)$$

that is, 1.625 in base-10.

Under the IEEE format, the mantissa must be in the form  $\pm 1.x$ , where ‘x’ is some bit string. In other words, the absolute value of the mantissa must be a number between 1 and 2. The number 1.625 is 1.101 in base-2, as seen above, so it already has this form. Thus we would take the exponent to be 0, that is, we would represent 1.625 as

$$1.101 \times 2^0 \quad (5)$$

What about the number 0.375? In base-2 this number is 0.011, so we *could* write 0.375 as

$$0.011 \times 2^0 \quad (6)$$

but again, the IEEE format insists on a mantissa of the form  $\pm 1.x$ . So, we would write 0.375 instead as

$$1.1 \times 2^{-2} \quad (7)$$

which of course is equivalent, but the point is that it fits IEEE’s convention.

Now since that convention requires that the leading bit of the mantissa be 1, there is no point in storing it! Thus the Mantissa Field only contains the bits to the right of that leading 1, so that the mantissa consists of  $\pm 1.x$ , where ‘x’ means the bits stored in the Mantissa field. The sign of the mantissa is given by the Sign Bit, 0 for positive, 1 for negative.<sup>3</sup> The circuitry in the machine will be set up so that it restores the leading “1.” at the time a computation is done, but meanwhile we save one bit per **float**.<sup>4</sup>

Note that the Mantissa Field, being 23 bits long, represents the fractional portion of the number to 23 “decimal places,” i.e. 23 binary digits. So for our example of 1.625, which is 1.101 base 2, we have to write 1.101 as 1.10100000000000000000000.<sup>5</sup> So the Mantissa Field here would be 10100000000000000000000. The Exponent Field actually does *not* directly contain the exponent; instead, it stores the exponent plus a **bias** of 127. The Exponent Field itself is considered as an 8-bit unsigned number, and thus has values ranging from 0 to 255. However, the values 0 and 255 are reserved for “special” quantities: 0 means that the floating-point number is 0, and 255 means that it is in a sense “infinity,” the result of dividing by 0, for example. Thus the Exponent Field has a range of 1 to 254, which after accounting for the bias term mentioned above means that the exponent is a number in the range -126 to +127 ( $1-127 = -126$  and  $254-127 = +127$ ).

<sup>3</sup>Again, keep in mind the distinction between the mantissa and the Mantissa field. Here the mantissa is  $\pm 1.x$  while the Mantissa field is just x.

<sup>4</sup>This doesn’t actually make storage shorter; it simply gives us an extra bit position to use otherwise, thus increasing accuracy.

<sup>5</sup>Note that trailing 0s do not change things in the fractional part of a number. In base 10, for instance, the number 1.570000 is the same as the number 1.57.

Note that the floating-point number is being stored is (except for the sign) equal to

$$(1 + M/2^{23}) \times 2^{(E-127)} \quad (8)$$

where M is the Mantissa and E is the Exponent. Make sure you agree with this.

With all this in mind, let us find the representation for the example number 1.625 mentioned above. We found that the mantissa is 1.101 and the exponent is 0, and as noted earlier, the Mantissa Field is 1010000000000000000000. The Exponent Field is  $0 + 127 = 127$ , or in bit form, 01111111.

The Sign Bit is 0, since 1.625 is a positive number.

So, how are the three fields then stored altogether in one 32-bit string? Well, 32 bits fill four bytes, say at addresses n, n+1, n+2 and n+3. The format for storing the three fields is then as follows:

- Byte n: least significant eight bits of the Mantissa Field
- Byte n+1: middle eight bits of the Mantissa Field
- Byte n+2: least significant bit of the Exponent Field, and most significant seven bits of the Mantissa Field
- Byte n+3: Sign Bit, and most significant seven bits of the Exponent Field

Suppose for example, we have a variable, say T, of type **float** in C, which the compiler has decided to store beginning at Byte 0x304a0. If the current value of T is 1.625, the bit pattern will be

```
Byte 0x304a0: 0x00; Byte 0x304a1: 0x00; Byte 0x304a2: 0xd0; Byte
0x304a3: 0x3f
```

The reader should also verify that if the four bytes' contents are 0xe1 0x7a 0x60 0x42, then the number being represented is 56.12.

Note carefully: The storage we've been discussing here is NOT base-10. It's not even base-2, though certain components within the format are base-2. It's a different kind of representation, not "base-based."

### 4.3 Representing Character Data

This is merely a matter of choosing which bit patterns will represent which characters. The two most famous systems are the American Standard Code for Information Interchange (ASCII) and the Extended Binary Coded Decimal Information Code (EBCDIC). ASCII stores each character as the base-2 form of a number between 0 and 127. For example, 'A' is stored as  $65_{10}$  (01000001 = 0x41), '%' as  $37_{10}$  (00100101 = 0x25), and so on.

A complete list of standard ASCII codes may be obtained by typing

```
man ascii
```

on most UNIX systems. Note that even keys such as Carriage Return, Line Feed, and so on, are considered characters, and have ASCII codes.

Since ASCII codes are taken from numbers in the range  $0$  to  $2^7 - 1 = 127$ , each code consists of seven bits. The EBCDIC system consists of eight bits, and thus can code 256 different characters, as opposed to

ASCII's 128. In either system, a character can be stored in one byte. The vast majority of machines today use the ASCII system.

What about characters in languages other than English? Codings exist for them too. Consider for example Chinese. Given that there are tens of thousands of characters, far more than 256, two bytes are used for each Chinese character. Since documents will often contain both Chinese and English text, there needs to be a way to distinguish the two. Big5 and Guobiao, two of the most widely-used coding systems used for Chinese, work as follows. The first of the two bytes in a Chinese character will have its most significant bit set to 1. This distinguishes it from ASCII (English) characters, whose most significant bits are 0s. The software which is being used to read (or write) the document, such as **cemacs**, a Chinese version of the famous **emacs** text editor, will inspect the high bit of a byte in the file. If that bit is 0, then the byte will be interpreted as an ASCII character; if it is 1, then that byte and the one following it will be interpreted as a Chinese character.<sup>6</sup>

#### 4.4 Representing Machine Instructions

Each computer type has a set of binary codes used to specify various operations done by the computer's **Central Processing Unit** (CPU). For example, in the Intel CPU chip family, the code 0xc7070100, i.e.

```
110001111000001110000000100000000,
```

means to put the value 1 into a certain cell of the computer's memory. The circuitry in the computer is designed to recognize such patterns and act accordingly. You will learn how to generate these patterns in later chapters, but for now, the thing to keep in mind is that a computer's machine instructions consist of patterns of 0s and 1s.

Note that an instruction can get into the computer in one of two ways:

- (a) We write a program in machine language (or assembly language, which we will see is essentially the same), directly producing instructions such as the one above.
- (b) We write a program in a high-level language (HLL) such as C, and the compiler translates that program into instructions like the one above.

[By the way, the reader should keep in mind that the compilers themselves are programs. Thus they consist of machine language instructions, though of course these instructions might have themselves been generated from an HLL source too.]

#### 4.5 What Type of Information is Stored Here?

A natural question to ask at this point would be how the computer "knows" what kind of information is being stored in a given bit string. For example, suppose we have the 16-bit string 0111010000101011, i.e. in hex form 0x742b, on a machine using an Intel CPU chip in 16-bit mode. Then

- (a) if this string is being used by the programmer to store a signed integer, then its value will be 29,739;

---

<sup>6</sup>Though in the Chinese case the character will consist of two bytes whether we use the Big5 or Guobiao systems, with the first bit being 1 in either case, the remaining 15 bits will be the different under the Guobiao encoding than under the Big5 one.

## 5 EXAMPLES OF THE THEME, “THERE ARE NO TYPES AT THE HARDWARE LEVEL”

- (b) if this string is being by the programmer used to store characters, then its contents will be the characters ‘t’ and ‘+’;
- (c) if this string is being used by the programmer to store a machine instruction, then the instruction says to “jump” (like a **goto** in C) forward 43 bytes.

So, in this context the question raised above is,

How does the computer “know” which of the above three kinds (or other kinds) of information is being stored in the bit string 0x742b? Is it 29,739? Is it ‘t’ and ‘+’? Or is it a jump-ahead-43-bytes machine instruction?

The answer is, “The computer does *not* know!” As far as the computer is concerned, this is just a string of 16 0s and 1s, with *no* special meaning. So, the responsibility rests with the person who writes the program—he or she must remember what kind of information he or she stored in that bit string. If the programmer makes a mistake, the computer will not notice, and will carry out the programmer’s instruction, no matter how ridiculous it is. For example, suppose the programmer had stored *characters* in each of two bit strings, but forgets this and mistakenly thinks that he/she had stored *integers* in those strings. If the programmer tells the computer to multiply those two “numbers,” the computer will dutifully obey!

The discussion in the last paragraph refers to the case in which we program in machine language directly. What about the case in which we program in an HLL, say C, in which the *compiler* is producing this machine language from our HLL source? In this case, during the time the compiler is translating the HLL source to machine language, the compiler must “remember” the type of each variable, and react accordingly. In other words, the responsibility for handling various data types properly is now in the hands of the compiler, rather than directly in the hands of the programmer—but still not in the hands of the hardware, which as indicated above, remains ignorant of type.

## 5 Examples of the Theme, “There Are No Types at the Hardware Level”

In the previous sections we mentioned several times that the hardware is ignorant of data type. We found that it is the software which enforces data types (or not), rather than the hardware. This is such an important point that in this section we present a number of examples with this theme. Another theme will be the issue of the roles of hardware and software, and in the latter case, the roles of your own software versus the OS and compiler.

### 5.1 Example

As an example, suppose in a C program X and Y are both declared of type **char**, and the program includes the statement

```
X += Y;
```

Of course, that statement is nonsensical. But the hardware knows nothing about type, so the hardware wouldn’t care if the compiler were to generate an add machine instruction from this statement. Thus the only gatekeeper, if any, would be the compiler. The compiler could either (a) just ignore the oddity, and

## 5 EXAMPLES OF THE THEME, “THERE ARE NO TYPES AT THE HARDWARE LEVEL” Example

generate the add instruction, or (b) refuse to generate the instruction, and issue an error message. In fact the compiler will do (a), but the main point here is that the compiler is the only possible gatekeeper here; the hardware doesn't care.

So, the compiler won't prevent us from doing the above statement, and will produce machine code from it. However, the compiler will produce different machine code depending on whether the variables are of type **int** or **char**. On an Intel-based machine, for example, there are two<sup>7</sup> forms of the addition instruction, one named **addl** which operates on 32-bit quantities and another named **addb** which works on 8-bit quantities. The compiler will store **int** variables in 32-bit cells but will store **char** variables in 8-bit cells.<sup>8</sup> So, the compiler will react to the C code

```
X += Y;
```

by generating an **addl** instruction if X and Y are both of type **int** or generating an **addb** instruction, if they are of type **char**.

*The point here, again, is that it is the software which is controlling this, not the hardware.* The hardware will obey whichever machine instructions you give it, even if they are nonsense.

### 5.2 Example

So, the machine doesn't know whether we humans intend the bit string we have stored in a 4-byte memory cell to be interpreted as an integer or as a 4-element character string or whatever. To the machine, it is just a bit string, 32 bits long.

The place the notion of types arises is at the compiler/language level, not the machine level. The C/C++ language has its notion of types, e.g. **int** and **char**, and the compiler produces machine code accordingly.<sup>9</sup> But that machine code itself does not recognize type. Again, the machine cannot tell whether the contents of a given word are being thought of by the programmer as an integer or as a 4-character string or whatever else.

For example, consider this code:

```
...
int Y; // local variable
...
strcpy(&Y, "abcd", 4);
...
```

At first, you may believe that this code would not even compile successfully, let alone run correctly. After all, the first argument to **strcpy()** is supposed to be of type **char \***, yet we have the argument as type **int \***. But the C compiler, say GCC, will indeed compile this code without error,<sup>10</sup> and the machine code will indeed run correctly, placing “abcd” into Y. The machine won't know about our argument type mismatch.

If we run the same code through a C++ compiler, say **g++**, then the compiler will give us an error message, since C++ is strongly typed. We will then be forced to use a cast:

```
strcpy((char *) &Y, "abcd", 4);
```

<sup>7</sup>More than two, actually.

<sup>8</sup>Details below.

<sup>9</sup>For example, as we saw above, the compiler will generate word-accessing machine instructions for **ints** and byte-accessing machine instructions for **chars**.

<sup>10</sup>It may give a warning message, though.

### 5.3 Example

When we say that the hardware doesn’t know types, that includes array types. Consider the following program:

```

1  main()
2
3  {  int X[5], Y[20], I;
4
5      X[0] = 12;
6      scanf("%d", &I);  // read in I = 20
7      Y[I] = 15;
8      printf("X[0] = %d\n", X[0]);  // prints out 15!
9  }
```

There appears to be a glaring problem with **Y** here. We assign 15 to **Y[20]**, even though to us humans there is no such thing as **Y[20]**; the last element of **Y** is **Y[19]**. Yet the program will indeed run without any error message, and 15 will be printed out.

To understand why, keep in mind that at the machine level there is really no such thing as an array. **Y** is just a name for the first word of the 20 words we humans think of as comprising one package here. When we write the C/C++ expression **Y[I]**, the compiler merely translates that to machine code which accesses “the location **I** ints after **Y**.”

This should make sense to you since another way to write **Y[I]** is **Y+I**. So, there is nothing syntactically wrong with the expression **Y[20]**. Now, where is “**Y[20]**”? C/C++ rules require that local variables be stored in reverse order,<sup>11</sup> i.e. **Y** first and then **X**. So, **X[0]** will follow immediately after **Y[19]**. Thus “**Y[20]**” is really **X[0]**, and thus **X[0]** will become equal to 15!

Note that the compiler could be designed to generate machine code which checks for the condition **Y > 19**. But the official C/C++ standards do not require this, and it is not usually done. In any case, the point is again that it is the software which might do this, not the hardware. Indeed, the hardware doesn’t even know that we have variables **X** and **Y**, that **Y** is an array, etc.

### 5.4 Example

As another example, consider the C/C++ library function **printf()**, which is used to write the values of program variables to the screen. Consider the C code

```

1  int W;
2  ...
3  W = -32697;
4  printf("%d %u %c\n", W, W, W);
```

again on a machine using an Intel CPU chip in 16-bit mode. We are printing the bit string in **W** to the screen three times, but are telling **printf()**, “We want this bit string to first be interpreted as a decimal signed integer (**%d**); then as a decimal unsigned integer (**%u**); then as an ASCII character (**%c**). Here is the output that would appear on the screen:

```
-32697  32839  G
```

---

<sup>11</sup>Details below.

## 5 EXAMPLES OF THE THEME, “THERE ARE NO TYPES AT THE HARDWARE LEVEL” Example

The bit string in **W** is 0x8047. Interpreted as a 16-bit 2s complement number, this string represents the number -32,697. Interpreted as an unsigned number, this string represents 32,839. If the least significant 8 bits of this string are interpreted as an ASCII character (which is the convention for %c), they represent the character ‘G’.

But remember, the key point is that the *hardware* is ignorant; it has no idea as to what type of data we intended to be stored in **W**'s memory location. The interpretation of data types was solely in the software. As far as the hardware is concerned, the contents of a memory location is just a bit string, nothing more.

### 5.5 Example

In fact, we can view that bit string without interpretation as some data type, by using the %x format in the call to **printf()**. This will result in the bit string itself being printed out (in hex notation). In other words, we are telling **printf()**, “Just tell me what bits are in this string; don’t do any interpretation.” Remember, hex notation is just that—notation, a shorthand system to make things easier on us humans, saving us the misery of writing out lengthy bit strings in longhand. So here we are just asking **printf()** to tell us what bits are in the variable being queried.<sup>12</sup>

A similar situation occurs with input. Say on a machine with 32-bit memory cells we have the statement

```
scanf ("%x", &X);
```

and we input bbc0a168.<sup>13</sup> Then we are saying, “Put 0xb, i.e. 1011, in the first 4 bits of X (i.e. the most-significant 4 bits of X), then put 1011 in the next 4 bits, then put 1100 in the next 4 bits, etc. Don’t do any interpretation of the meaning of the string; just copy these bits to the memory cell named X.” So, the memory cell X will consist of the bits 1011101111000000101000101101000.

By contrast, if we have the statement

```
scanf ("%d", &X);
```

and we input, say, 168, then we are saying, “Interpret the characters typed at the keyboard as describing the base-10 representation of an integer, then calculate that number (do  $1 \times 100 + 6 \times 10 + 8$ ), and store that number in base-2 form in X.” So, the memory cell X will consist of the bits 00000000000000000000000010101000. So, in summary, in the first of the two **scanf()** calls above we are simply giving the machine specific bits to store in X, while in the second one we are asking the machine to convert what we input into some bit string and place that in X.

### 5.6 Example

By the way, **printf()** is only a function within the C/C++ library, and thus does not itself directly do I/O. Instead, **printf()** calls another function, **write()**, which is in the operating system (and thus the call is referred to as a **system call**).<sup>14</sup> All **printf()** does is convert what we want to write to the proper sequence of bytes, and then **write()** does the actually writing to the screen.

So,

<sup>12</sup>But the endian-ness of the machine will play a role, as explained earlier.

<sup>13</sup>Note that we do not type “0x”.

<sup>14</sup>The name of the function is write() on UNIX; the name is different in other OSs.



for instance, the software would send 0x33 along the cable from the computer to the screen, and the screen then would form the proper dots to draw the character '3', called the **font** for '3'. The screen was capable only of drawing ASCII characters, no pictures etc.

By contrast, on modern graphics screens, software has control over individual pixels. For concreteness, consider an **xterm** window on UNIX systems. (You need not have used **xterm** to follow this discussion. If you have used Microsoft Windows, you can relate the discussion to that context if you wish.) The **write()** call we saw above will send the value 0x33 not directly to the screen, but rather to the **xterm** program. The latter will look up the font for '3' and then send it to the screen, and the screen hardware then displays it.

## 6.2 Non-English Text

A Chinese version of **xterm**, named **cxterm**, will do the same, except it will first sense whether the character is ASCII or Chinese, and send the appropriate font to the screen.

## 6.3 It's the Software, Not the Hardware

Once again, almost all of this process is controlled by software. For instance, in our example above in which we were storing the integer 32,697 and wished to print it to the screen, it was the software—**printf()** and the OS—which did almost all the work. The only thing the hardware did was print to the correct pixels after the OS sent it the various fonts (which are pixel patterns).

As another example, what do we mean when we say that "A PC uses the ASCII system"? We do not mean that, for example, the CPU is designed for ASCII; it isn't. We do mean that the C compiler on a PC will use ASCII for **char** variables, that the windowing software, say **xterm** will use ASCII for specifying fonts, etc.

## 6.4 Text Cursor Movement

A related question is, how does cursor movement work? For example, when you are using the **vi** text editor, you can hit the k key (or the up-arrow key) to make the cursor go up one line. How does this work?

Historically, a problem with all this was that different terminals had different ways in which to specify a given type of cursor motion. For example, if a program needed to make the cursor move up one line on a VT100 terminal, the program would need to send the characters Escape, [, and A:

```
printf("%c%c%c", 27, '[', 'A');
```

(the character code for the Escape key is 27). But for a Televideo 920C terminal, the program would have to send the ctrl-K character, which has code 11:

```
printf("%c", 11);
```

Clearly, the authors of programs like **vi** would go crazy trying to write different versions for every terminal, and worse yet, anyone else writing a program which needed cursor movement would have to "re-invent the wheel," i.e. do the same work that the That is why the Curses library was developed. The goal was to alleviate authors of cursor-oriented programs like **vi** of the need to write different code for different terminals. The programs would make calls to the API library, and the library would sort out what to do for the given terminal type. One would simply do an **#include** of the Curses header file and link in the Curses library (**-lcurses**), and then have one's program make the API calls. (To learn more about **curses**, see

my tutorial on it, at <http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Curses.pdf>.)

The library would know which type of terminal you were using, via the environment variable **TERM**. The library would look up your terminal type in its terminal database (the file **/etc/termcap**). When you, the programmer, would call the Curses API to, say, move the cursor up one line, the API would determine which character sequence was needed to make this happen.

For example, if your program wanted to clear the screen, it would not directly use any character sequences like those above. Instead, it would simply make the call

```
clear();
```

and Curses would do the work on the program's behalf.

Many dazzling GUI programs are popular today. But although the GUI programs may provide more "eye candy," they can take a long time to load into memory, and they occupy large amounts of territory on your screen. So, Curses programs such as **vi** and **emacs** are still in wide usage.<sup>15</sup>

For information on Curses programming, see <http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Curses.pdf>.

## 6.5 Mouse Actions

Each time you move the mouse, or click it, the mouse hardware sends a pulse of current to the CPU, called an **interrupt**. This causes the OS to run, and it then sends a signal to whatever application program was associated with the mouse action. That program would then act accordingly. And in the case of a mouse movement, the OS would update the position of the mouse pointer on the screen.

## 6.6 Display of Images

Programs which display images work on individual pixels at an even finer level than the fonts used for text, but the principles are the same.

# 7 There's Really No Such Thing As "Type" for Disk Files Either

**You will encounter the terms *text file* and *binary file* quite often in the computer world, so it is important to understand them well—especially they are rather misleading.**

## 7.1 Disk Geometry

Files are stored on disks. A disk is a rotating round platter with magnetized spots on its surface.<sup>16</sup> Each magnetized spot records one bit of the file.

<sup>15</sup>Interestingly, even some of those classical Curses programs have also become somewhat GUI-ish. For instance **vim**, the most popular version of **vi** (it's the version which comes with most Linux distributions, for example), can be run in **gvim** mode. There, in addition to having the standard keyboard-based operations, one can also use the mouse. One can move the cursor to another location by clicking the mouse at that point; one can use the mouse to select blocks of text for deletion or movement; etc. There are icons at the top of the editing window, for operations like Find, Make, etc.

<sup>16</sup>Colloquially people refer to the disk as a **disk drive**. However, that term should refer only to the motor which turns the disk.

## 7 THERE'S REALLY NO SUCH THING AS "TYPE" FOR DISK FILES EITHER

The magnetized spots are located on concentric rings called **tracks**. Each track is further divided in **sectors**, consisting of, say 512 bytes (4096 bits) each.

When one needs to access part of the disk, the read/write head must first be moved from its current track to the track of interest; this movement is called a **seek**. Then the head must wait until the sector of interest rotates around to the head, so that the read or write of the sector can be performed.

When a file is created, the operating system finds unused sectors on the disk in which to place the bytes of the file. The OS then records the locations (track number, sector number within track) of the sectors of the file, so that the file can be accessed by users later on. Each time a user wants to access the file, the OS will look in its records to determine where the file is on disk.

Again, the basic issue will be that the hardware does not know data types. The bits in a file are just that, bits, and the hardware doesn't know if the creator of the file intended those bits to represent numbers or characters or machine instructions or whatever.

### 7.2 Definitions of "Text File" and "Binary File"

Keep in mind that the term **binary file** is a misnomer. After all, ANY file is "binary," whether it consists of "text" or not, in the sense that it consists of bits no matter what. So what do people mean when they refer to a "binary" file?

First, let's define the term **text file** to mean a file satisfying all of the following conditions:

- (a) Each byte in the file is in the ASCII range 00000000-01111111, i.e. 0-127.
- (b) Each byte in the file is *intended* to be thought of as an ASCII character.
- (c) The file is *intended* to be broken into what we think of (and typically display) as lines. Here the term *line* is defined technically in terms of end-of-line (EOL) markers.

In UNIX, the EOL is a single byte, 0xa, while for Windows it is a pair of bytes, 0xd and 0xa. If you write the character '\n' in a C program, it will write the EOL, whichever it is for that OS.

Any file which does not satisfy these conditions has traditionally been termed a **binary file**.<sup>17</sup>

Say for example I use a text editor, say the **vim** extension of **vi**, to create a file named **FoxStory**, whose contents are

```
The quick brown fox
jumped over the fence.
```

Then **vim** will write the ASCII codes for the characters 'T', 'h', 'e' and so on (including the ASCII code for newline in the OS we are using) onto the disk. This is a text file. The first byte of the file, for instance, will be 01010100, the ASCII code for 'T', and we do intend that that 01010100 be thought of as 'T' by humans.

On the other hand, consider a JPEG image file, **FoxJumpedFence.jpg**, showing the fox jumping over the fence. The bytes in this file will represent pixels in the image, under the special format used by JPEG. It's highly likely that some of those bytes will also be 01010100, just by accident; they are certainly not intended as the letter 'T'. And lots of bytes will be in the range 10000000-11111111, i.e. 128-255, outside the ASCII range. So, this is a binary file.

Other examples of (so-called) binary files:

<sup>17</sup>Even this definition is arguably too restrictive. If we produce a non-English file which we intend as "text," it will have some non-ASCII bytes.

- audio files
- machine language files
- compressed files

### 7.3 Programs That Access of Text Files

Suppose we display the file **FoxStory** on our screen by typing<sup>18</sup>

```
cat FoxStory
```

Your screen will then indeed show the following:

```
The quick brown fox
jumped over the fence.
```

The reason this occurred is that the **cat** program did interpret the contents of the file **FoxStory** to be ASCII codes. What “interpret” means here is the following:

Consider what happens when **cat** reads the first byte of the file, ‘T’. The ASCII code for ‘T’ is 0x54 = 01010100. The program **cat** contains **printf()** calls which use the **%c** format. This format sends the byte, in this case to the screen.<sup>19</sup> The latter looks up the font corresponding to the number 0x54, which is the font for ‘T’, and that is why you see the ‘T’ on the screen.

Note also that in the example above, **cat** printed out a new line when it encountered the newline character, ASCII 12, 00001100. Again, **cat** was written to do that, but keep in mind that otherwise 00001100 is just another bit pattern, nothing special.

By contrast, consider what would happen if you were to type

```
cat FoxJumpedFence.jpg
```

The **cat** program will NOT know that **FoxJumpedFence.jpg** is not a text file; on the contrary, **cat** assumes that any file given to it will be a text file. Thus **cat** will use **%c** format and the screen hardware will look up and display fonts, etc., even though it is all meaningless garbage.

### 7.4 Programs That Access Binary Files

These programs are quite different for each application, of course, since the interpretation of the bit patterns will be different, for instance, for an image file than for a machine language file.

One point, though, is that when you deal with such files in, say, C/C++, you may need to warn the system that you will be accessing binary files. In the C library function **fopen()**, for example, to read a binary file you may need to specify “**rb**” mode. In the C++ class **ifstream** you may need to specify the mode **ios::binary**.

<sup>18</sup>The **cat** command is UNIX, but the same would be true, for instance, if we typed `type FoxStory` into a command window on a Windows machine.

<sup>19</sup>As noted earlier, in modern computer systems the byte is not directly sent to the screen, but rather to the windowing software, which looks up the font and then sends the font to the screen.

## 8 STORAGE OF VARIABLES IN HLL PROGRAMS

The main reason for this is apparently due to the Windows situation. Windows text files use ctrl-z as an end-of-file marker. (UNIX has no end-of-file marker at all, and it simply relies on its knowledge of the length of the file to determine whether a given byte is the last one or not.) Apparently if the programmer did not warn the system that a non-text file is being read, the system may interpret a coincidental ASCII 26 (ctrl-z) in the file as being the end of the file.<sup>20</sup>

## 8 Storage of Variables in HLL Programs

### 8.1 What Are HLL Variables, Anyway?

When you execute a program, both its instructions and its data are stored in memory. Keep in mind, the word *instructions* here means machine language instructions. Again, these machine language instructions were either written by the programmer directly, or they were produced by a compiler from a source file written by the programmer in C/C++ or some other HLL. Let us now look at the storage of the *data* in the C case.<sup>21</sup> In an HLL program, we specify our data via names. The compiler will assign each variable a location in memory. Generally (but not always, depending on the compiler), these locations will be consecutive.

### 8.2 Order of Storage

C compilers tend to store local variables in the reverse of the order in which they are declared. For example, consider the following program:

```
1 PrintAddr(char *VarName,int Addr)
2
3 { printf("the address of %s is %x\n",VarName,Addr); }
4
5 main()
6
7 { int X,Y,W[4];
8   char U,V;
9
10  PrintAddr("X",&X);
11  PrintAddr("Y",&Y);
12  PrintAddr("W[3]",&W[3]);
13  PrintAddr("W[0]",&W[0]);
14  PrintAddr("U",&U);
15  PrintAddr("V",&V);
16 }
```

I ran this on a 32-bit Pentium machine running the Linux operating system. The output of the program was

```
the address of X is bffffb84
the address of Y is bffffb80
the address of W[3] is bffffb7c
the address of W[0] is bffffb70
the address of U is bffffb6f
the address of V is bffffb6e
```

<sup>20</sup>I say “apparently,” because after much searching I have not been able to confirm this.

<sup>21</sup>The C language became popular in the early 1980s. It was invented by the ATT&T engineers who developed UNIX. The ensuing popularity of UNIX then led to popularity of C, even for non-UNIX users. Later it was extended to C++, by adding class structures (it was originally called “C with classes”). If you are wondering whether a certain C++ construct is also part of C, the answer is basically that if it is not something involving classes (or the `new` operator), it is C.

Note that the **ints** are 4 bytes apart while the **chars** are 1 byte apart, reflecting the sizes of these types. More on this below.

On the other hand, there is wide variation among C compilers as to how they store global variables.

### 8.2.1 Scalar Types

Virtually all C compilers store variables of type **int** in one word. This is natural, as the word size is the size of integer operands that the hardware works on.

Variables of type **float** are usually stored in 32-bit units, since that is the size of the IEEE floating-point standard. So, in modern 32-bit machines, a **float** is stored in one word. For 16-bit machines, this means that **float** variables must be stored in a pair of words.

Typically **char** variables are stored as one byte each, since they are 7 or 8 bits long and thus fit in one byte. By the way, C/C++ treats **char** as an integer type, storing a 1-byte signed integer, as opposed to the typically 4-byte **int**. This can save a lot of space in memory if one knows that 8 bits is enough for each of the integers we will store. Thus in C and even C++, the code

```
char G;
...
if (G < 0) ...
```

is legal and usable in the integer sense.

What about **int** variants such as **short** and **long**? Well, first one must note that a compiler need not treat them any differently from **ints**. GCC on Intel machines, for instance, stores a **long** in 4 bytes, just like an **int**. On the other hand, it stores a **short** in 2 bytes. It does offer a type **long long**, which it stores in 8 bytes, i.e. as a 64-bit quantity.

In the C language, the **sizeof** operator tells us how many bytes of storage the compiler will allocate for any variable type.

### 8.2.2 Complex Data Structures

As seen in an example above, array variables are generally implemented by compilers as contiguous blocks of memory. For example, the array declared as

```
int X[100];
```

would be allocated to some set of 100 consecutive words in memory.

What about two-dimensional arrays? For example, consider the four-row, six-column array

```
int G[4][6];
```

Again, this array will be implemented in a block of consecutive words of memory, more specifically 24 consecutive words, since these arrays here consist of  $4 \times 6 = 24$  elements. But in what order will those 24 elements be arranged? Most compilers use either **row-major** or **column-major** ordering. To illustrate this, let us assume for the moment that this is a global array and that the compiler stores in non-reverse order, i.e.  $G[0][0]$  first and  $G[3][5]$  last.

In row-major order, all elements of the first row are stored at the beginning of the block, then all the elements of the second row are stored, then the third row, and so on. In column-major order, it is just the opposite: All of the first column is stored first, then all of the second column, and so on.

In the example above, consider  $G[1][4]$ . With row-major order, this element would be stored as the 11th word in the block of words allocated to  $G$ . If column-major order were used, then it would be stored in the 18th word. (The reader should verify both of these statements, to make sure to understand the concept.) C/C++ compilers use the row-major system.

Advanced data types are handled similarly, again in contiguous memory locations. For instance, consider the **struct** type in C, say

```
struct S {
    int X;
    char A,B;
};
```

would on a 16-bit machine be stored in four consecutive bytes, first two for  $X$  and then one each for  $A$  and  $B$ . On a 32-bit machine, it would be stored in six consecutive bytes, since  $X$  would take up a word and thus four bytes instead of two.

Note, though, that most compilers will store complex objects like this to be **aligned on word boundaries**. Consider the above **struct**  $S$  on a 32-bit machine, and suppose we have the local declaration

```
struct S A,B;
```

$A$  and  $B$  collectively occupy 12 bytes, thus seemingly three words. But most compilers will now start an instance of  $S$  in the middle of a word. Thus although  $A$  and  $B$  will be stored contiguously in the sense that no other variables will be stored between them, they will be stored in four consecutive words, not three, with their addresses differing by 8, not 6. There will be two bytes of unused space between them.

In C++, an object of a **class** is stored in a manner similar to that of a **struct**.

### 8.2.3 Pointer Variables

We have not yet discussed how **pointer** variables are stored. *A pointer is an address*. In C, for example, suppose we have the declaration

```
int *P,X;
```

We are telling the compiler to allocate memory for two variables, one named  $P$  and the other named  $X$ . We are also telling the compiler what types these variables will have:  $X$  will be of integer type, while the “ $*$ ” says that  $P$  will be of pointer-to-integer type—meaning that  $P$  will store the address of some integer. For instance, in our program, we might have the statement

```
P = &X;
```

which would place the address of  $X$  in  $P$ . (More precisely, we should define this as the lowest-address byte of  $X$ .) If one then executed the statement

```
printf("%x\n",P);
```

the address of X would be printed.

As mentioned earlier, most machines today have equal word and address sizes. So, not only will the variable X here be stored in one word, but also the variable P would occupy a word too.

Note carefully that this would be true no matter what type P were to point to. For instance, consider the **struct S** we had above. The code

```
struct S *Q;
```

would *also* result in Q being stored in one word. An address is an address, no matter what kind of data item is stored beginning at that address..

Another aspect of C/C++ pointers which explicitly exposes the nature of storage of program objects in memory is **pointer arithmetic**. For example, suppose P is of pointer type. Then P+1, for example, points to the next consecutive object of the type to which P points.

The following example will bring together a number of different concepts we've looked at in the last few pages (be SURE to followup if you don't understand this well):

Consider the code

```
1 main()
2
3 { float A,B,C,*Q;
4
5     Q = &B;
6     Q = Q-2;
```

Suppose Q is the memory location, say, 0x278. Then C will be the next word after Q, i.e. word 0x27c, then B will be word 0x280 and A will be word 0x284.

After the first statement,

```
Q = &B;
```

is executed, we will have  $c(0x278) = 0x280$ .

Now, what happens when the second statement,

```
Q = Q-2;
```

is executed? The expression Q-2 means "2 objects before Q in memory," where the word *object* means the type of variable pointed to by Q. So, Q-2 is the place 2 **floats** before where Q is pointing in memory. From Section 8.2.1 we know that this then means 2 words before where Q is pointing in memory. In other words, Q-2 is pointing to Q itself!

Keep in mind that in C/C++, arrays are equivalent to pointer references. For example, Y[200] means the same thing as Y+200.

### 8.3 Local Variables

Compilers assign global variables to fixed locations determined at compile-time.

On the other hand, variables which are local to a given function storage on the **stack**, which is a section of memory. Thus local variables are in memory too, just like globals. However, a variable's position on the

stack won't be known until run-time. The compiler will merely specify the location of a variable relative to the top of the stack.

This is hard to understand given the background you have so far. You will understand it much better when we cover our unit on subroutines, but to give you a brief idea now, think of a recursive function `f()` which has a local variable `X`. As the function is called recursively multiple times, the stack will expand more and more, and there will be a separate space for `X` in each call. These multiple versions of `X` will occupy multiple memory locations, so you can see that `&X` cannot be predicted at compile-time.

## 8.4 Variable Names and Types Are Imaginary

When you compile a program from an HLL source file, the compiler first assigns memory locations for each declared variable.<sup>22</sup> For its own reference purposes, the compiler will set up a **symbol table** which shows which addresses the variables are assigned to. However, it is very important to note is that *the variable names in a C/C++ or other HLL program are just for the convenience of us humans*. In the machine language program produced from our HLL file by the compiler, all references to the variables are through their addresses in memory, with no references whatsoever to the original variables names.


For example, suppose we have the statement

```
X = Y + 4;
```

When the compiler produces some machine-language instructions to implement this action, the key point is that these instructions will *not* refer to `X` and `Y`. Say `X` and `Y` are stored in Words 504 and 1712, respectively. Then the machine instructions the compiler generates from the above statement will only refer to Words 504 and 1712, the locations which the compiler chose for the C/C++ variables `X` and `Y`.

For instance, the compiler might produce the following sequence of three machine language instructions:

```
copy Word 1712 to a cell in the CPU
add 4 to the contents of that cell
copy that cell to Word 504
```

There is no mention of `X` and `Y` at all! The names `X` and `Y` were just *temporary* entities, for communication between you and the compiler. The compiler chose locations for each variable, and then translated all of your C/C++ references to those variables to references to those memory locations, as you see here. 

Again, when the compiler is done compiling your program, it will simply discard the symbol table.<sup>23</sup> If say on a UNIX system you type

```
gcc x.c
```

which creates the executable file **a.out**, that file will not contain the symbol table.

There is one exception to this, though, which occurs when you are debugging a program, using a debugging tool such as GDB.<sup>24</sup> Within the debugging tool you will want to be able to refer to variables by name, rather

<sup>22</sup>These locations will not actually be used at that time. They will only be used later, when the program is actually run.

<sup>23</sup>It may retain a list of function names and names of global variables. For simplicity, though, we'll simply say that the entire table is discarded.

<sup>24</sup>You should always use a debugging tool when you are debugging a program. *Don't debug by just inserting a lot of `printf()` calls!* Use of a debugging tool will save you large amounts of time and effort. Professional software engineers use debugging tools as a matter of course. There is a lot of information on debugging, and debugging tools, on my debug Web page, at <http://heather.cs.ucdavis.edu/~matloff/debug.html>

than by memory location, so in this case you do need to tell the compiler to retain the symbol table in **a.out**.<sup>25</sup> You do so with the **-g** option, i.e.

```
gcc -g x.c
```

But even in this case, the actually machine-language portion (as opposed to the add-on section containing the symbol table) of the executable file **a.out** will not make any references to variable names.

The same comments apply to function names; they're gone too once we create **a.out**. Similarly, there is no such thing as a type of a variable at the machine level, as earlier.

## 8.5 Segmentation Faults and Bus Errors

Most machines today which are at the desktop/laptop level or higher (i.e. excluding handheld PDAs, chips inside cars and other machines, etc.) have **virtual memory** (VM) hardware. We will discuss this in another unit in detail, but the relevance here is that it can be used to detect situations in which a program tries to access a section of memory which was not allocated to it.

In a VM system, memory is (conceptually) broken into chunks called **pages**. When the operating system loads a program, say **a.out**, into memory, the **OS** also sets up a **page table**, which shows exactly which parts of memory the program is allowed to access. Then when the program is actually run, the hardware monitors each memory access, checking the page table. If the program accesses a part of memory which was not allocated to it, the hardware notices this, and switches control to the OS, which announces an error and kills in the program. This is called a **segmentation fault** in UNIX and a **general protection error** in VM Windows systems.

For example, consider the following program, which we saw in Section 4.5:

```
1 main()
2
3 { int X[5],Y[20];
4
5     X[0] = 12;
6     Y[20] = 15;
7     printf("X[0] = %d\n",X[0]); // prints out 15!
8 }
```

Say we compile this, say to **a.out**, and run it. What will happen?

As noted earlier, **Y[20]** is a perfectly legal expression here, which will turn out to be identical to **X[0]**. That's how **X[0]** becomes equal to 15.<sup>26</sup>

Even though that is clearly not what the programmer intended, the VM hardware will not detect this problem, and the program will execute without any announcement of error. The reason is that the program does not access memory that is not allocated to it.

On the other hand, if **Y[20]** above were **Y[20000]**, then the location of “**Y[20000]**” would be far outside the memory needed by the program, and thus definitely outside the memory allocated to the program.<sup>27</sup> The

<sup>25</sup>In fact, you'll need more than the symbol table. You certainly want the debugger to tell you what line of source code you are currently on, so we need to save the memory locations of machine code produced by each line of source code.

<sup>26</sup>C and C++ do not implement array **bounds checking**, i.e. the compiler will not produce code to check for this, which would slow down execution speed.

<sup>27</sup>Since the memory is allocated in units of pages, the actual amount of memory allocated will be somewhat larger than the program needs; we cannot allocate only a partial page.

## A ASCII TABLE

hardware would detect this, then jump to the OS, which would (in UNIX) shout out to us, “Segmentation fault.”

**So, remember:**

**A “seg fault” error message means that your program has tried to access a portion of memory which is outside the memory allocated to it. This is due to a buggy array index or pointer variable.**

Note that the case of bad pointer variables includes problems like the one in the classical error made by beginning C programmers:

```
1 scanf("%d", X);
```

where X is, say, of type **int**; the programmer here has forgotten the ampersand.

The first thing you should do when a seg fault occurs is to find out where in the program it occurred. **Use a debugging tool, say DDD or GDB, to do this!**

Note again that seg faults require that our machine have VM hardware, and that the OS is set up to make use of that hardware. Without both of these, an error like “Y[20000]” will NOT be detected.

Many CPU types require that word accesses be **aligned**. On Intel machines, for example, words must begin at addresses which are multiples of 4. Word 200, for instance, consists of Bytes 200, 201, 202 and 203, and Word 204 consists of Bytes 204-207, but there is no such thing as “Word 201,” consisting of Bytes 201-204. Suppose your program has code like

```
1 int X, *P;
2 ...
3 P = (int *) 201;
4 X = *P;
```

The compiler will translate the line

```
X = *P;
```

into some word-accessing machine instruction. When that instruction is executed, the CPU hardware will see that P’s value is not a multiple of 4.<sup>28</sup> The CPU will then cause a jump to the operating system, which will (for example in UNIX) announce “Bus error.”

As is the case with seg faults, you should use a debugging tool to determine where a bus error has occurred.

## A ASCII Table

This is the output of typing

```
% man ascii
```

on a UNIX machine.

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL '\0'	100	64	40	@
001	1	01	SOH	101	65	41	A
002	2	02	STX	102	66	42	B

<sup>28</sup>Note that this does not require VM capability.

## A ASCII TABLE

003	3	03	ETX	103	67	43	C
004	4	04	EOT	104	68	44	D
005	5	05	ENQ	105	69	45	E
006	6	06	ACK	106	70	46	F
007	7	07	BEL '\a'	107	71	47	G
010	8	08	BS '\b'	110	72	48	H
011	9	09	HT '\t'	111	73	49	I
012	10	0A	LF '\n'	112	74	4A	J
013	11	0B	VT '\v'	113	75	4B	K
014	12	0C	FF '\f'	114	76	4C	L
015	13	0D	CR '\r'	115	77	4D	M
016	14	0E	SO	116	78	4E	N
017	15	0F	SI	117	79	4F	O
020	16	10	DLE	120	80	50	P
021	17	11	DC1	121	81	51	Q
022	18	12	DC2	122	82	52	R
023	19	13	DC3	123	83	53	S
024	20	14	DC4	124	84	54	T
025	21	15	NAK	125	85	55	U
026	22	16	SYN	126	86	56	V
027	23	17	ETB	127	87	57	W
030	24	18	CAN	130	88	58	X
031	25	19	EM	131	89	59	Y
032	26	1A	SUB	132	90	5A	Z
033	27	1B	ESC	133	91	5B	[
034	28	1C	FS	134	92	5C	\ '\\'
035	29	1D	GS	135	93	5D	]
036	30	1E	RS	136	94	5E	^
037	31	1F	US	137	95	5F	_
040	32	20	SPACE	140	96	60	`
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	\$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27	'	147	103	67	g
050	40	28	(	150	104	68	h
051	41	29	)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

## B An Example of How One Can Exploit Big-Endian Machines for Fast Character String Sorting

The endian-ness of a machine can certainly make a difference. As an example, suppose we are writing a program to sort character strings in alphabetical order. The word “card” should come after “car” but before “den”, for instance. Suppose our machine has 32-bit word size, variables X and Y are of type **int**, and we have the code

```
strncat (&X, "card", 4);
strncat (&Y, "den ", 4);
```

Say X and Y are in Words 240 and 804, respectively. This means that Byte 240 contains 0x63, the ASCII code for ‘c’, Byte 241 contains 0x61, the code for ‘a’, byte 242 contains 0x72, the code for ‘r’ and byte 243 contains 0x64. Similarly, bytes 804-807 will contain 0x64, 0x65, 0x6e and 0x20. All of this will be true regardless of whether the machine is big- or little-endian.

But on a big-endian machine, we can actually use word subtraction to determine which of the strings “card” and “den” should precede the other alphabetically, by subtracting Word 240 from Word 804. To see that this works, note that the contents of the two words will be 0x63617264 (1667330660 decimal) and 0x64656e20 (1684368928 decimal), so the result of the subtraction will be negative, and thus we will decide that “card” is less than (i.e. alphabetically precedes) “den”—exactly what we want to happen.

The reader should pause to consider the speed advantage of such a comparison. If we did not use word subtraction, we would have to do a character-by-character comparison, that is four subtractions. (Note that these are word-based subtractions, even though we are only working with single bytes.) The arithmetic hardware works on a word basis.) Suppose for example that we are dealing with 12-character strings. We can base our sort program on comparing (up to) three pairs of words if we use word subtraction, and thus gain roughly a fourfold speed increase over a more straightforward sort which compares up to 12 pairs of characters.

We could do the same thing on a little-endian machine if we were to store character strings backwards. However, this may make programming inconvenient.

## C How to Inspect the Bits of a Floating-Point Variable

Suppose we wish to see the individual bits in a **float** variable. The code

```
printf ("%x\n", y);
```

will not work, as it would tell **printf()** to consider the bits in **y** to represent an integer, and would then print that integer in base-16. Why would this be a problem? Recall that **printf()** treats any quantity printed using **%x** format as an integer, and thus the endian-ness will play a role. That would mean the printout would be “backwards” on a little-endian machine.

Instead, we must write something like this:

```
1 float y;
2 char *p; // char type focuses on individual bytes
3 ...
4 ...
```

