The Relationship of Programming Languages to Binary Machine Code and the Computer's Digital Electronics

Robert Milton Underwood, Jr.

© 2001

Table of Contents

Section	Page
Abstract	3
Standards	5
ASCII	9
Hexadecimal	13
Translation Hierarchy	17
Character Transmission	20
Machine Architecture	21
Conclusion	22
References	24

Abstract

The digital age has allowed computers to have the vast computing power that they do. Digital technology provides far greater flexibility than can analog technology because by merely changing a set of written instructions via programming, virtually infinite uses for computers can be created. The way the computer recognizes the code created by programmers is via binary notation, which is consistent with the very nature of the computer's digital electronics.

 © 2001

The Relationship of Programming Languages to Binary Machine Code and the Computer's Digital Electronics

In terms of systems design, programming is the process of translating the system specifications that are prepared during the design phase of the system development lifecycle into program code. From a perspective of software development, programming involves the designing of useful applications. The computer's ability to process complex transactions as a result of advances in programming is astonishing. Leebaert (1995) wrote about a physicist studying immunology who works with "cellular automata," which are segments of computer code programmed to act like simple biological organisms. The cellular automata are used to simulate key parts of the human immune system in a mainframe computer, and self-contained segments of the software interact with their machine environment to fight for resources, reproduce, mutate, or flee.

Software can also be used as a tool to send a space shuttle into space. After a space shuttle is launched, the computer is constantly receiving new data – the speed, current position, and angle of procession – and making decisions based on that data to increase or decrease the fuel flow to the various thrusters that keep the space shuttle on target. While the computer appears to *make decisions*, it is actually the

programmer, or programming team, who determines the conditions under which the computer will execute one or another set of instructions. The computer does only what it has been programmed to do, and when it has been programmed so that one or more conditions will determine which instruction set to follow, it is said to have *conditional branch instructions* (Kohanski, 1998).

Regardless of the type of application, simple or complex, that is used by computers, it is helpful for programmers to understand the fundamental nature of the computer's use of binary notation. Understanding the binary nature of the computer's electronics is the understanding of how programming instructions are recognized at the system level and the way that the computer receives and stores data.

Standards

Certain standards are almost universally recognized, and as computer technology advances, standards evolve. Standards are important for the sake of consistency, and most start out as a working convenience. Almost any commonly agreed-upon measurement or procedure, however simple, is preferable to a "wilderness of private reckonings, however sophisticated" (Leebaert, 1995). A standard then becomes a prescription around which a new world is to be built. Standards change and evolve. One thing that results from the almost exponential increase of

technological complexity is that if something has reached a general working level, it is likely to be obsolete by the latest standards of the items at the cutting edge of development.

One standard that remains constant, however, is the use of binary notation for computers. Binary signals are generated by computers, and binary techniques dominate the processing of messages. Binary functions can be described as a subclass of digital functions that exists in two states only (Carne, 1999). With the binary signals of the computer, information is carried in the sequence of electronic states represented by 1s and 0s, which effectively function as states of "on" or "off."

There are several different types of binary electronic signal formats, but the type most relevant to the sending of character data from the keyboard to the computer is known as unipolar. With the unipolar binary signal format, a current represents a 1 and the absence of a current represents a 0. Generally speaking, a binary signal current can be either positive or negative, but a positive current is pertinent to the scope of this research paper[i]. The current's flow is measured in AMPS (A). The binary 1 is represented by a current

---

[i] Before solid-state memory chips were developed, computers with vacuum tubes and other discrete components actually used both negative and positive voltages. But modern solid-state technology (VLSI) does not work with any negative voltages. VSLI stands for Very Large Scale Integration, and is used to describe microchips that typically have more than 250,000 components each, or over 10,000 logic gates each. VLSI methods allow thousands of integrated circuits to be placed on each chip (Spencer, 1994).

 © 2001

of 2A signal units, and the binary 0 is represented by a current of zero signal units (Carne, 1999). The unipolar signal format is implemented as *nonreturn to zero* (NRZ). With the NRZ implementation, currents are maintained for the entire bit period (i.e., the entire time slot). In a long sequence of 1s and 0s, the power in a unipolar function is $\frac{1}{2}(2A)^2$, which equals $2A^2$ signal watts. With NRZ operation, long strings of 1s produce periods in which only positive current is generated, and long strings of 0s produce periods in which little or no current is generated.

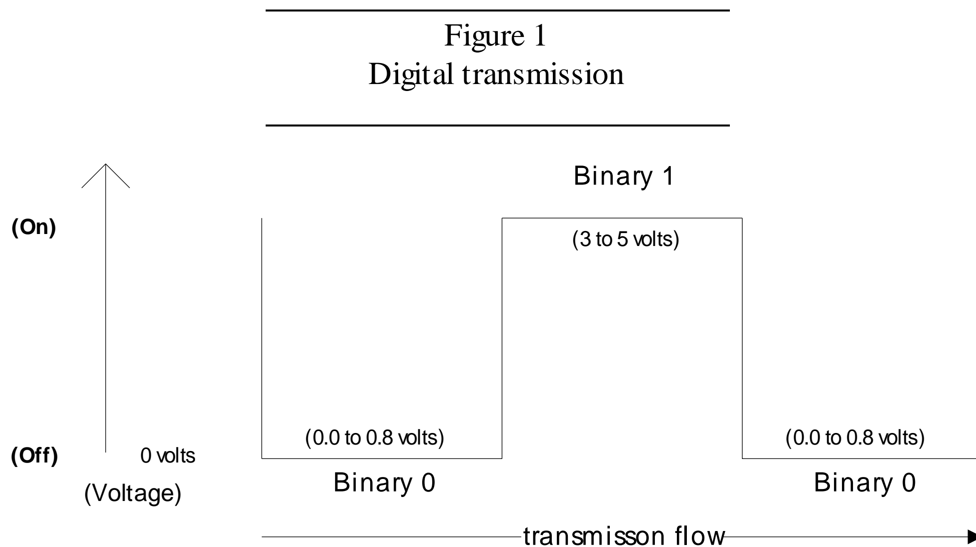
Today's computers are built on two key principles (Patterson & Hennessy, 1998): (a) instructions are represented as numbers (i.e., 1s and 0s), and (b) programs can be stored in memory and accessed just like numbers. The latter principle is referred to as the *stored-program* concept. The importance of this concept is that memory can contain the source code, the compiled machine code, data that the compiled program is using, and even the compiler software that generated the machine code. Treating numbers in the same way as data greatly simplifies both the memory hardware and the software of computer systems. Specifically, the memory technology needed for data can also be used for programs, and programs (like compilers) can translate code from a notation that is understandable for people into the code that the computer can understand.

The electronics inside a modern computer are digital. Digital electronics operate with only two voltage levels of interest: a high voltage and no (or low) voltage. All other voltages are temporary and occur while transitioning between the high and low values. This is the key reason why computers use binary numbers, since a binary system "matches the underlying abstraction inherent in the electronics" (Patterson & Hennessy, 1998). Another point of view is presented by Kohanski (1998). He wrote that to actually communicate to an electronic machine such as the computer, one must send electrical signals. The easiest signals for machines to understand are *on* and *off*, and the two symbols to represent these two signals are the numbers *1* and *0*. Therefore, each 1 or 0 is representative of an *on* or *off* state of electricity. We must therefore think of the machine language in terms of binary numbers, or as numbers in base two ($base_{two}$). Each 1 or 0 is referred to as a *binary digit* or *bit*.

The "high voltage/low voltage" description of the computer's electronics provided by Patterson and Hennessy (1998) is essentially the same as the "on/off" description provided by Kohanski (1998). Each represents the 1/0 binary notational system. But while "off" does not seem to be the equivalent of "low voltage," because semantically "low voltage" is not the same as "no voltage," they actually represent the

same thing. According to Naviwala (2001), the "high voltage" is usually around three to five volts[ii], and the "low voltage" ranges from 0.0 - 0.8 volts. So, from a transmission perspective, the electrical voltage value of the binary 0 ranges from "off" to 0.8 volts. From the theoretical perspective of the computer's binary system, the 0 is essentially the "off" position (see Figure 1).

Figure 1
Digital transmission



<u>ASCII</u>

The smallest usable grouping of bits is eight, and is known as a *byte*. A byte is equivalent to one character, and each character has a different byte representation. Each byte can be used to store a decimal number, a symbol, a character,

[ii] As was mentioned previousl,y binary 1 is represented by 2A signal units. The value of "2A" is a measure of how much current is flowing when a 1 is represented (Raad, 2001). In unipolar binary signal format, a "1" is "high voltage" which means that current is flowing through the signal. The relationship between current and voltage is defined by Ohm's law: For any circuit the electric current is directly proportional to the voltage and is inversely proportional to the resistance.

or part of a picture (Laudon & Laudon, 2000). For example, the character R (capital letter) is equivalent to 01010010 in binary. The lower case r is represented differently as the byte 01110010.

Each byte of eight bits can hold one of 256 possible values because there are exactly 256 possible combinations of eight bits. That is, there are 256 different eight-bit combinations ranging from 00000000 to 11111111. The decimal range of these 256 numbers is 0 through 255. Table 1 shows how the translation from binary values (i.e., $base_{two}$) to decimal values (i.e., $base_{ten}$) is calculated[iii].

Table 1
The values of a byte

2^7 128	2^6 64	2^5 32	2^4 16	2^3 8	2^2 4	2^1 2	2^0 1	← exponent of 2 ← decimal value	
0	0	0	0	0	0	0	0	=	0
0	0	0	0	0	0	0	1	=	1
0	0	0	0	0	0	1	0	=	2
0	0	0	0	0	0	1	1	=	3
				↕					
0	1	0	1	0	1	0	1	=	85
				↕					
1	1	1	1	1	1	1	1	=	255

As Table 1 illustrates, each of the 256 possible bytes is represented by a decimal equivalent, and 01010101 has as its decimal equivalent[iv] the number 85. Each of the 256 bytes is

---

[iii] In any number base, the value of the ith digit d is $d * Base^i$.
[iv] Although we read the values in Table 1 from left to right, the actual electronic transmission of the binary values is from right to left. The electronic transmission is sent in order of the least significant exponential position of 2

 © 2001

represented by a decimal equivalent, which in turns represents one specific character. True binary cannot be used by a computer because in addition to representing numbers, a computer must also be able to represent alphabetic characters and many other symbols used in natural language, such as %, (, +, @, &, and # (Laudon & Laudon, 2000). This requirement led manufacturers of computer hardware and input/output devices to develop standard binary codes. One standard for these codes, or representations, is known as the American Standard Code for Information Interchange (ASCII), and is represented on what is referred to as the *ASCII table* or *ASCII character set*. ASCII was originally designed as a seven-bit code, but most computers now use eight-bit versions.

The importance of the ASCII table is that it does represent a very necessary industry standard, especially for the programming industry. Programmers worldwide know that when they type the capital letter C on the keyboard, it actually is the byte 01000011 inside the computer's memory, and has 67 as its decimal equivalent on the ASCII table. Almost all character input/output devices (e.g., keyboards, printers, monitors) process ASCII characters, and almost all text-mode telecommunications are performed in ASCII (Kohanski, 1998).

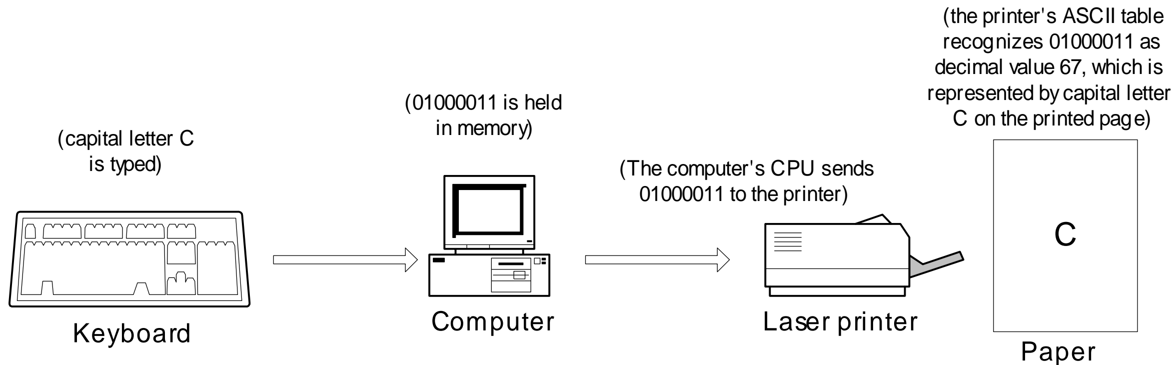
(i.e., 2^0) to the most significant exponential position of 2 (i.e., 2^7). Thus, while 01010101 is the way we read the binary representation of the number 85 on the printed page (sometimes written "←01010101" to indicate direction of flow), it is actually transmitted in reverse order as 10101010.

 © 2001

The first 32 characters (i.e., codes 00 through 31) on the ASCII table are used for printing and transmission control. Code 32 is representative of a blank space, and is obtained on the keyboard by striking the space bar once. Codes 33 through 176 are printable characters and include letters (e.g., A, a, E, f), numbers (e.g., 4, 7, 8), symbols (e.g., $, &, @), and punctuation marks (e.g., ?, !). The remaining codes through code 255 represent other functions, such as the DEL key, or other characters, often those that appear in other alphabets, such as Ñ and Ö.

One of the advantages of high-level programming languages is that they let programmers use the decimal equivalents, which are obviously easier for humans, and the programming language converts the decimal equivalent value to the eight-bit binary value used inside the computer (Perry, 1998). The general process of typing a character on the keyboard, and its translation all the way through the printing process, is depicted in Figure 2.

Figure 2
Capital letter C from keyboard to printer



Each byte is assigned an address in the memory boards. This address is a number that describes the physical location of the byte for the computer circuitry. The contents of the byte may be changed by any programming instruction, but the address of the byte cannot be changed. Within the byte itself all values are represented by electrical currents preserved by the state of a relay, vacuum tube, transistor, magnetic core, or integrated circuit, depending on the computer (Kohanski, 1998). Thus, the significance of binary notation (i.e., 1s and 0s) is that it represents the *way* that the computer actually *stores* data (Stevens, 1994).

Hexadecimal

Our decimal (i.e., $base_{ten}$) system has 10 digits, 0 through 9. Each position of a number is a power of ten. For instance, **354** can be represented as $[(\mathbf{3} * 10^2) + (\mathbf{5} * 10^1) + (\mathbf{4} * 10^0)]$, which equals 300 + 50 + 4. Since the binary numbering system

 © 2001

(i.e., $base_{two}$, or $base_2$) uses only two digits (i.e., 0 and 1), each position of a number is a power of two. Any number in the decimal system can be reduced to a binary number, and any binary number can be represented as a $base_{ten}$ value. Thus, the binary number 10111 has as its decimal (i.e., $base_{ten}$) equivalent the number 23, and is calculated as follows: $[(1 * 2^4) + (0 * 2^3) + (1 * 2^2) + (1 * 2^1) + (1 * 2^0)] = [16 + 0 + 4 + 2 + 1] = 23_{ten}$.

Viewing an eight-digit string of binary numbers is not very easy, and when larger numbers are represented, it becomes clear that writing them out in binary is cumbersome and error-prone. It is much more convenient to use a larger number system that is also an exact power of two. The first such system in common use was *octal*, or $base_{eight}$ numbering. More prevalent today, however, is the hexadecimal (hex) system, which is written in $base_{sixteen}$. In modern systems, binary numbers are almost always represented as hexadecimal digits (Kohanski, 1998). Hexadecimal (hex) is a sixteen-number set equivalent to binary numbers in which each group of four binary numbers is represented as one hex digit. The 16 digits of the hex system are the numbers 0 through 9, and the capital letters A through F. Table 2 shows the list of hex values and their equivalent values in binary and decimal.

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

To convert binary numbers to hexadecimal, consider the following example of how the binary number 01010010 becomes 52 in hexadecimal. From left to right, the first grouping of four digits is 0101, which equals 5 in $base_{two}$ (i.e., $2^2 + 2^0 = 4 + 1 = 5$). The second grouping of four digits is 0010, which equals 2 in $base_{two}$ (i.e., 2^1). Thus 01010010 becomes 52. To convert 52 to decimal, the following procedure is used: $(5 * 16^1) + (2 * 16^0) = (80 + 2) = 82$ in $base_{ten}$. On the ASCII table it is understood that $52_{sixteen} = 82_{ten}$ = R (i.e., capital letter R).

Whereas an eight-bit byte can only have 256 (i.e., 2^8) possible values, a 16-bit “word” can handle 65,536 (i.e., 2^{16}) different values. Combining four bytes into a single “longword” of 32 bits lets us work with over 4.2 billion (i.e., 2^{32}) possible values. It is easy to see the benefits of modern 32-bit processing, especially with complex applications such as video encoding (Gain, 2000). Larger still are “quadwords,” with 64 bits (i.e., 2^{64} possible values).

 © 2001

To show a comparison of how binary code, ASCII code, and hexadecimal code compare, consider the way each represents the sentence,

Robert Underwood is a real estate broker in Austin, Texas.

In binary code, it would be represented as follows:

```
01010010 01101111 01100010 01100101 01110010 01110100 00100000
01010101 01101110 01100100 01100101 01110010 01110111 01101111
01101111 01100100 00100000 01101001 01110011 00100000 01100001
00100000 01110010 01100101 01100001 01101100 00100000 01100101
01110011 01110100 01100001 01110100 01100101 00100000 01100010
01110010 01101111 01101011 01100101 01110010 00100000 01101001
01101110 00100000 01000001 01110101 01110011 01110100 01101001
01101110 00101100 00100000 01010100 01100101 01111000 01100001
01110011 00101110
```

The decimal equivalent (i.e., ASCII code) would be:

```
82 111 98 101 114 116 32 85 110 100 101 114 119 111 111 100 32
105 115 32 97 32 114 101 97 108 32 101 115 116 97 116 101 32 98
114 111 107 101 114 32 105 110 32 65 117 115 116 105 110 44 32
84 101 120 97 115 46
```

The hexadecimal representation would be:

```
52 6F 62 65 72 74 20 55 6E 64 65 72 77 6F 6F 64 20 69 73 20 61
20 72 65 61 6C 20 65 73 74 61 74 65 20 62 72 6F 6B 65 72 20 69
6E 20 41 75 73 74 69 6E 2C 20 54 65 78 61 73 2E
```

In each of the three representations above, there are 58 sets of codes. Each of the 58 binary sets, or bytes, has eight digits. Each of the 58 ASCII code sets has two or three digits. Each of the 58 hexadecimal sets has two digits. Notice that even the space between the words Robert and Underwood, obtained by striking the space bar once on the keyboard, is represented by

a value: the byte representation is 00100000, the ASCII code is 32, and the hexadecimal code is 20.

Regarding programming, it may be difficult to understand how the computer knows the difference between the numerical value 82, which is the ASCII character table's representation for the letter R. There actually is no difference as far as the computer itself is concerned. The difference is found in the context in which the program uses the byte. The following QBASIC programming language statements declare two variables and assign the value of 82 to them.

```
DIM Alpha AS STRING * 1
Alpha = "R"

DIM Num AS INTEGER
Num = 82
```

The difference is that the QBASIC interpreter allows the programmer to use the variable named "Alpha" only in the context where it expects to see a string. Similarly QBASIC allows the programmer to use the variable named "Num" in the context of where it expects to see a numeric value.

<u>Translation Hierarchy</u>

The origins of programming languages lie in machines. The native language of a computer, its machine language, is the notation to which the computer responds directly (Sethi, 1989). According to Hennessy and Patterson (1998), the first programmers communicated to computers using only binary numbers

(i.e., 1s and 0s), but the process was so tedious that they quickly invented new notations that were easier for people to understand. At first, these notations were translated to binary by hand, but eventually programmers realized that they could use the machine to help program itself through the use of programs. The first of these programs was called an *assembler*. Assembly language is a variant of machine language in which names take the place of the actual codes for machine code for machine operations, values, and storage locations. With assembly language, for example, a programmer could write "add A,B" and the assembler would translate this notation into "1000110010100000," its binary equivalent in machine code.

Assembly language was an important improvement over machine language, but it still required the programmer to write one line for every instruction that the machine would follow, forcing the programmer to think like the machine. For example, while

```
LOAD I
ADD J
STORE K
```

may be the symbolic variant in assembly language of

```
000000101011110010 10
000000101111110010 00
000000110011101010 00
```

neither is as understandable as the way we would represent the same expression in simple English: $K = I + J$ (Sethi, 1989).

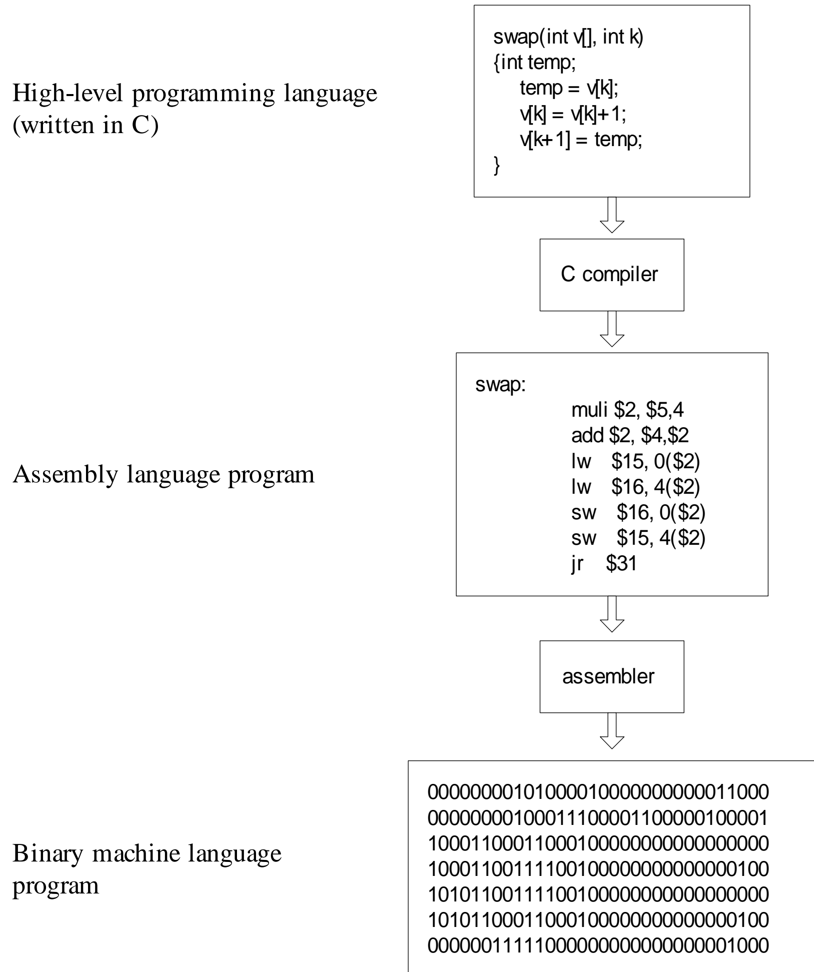
 © 2001

Developers eventually realized that if they could write a program to translate from assembly language to binary instructions, then they ought to be able to write a program that translated from some higher-level notation down to assembly language. Such was the essence of *compilers*. The languages they compile are referred to as high-level programming languages. With the use of a compiler software program, a programmer could write the high-level language expression "A + B." The compiler would then compile it into the assembly language statement "Add A,B" and the assembler would translate it into "1000110010100000" — the binary instruction that tells the computer to add the two numbers A and B. The compiler program actually examines the words or phrases of the programming instructions and converts them into assembly language, which in turn converts the modified code to numeric instructions (i.e., the machine code of 1s and 0s), which the computer circuitry can execute in the form of on/off switches. Figure 3 illustrates a more complex example showing the relationship between a high-level language, assembly language, and binary machine language[v].

[v] It is important to note that some compilers translate directly into machine language without first using assembly language.

 © 2001

Figure 3
High-level program compiled into assembly language,
and then assembled into binary machine language



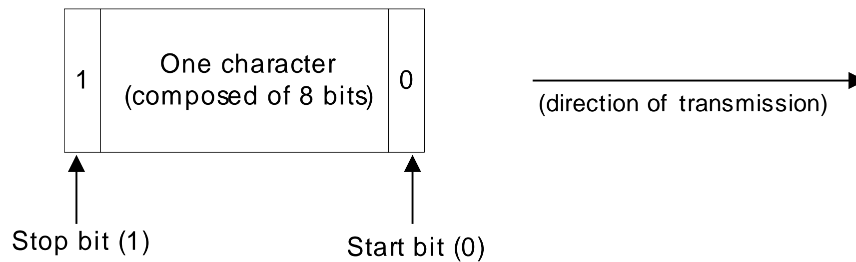
Character Transmission

In some computer terminals, characters are generated at a keyboard and transmitted singly (i.e., one character after the other). In other computer terminals, characters are run together to form a block of data. If transmitted singly, as is typical of keyboard input to a terminal, each character is

 © 2001

framed (see Figure 4) by a start bit, 0, and one or two stop bits, 1 or 11. The start and stop bits provide a positive indication of the beginning and ending of a character.

Figure 4
Framed character with overhead of
one start bit and one stop bit



Some systems provide that characters are concatenated and sent once a string or line of data is completed. That is, rather than being sent one character at a time, characters are sent as a group as each string or line of data is completed. Each character is still framed individually, but characters are sent in a group. This type of datastream is typical of a computer terminal sending keyboard input line-by-line to a communications controller (Carne, 1999).

Machine Architecture

One final aspect to consider includes a brief description of the computer’s architecture as it relates to memory. The

 © 2001

computer has 16 general-purpose registers[vi] identified by the 16 hex digits (0 through F). For identifying registers with instructions, each register is given the singular four-bit pattern that represents its unique register number (Brookshear, 2000). Thus, for example, register 0 is identified by 0000 (hexadecimal 0), register 3 is identified by 0011 (hexadecimal 3), and register F is identified by 1111 (hexadecimal F).

Since there are 256 cells in memory, each cell is given a singular address consisting of an integer in the range of 0 through 255. A memory address can therefore be represented by a pattern of hexadecimal values ranging from 00 through FF — the byte equivalents of 00000000 through 11111111, respectively.

Conclusion

There are several different ways to represent the binary nature of the computer's digital electronics: (a) on/off, (b) high voltage/low voltage, (c) 1/0, and (d) positive/negative. Each of these conventions represents essentially the same thing. Therefore, "on" ≈ "high voltage" ≈ "1" ≈ "positive." Similarly, "off" ≈ "low (or no) voltage" ≈ "0" ≈ "negative." Regardless of which convention is used to describe the computer's binary nature, it is worthwhile for programmers to understand the fundamentals of programming at the machine code

[vi] The registers are programmable, and essentially provide a high-speed cache for main memory. It is the job of the programmer to implement the register-allocation and register-replacement algorithms to determine which information to store in registers and which to store in main memory (Silberschatz & Galvin, 1999).

 © 2001

level as they design code and create programs. Creating programs that are straightforward to design, test, run, and document, are issues that programmers face regularly in both creative and professional contexts.

Programming has evolved from writing programs in the pure machine code of 1s and 0s, to the improved readability of assembly language, to the more powerful and efficient compiled programs. Nevertheless, the fundamental digital nature of the computer and binary conventions associated with it has remained essentially unchanged. The improvements that have been made have not only helped programmers create more reliable code, but have made possible the use of computers by almost anyone. The almost universal use of ASCII, hexadecimal, and 64-bit processing have all been designed to make the life of the programmer easier. As professionals, programmers should understand the basics of what happens at the machine level when they create programs. What Wilson and Clark wrote over a decade ago is still valid today, "the main purpose of a programming language is to support the construction of reliable software" (1988). The objective to be realized by programming professionals is that users are ultimately satisfied with programs that are useful. And then, like the evolution of technology and standards, so do programmers evolve in greater

 © 2001

maturity as professionals with both the art and science of programming.

 © 2001

References

Brookshear, J. (2000). Computer science: an overview (6th ed., p. 545). Reading MA: Addison Wesley Longman, Inc.

Carne, E. (1999). Telecommunications Primer: Data, Voice and Video Communications (2nd ed., pp. 84, 94, 108). Upper Saddle River, NJ: Prentice Hall.

Dilligan, R. (1998). Computing in the web age: a web-interactive introduction. (pp. 61-71). New York: Plenum Press.

Ford, N. (1990). Computer programming languages: a comparative introduction. (pp. 3-5). New York: Ellis Horwood.

Gain, B. (November 20, 2000). Vweb's chips sets sights on video encoding. Electronic Buyer's News. [on-line].

Hennessy, J. and Patterson, D. (1998). Computer organization and design: the hardware software interface. (pp. 5-7, 16, 156-163, 210, 211, 231). San Francisco: Morgan Kaufman.

Kohanski, D. (1998). The philosophical programmer. New York: St. Martin's Press.

Laudon, K. C., & Laudon, J. P. (2000). Management information systems: organization and technology in the networked enterprise (6th ed., pp. 163-164). Upper Saddle River, NJ: Prentice Hall.

Leebaert, D., (editor). (1995). The future of software. (p. 8). Cambridge: The MIT Press.

Perry. G. (1998). Sams teach yourself beginning programming in 24 hours. Indianapolis: Sams.

Silberschatz, A. & Galvin, P. (1999). Operating systems concepts (5th ed., p. 36). New York: John Wiley & Sons.

Spencer, D. (1994). Webster's new world dictionary of computer terms (5th ed.). Cleveland, OH: MacMillan General Reference.

Stevens, A. (1994). Welcome to programming: from mystery to mastery. New York: MIS:Press.

Wilson, L. and Clark, R. (1988). Comparative programming languages. Wokingham, England: Addison-Wesley.

Wang, W. (1999). Beginning programming for dummies. (pp. 36-43, 238). Foster City, CA: IDG Books Worldwide.

 © 2001