More Data Representation Chapter Four

4.1 Chapter Overview

Although the basic machine data objects (bytes, words, and double words) appear to represent nothing more than signed or unsigned numeric values, we can employ these data types to represent many other types of objects. This chapter discusses some of the other objects and their internal computer representation.

This chapter begins by discussing the floating point (real) numeric format. After integer representation, floating point representation is the second most popular numeric format in use on modern computer systems[1]. Although the floating point format is somewhat complex, the necessity to handle non-integer calculations in modern programs requires that you understand this numeric format and its limitations.

Binary Coded Decimal (BCD) is another numeric data representation that is useful in certain contexts. Although BCD is not suitable for general purpose arithmetic, it is useful in some embedded applications. The principle benefit of the BCD format is the ease with which you can convert between string and BCD format. When we look at the BCD format a little later in this chapter, you'll see why this is the case.

Computers can represent all kinds of different objects, not just numeric values. Characters are, unquestionably, one of the more popular data types a computer manipulates. In this chapter you will take a look at a couple of different ways we can represent individual characters on a computer system. This chapter discusses two of the more common character sets in use today: the ASCII character set and the Unicode character set.

This chapter concludes by discussing some common non-numeric data types like pixel colors on a video display, audio data, video data, and so on. Of course, there are lots of different representations for any kind of standard data you could envision; there is no way two chapters in a textbook can cover them all. (And that's not even considering specialized data types you could create). Nevertheless, this chapter (and the last) should give you the basic idea behind representing data on a computer system.

4.2 An Introduction to Floating Point Arithmetic

Integer arithmetic does not let you represent fractional numeric values. Therefore, modern CPUs support an approximation of *real* arithmetic: floating point arithmetic. A big problem with floating point arithmetic is that it does not follow the standard rules of algebra. Nevertheless, many programmers apply normal algebraic rules when using floating point arithmetic. This is a source of defects in many programs. One of the primary goals of this section is to describe the limitations of floating point arithmetic so you will understand how to use it properly.

Normal algebraic rules apply only to *infinite precision* arithmetic. Consider the simple statement "x:=x+1," *x* is an integer. On any modern computer this statement follows the normal rules of algebra *as long as overflow does not occur.* That is, this statement is valid only for certain values of *x* (*minint* <= *x* < *maxint*). Most programmers do not have a problem with this because they are well aware of the fact that integers in a program do not follow the standard algebraic rules (e.g., $5/2 \neq 2.5$).

Integers do not follow the standard rules of algebra because the computer represents them with a finite number of bits. You cannot represent any of the (integer) values above the maximum integer or below the minimum integer. Floating point values suffer from this same problem, only worse. After all, the integers are a subset of the real numbers. Therefore, the floating point values must represent the same infinite set of integers. However, there are an infinite number of values between any two real values, so this problem is infinitely worse. Therefore, as well as having to limit your values between a maximum and minimum range, you cannot represent all the values between those two ranges, either.

1. There are other numeric formats, such as fixed point formats and binary coded decimal format.

To represent real numbers, most floating point formats employ scientific notation and use some number of bits to represent a *mantissa* and a smaller number of bits to represent an *exponent*. The end result is that floating point numbers can only represent numbers with a specific number of *significant* digits. This has a big impact on how floating point arithmetic operates. To easily see the impact of limited precision arithmetic, we will adopt a simplified decimal floating point format for our examples. Our floating point format will provide a mantissa with three significant digits and a decimal exponent with two digits. The mantissa and exponents are both signed values as shown in Figure 4.1

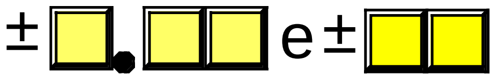

Figure 4.1 Simple Floating Point Format

When adding and subtracting two numbers in scientific notation, you must adjust the two values so that their exponents are the same. For example, when adding 1.23e1 and 4.56e0, you must adjust the values so they have the same exponent. One way to do this is to convert 4.56e0 to 0.456e1 and then add. This produces 1.686e1. Unfortunately, the result does not fit into three significant digits, so we must either *round* or *truncate* the result to three significant digits. Rounding generally produces the most accurate result, so let's round the result to obtain 1.69e1. As you can see, the lack of *precision* (the number of digits or bits we maintain in a computation) affects the accuracy (the correctness of the computation).

In the previous example, we were able to round the result because we maintained *four* significant digits *during* the calculation. If our floating point calculation is limited to three significant digits *during* computation, we would have had to truncate the last digit of the smaller number, obtaining 1.68e1 which is even less correct. To improve the accuracy of floating point calculations, it is necessary to add extra digits for use during the calculation. Extra digits available during a computation are known as *guard digits* (or *guard bits* in the case of a binary format). They greatly enhance accuracy during a long chain of computations.

The accuracy loss during a single computation usually isn't enough to worry about unless you are greatly concerned about the accuracy of your computations. However, if you compute a value which is the result of a sequence of floating point operations, the error can *accumulate* and greatly affect the computation itself. For example, suppose we were to add 1.23e3 with 1.00e0. Adjusting the numbers so their exponents are the same before the addition produces 1.23e3 + 0.001e3. The sum of these two values, even after rounding, is 1.23e3. This might seem perfectly reasonable to you; after all, we can only maintain three significant digits, adding in a small value shouldn't affect the result at all. However, suppose we were to add 1.00e0 to 1.23e3 *ten times*. The first time we add 1.00e0 to 1.23e3 we get 1.23e3. Likewise, we get this same result the second, third, fourth, ..., and tenth time we add 1.00e0 to 1.23e3. On the other hand, had we added 1.00e0 to itself ten times, then added the result (1.00e1) to 1.23e3, we would have gotten a different result, 1.24e3. This is an important thing to know about limited precision arithmetic:

❏ The order of evaluation can effect the accuracy of the result.

You will get more accurate results if the relative magnitudes (that is, the exponents) are close to one another. If you are performing a chain calculation involving addition and subtraction, you should attempt to group the values appropriately.

Another problem with addition and subtraction is that you can wind up with *false precision*. Consider the computation 1.23e0 - 1.22 e0. This produces 0.01e0. Although this is mathematically equivalent to 1.00e-2, this latter form suggests that the last two digits are exactly zero. Unfortunately, we've only got a single significant digit at this time. Indeed, some FPUs or floating point software packages might actually insert random digits (or bits) into the L.O. positions. This brings up a second important rule concerning limited precision arithmetic:

❏ Whenever subtracting two numbers with the same signs or adding two numbers with different signs, the accuracy of the result may be less than the precision available in the floating point format.

Multiplication and division do not suffer from the same problems as addition and subtraction since you do not have to adjust the exponents before the operation; all you need to do is add the exponents and multiply the mantissas (or subtract the exponents and divide the mantissas). By themselves, multiplication and division do not produce particularly poor results. However, they tend to multiply any error that already exists in a value. For example, if you multiply 1.23e0 by two, when you should be multiplying 1.24e0 by two, the result is even less accurate. This brings up a third important rule when working with limited precision arithmetic:

- ❏ When performing a chain of calculations involving addition, subtraction, multiplication, and division, try to perform the multiplication and division operations first.

Often, by applying normal algebraic transformations, you can arrange a calculation so the multiply and divide operations occur first. For example, suppose you want to compute x*(y+z). Normally you would add y and z together and multiply their sum by x. However, you will get a little more accuracy if you transform x*(y+z) to get x*y+x*z and compute the result by performing the multiplications first.

Multiplication and division are not without their own problems. When multiplying two very large or very small numbers, it is quite possible for *overflow* or *underflow* to occur. The same situation occurs when dividing a small number by a large number or dividing a large number by a small number. This brings up a fourth rule you should attempt to follow when multiplying or dividing values:

- ❏ When multiplying and dividing sets of numbers, try to arrange the multiplications so that they multiply large and small numbers together; likewise, try to divide numbers that have the same relative magnitudes.

Comparing floating point numbers is very dangerous. Given the inaccuracies present in any computation (including converting an input string to a floating point value), you should *never* compare two floating point values to see if they are equal. In a binary floating point format, different computations which produce the same (mathematical) result may differ in their least significant bits. For example, adding 1.31e0+1.69e0 should produce 3.00e0. Likewise, adding 1.50e0+1.50e0 should produce 3.00e0. However, were you to compare (1.31e0+1.69e0) against (1.50e0+1.50e0) you might find out that these sums are *not* equal to one another. The test for equality succeeds if and only if all bits (or digits) in the two operands are exactly the same. Since this is not necessarily true after two different floating point computations which should produce the same result, a straight test for equality may not work.

The standard way to test for equality between floating point numbers is to determine how much error (or tolerance) you will allow in a comparison and check to see if one value is within this error range of the other. The straight-forward way to do this is to use a test like the following:

```
if Value1 >= (Value2-error) and Value1 <= (Value2+error) then …
```

Another common way to handle this same comparison is to use a statement of the form:

```
if abs(Value1-Value2) <= error then …
```

Most texts, when discussing floating point comparisons, stop immediately after discussing the problem with floating point equality, assuming that other forms of comparison are perfectly okay with floating point numbers. This isn't true! If we are assuming that *x=y* if *x* is within *y±error*, then a simple bitwise comparison of *x* and *y* will claim that *x<y* if *y* is greater than *x* but less than *y+error*. However, in such a case *x* should really be treated as equal to *y*, not less than *y*. Therefore, we must always compare two floating point numbers using ranges, regardless of the actual comparison we want to perform. Trying to compare two floating point numbers directly can lead to an error. To compare two floating point numbers, *x* and *y*, against one another, you should use one of the following forms:

```
=   if abs(x-y) <= error then …
≠   if abs(x-y) > error then …
<   if (x-y) < -error then …
≤   if (x-y) <= error then …
>   if (x-y) > error then …
≥   if (x-y) >= -error then …
```

You must exercise care when choosing the value for *error*. This should be a value slightly greater than the largest amount of error which will creep into your computations. The exact value will depend upon the particular floating point format you use, but more on that a little later. The final rule we will state in this section is

- ❏ When comparing two floating point numbers, always compare one value to see if it is in the range given by the second value plus or minus some small error value.

There are many other little problems that can occur when using floating point values. This text can only point out some of the major problems and make you aware of the fact that you cannot treat floating point arithmetic like real arithmetic – the inaccuracies present in limited precision arithmetic can get you into trouble if you are not careful. A good text on numerical analysis or even scientific computing can help fill in the details that are beyond the scope of this text. If you are going to be working with floating point arithmetic, *in any language*, you should take the time to study the effects of limited precision arithmetic on your computations.

HLA's IF statement does not support boolean expressions involving floating point operands. Therefore, you cannot use statements like "IF(x < 3.141) THEN..." in your programs. In a later chapter that discusses floating point operations on the 80x86 you'll learn how to do floating point comparisons.

4.2.1 IEEE Floating Point Formats

When Intel planned to introduce a floating point coprocessor for their new 8086 microprocessor, they were smart enough to realize that the electrical engineers and solid-state physicists who design chips were, perhaps, not the best people to do the necessary numerical analysis to pick the best possible binary representation for a floating point format. So Intel went out and hired the best numerical analyst they could find to design a floating point format for their 8087 FPU. That person then hired two other experts in the field and the three of them (Kahn, Coonan, and Stone) designed Intel's floating point format. They did such a good job designing the KCS Floating Point Standard that the IEEE organization adopted this format for the IEEE floating point format[2].

To handle a wide range of performance and accuracy requirements, Intel actually introduced *three* floating point formats: single precision, double precision, and extended precision. The single and double precision formats corresponded to C's float and double types or FORTRAN's real and double precision types. Intel intended to use extended precision for long chains of computations. Extended precision contains 16 extra bits that the calculations could use as guard bits before rounding down to a double precision value when storing the result.

The single precision format uses a one's complement 24 bit mantissa and an eight bit excess-127 exponent. The mantissa usually represents a value between 1.0 to just under 2.0. The H.O. bit of the mantissa is always assumed to be one and represents a value just to the left of the *binary point*[3]. The remaining 23 mantissa bits appear to the right of the binary point. Therefore, the mantissa represents the value:

```
1.mmmmmmm mmmmmmmm mmmmmmmm
```

The "mmmm…" characters represent the 23 bits of the mantissa. Keep in mind that we are working with binary numbers here. Therefore, each position to the right of the binary point represents a value (zero or one) times a successive negative power of two. The implied one bit is always multiplied by 2^0, which is one. This is why the mantissa is always greater than or equal to one. Even if the other mantissa bits are all zero, the implied one bit always gives us the value one[4]. Of course, even if we had an almost infinite number of one bits after the binary point, they still would not add up to two. This is why the mantissa can represent values in the range one to just under two.

Although there are an infinite number of values between one and two, we can only represent eight million of them because we use a 23 bit mantissa (the 24th bit is always one). This is the reason for inaccuracy

2. There were some minor changes to the way certain degenerate operations were handled, but the bit representation remained essentially unchanged.
3. The binary point is the same thing as the decimal point except it appears in binary numbers rather than decimal numbers.
4. Actually, this isn't necessarily true. The IEEE floating point format supports *denormalized* values where the H.O. bit is not zero. However, we will ignore denormalized values in our discussion.

in floating point arithmetic – we are limited to 23 bits of precision in computations involving single precision floating point values.

The mantissa uses a *one's complement* format rather than two's complement. This means that the 24 bit value of the mantissa is simply an unsigned binary number and the sign bit determines whether that value is positive or negative. One's complement numbers have the unusual property that there are two representations for zero (with the sign bit set or clear). Generally, this is important only to the person designing the floating point software or hardware system. We will assume that the value zero always has the sign bit clear.

To represent values outside the range 1.0 to just under 2.0, the exponent portion of the floating point format comes into play. The floating point format raises two to the power specified by the exponent and then multiplies the mantissa by this value. The exponent is eight bits and is stored in an *excess-127* format. In excess-127 format, the exponent 2^0 is represented by the value 127 ($7f). Therefore, to convert an exponent to excess-127 format simply add 127 to the exponent value. The use of excess-127 format makes it easier to compare floating point values. The single precision floating point format takes the form shown in Figure 4.2.

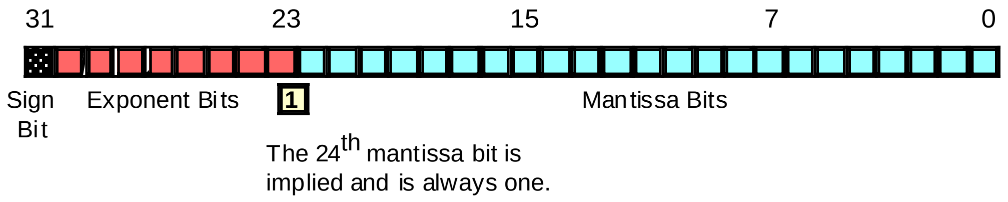

Figure 4.2 Single Precision (32-bit) Floating Point Format

With a 24 bit mantissa, you will get approximately 6-$^1/_2$ digits of precision (one half digit of precision means that the first six digits can all be in the range 0..9 but the seventh digit can only be in the range 0..x where x<9 and is generally close to five). With an eight bit excess-127 exponent, the dynamic range of single precision floating point numbers is approximately $2^{\pm 128}$ or about $10^{\pm 38}$.

Although single precision floating point numbers are perfectly suitable for many applications, the dynamic range is somewhat limited for many scientific applications and the very limited precision is unsuitable for many financial, scientific, and other applications. Furthermore, in long chains of computations, the limited precision of the single precision format may introduce serious error.

The double precision format helps overcome the problems of single precision floating point. Using twice the space, the double precision format has an 11-bit excess-1023 exponent and a 53 bit mantissa (with an implied H.O. bit of one) plus a sign bit. This provides a dynamic range of about $10^{\pm 308}$and 14-$^1/_2$ digits of precision, sufficient for most applications. Double precision floating point values take the form shown in Figure 4.3.

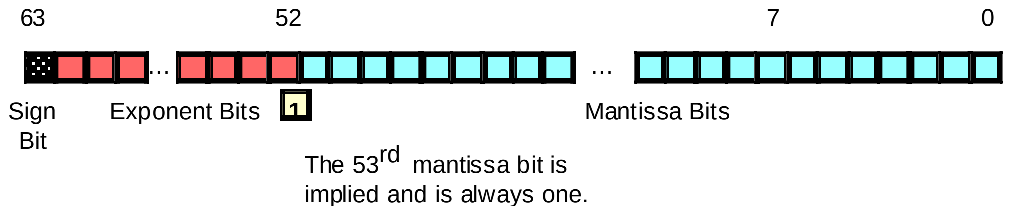

Figure 4.3 64-Bit Double Precision Floating Point Format

In order to help ensure accuracy during long chains of computations involving double precision floating point numbers, Intel designed the extended precision format. The extended precision format uses 80 bits. Twelve of the additional 16 bits are appended to the mantissa, four of the additional bits are appended to the end of the exponent. Unlike the single and double precision values, the extended precision format's mantissa does not have an implied H.O. bit which is always one. Therefore, the extended precision format provides a 64 bit mantissa, a 15 bit excess-16383 exponent, and a one bit sign. The format for the extended precision floating point value is shown in Figure 4.4:

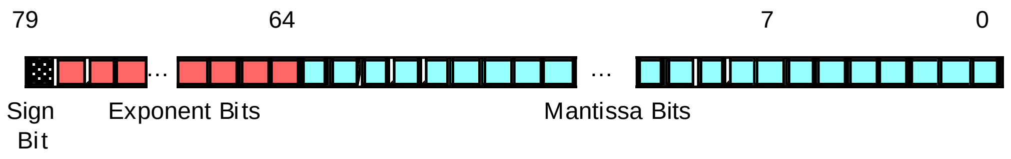

Figure 4.4 80-bit Extended Precision Floating Point Format

On the FPUs all computations are done using the extended precision form. Whenever you load a single or double precision value, the FPU automatically converts it to an extended precision value. Likewise, when you store a single or double precision value to memory, the FPU automatically rounds the value down to the appropriate size before storing it. By always working with the extended precision format, Intel guarantees a large number of guard bits are present to ensure the accuracy of your computations. Some texts erroneously claim that you should never use the extended precision format in your own programs, because Intel only guarantees accurate computations when using the single or double precision formats. This is foolish. By performing all computations using 80 bits, Intel helps ensure (but not guarantee) that you will get full 32 or 64 bit accuracy in your computations. Since the FPUs do not provide a large number of guard bits in 80 bit computations, some error will inevitably creep into the L.O. bits of an extended precision computation. However, if your computation is correct to 64 bits, the 80 bit computation will always provide *at least* 64 accurate bits. Most of the time you will get even more. While you cannot assume that you get an accurate 80 bit computation, you can usually do better than 64 when using the extended precision format.

To maintain maximum precision during computation, most computations use *normalized* values. A normalized floating point value is one whose H.O. mantissa bit contains one. Almost any non-normalized value can be normalized by shifting the mantissa bits to the left and decrementing the exponent until a one appears in the H.O. bit of the mantissa. Remember, the exponent is a binary exponent. Each time you increment the exponent, you multiply the floating point value by two. Likewise, whenever you decrement the exponent, you divide the floating point value by two. By the same token, shifting the mantissa to the left one bit position multiplies the floating point value by two; likewise, shifting the mantissa to the right divides the floating point value by two. Therefore, shifting the mantissa to the left one position *and* decrementing the exponent does not change the value of the floating point number at all.

Keeping floating point numbers normalized is beneficial because it maintains the maximum number of bits of precision for a computation. If the H.O. bits of the mantissa are all zero, the mantissa has that many fewer bits of precision available for computation. Therefore, a floating point computation will be more accurate if it involves only normalized values.

There are two important cases where a floating point number cannot be normalized. The value 0.0 is a special case. Obviously it cannot be normalized because the floating point representation for zero has no one bits in the mantissa. This, however, is not a problem since we can exactly represent the value zero with only a single bit.

The second case is when we have some H.O. bits in the mantissa which are zero but the biased exponent is also zero (and we cannot decrement it to normalize the mantissa). Rather than disallow certain small values, whose H.O. mantissa bits and biased exponent are zero (the most negative exponent possible), the IEEE standard allows special *denormalized* values to represent these smaller values[5]. Although the use of denormalized values allows IEEE floating point computations to produce better results than if underflow occurred, keep in mind that denormalized values offer less bits of precision.

5. The alternative would be to underflow the values to zero.

Since the FPU always converts single and double precision values to extended precision, extended precision arithmetic is actually *faster* than single or double precision. Therefore, the expected performance benefit of using the smaller formats is not present on these chips. However, when designing the Pentium/586 CPU, Intel redesigned the built-in floating point unit to better compete with RISC chips. Most RISC chips support a native 64 bit double precision format which is faster than Intel's extended precision format. Therefore, Intel provided native 64 bit operations on the Pentium to better compete against the RISC chips. Therefore, the double precision format is the fastest on the Pentium and later chips.

4.2.2 HLA Support for Floating Point Values

HLA provides several data types and library routines to support the use of floating point data in your assembly language programs. These include built-in types to declare floating point variables as well as routines that provide floating point input, output, and conversion.

Perhaps the best place to start when discussing HLA's floating point facilities is with a description of floating point literal constants. HLA floating point constants allow the following syntax:

- An optional "+" or "-" symbol, denoting the sign of the mantissa (if this is not present, HLA assumes that the mantissa is positive),
- Followed by one or more decimal digits,
- Optionally followed by a decimal point and one or more decimal digits,
- Optionally followed by an "e" or "E", optionally followed by a sign ("+" or "-") and one or more decimal digits.

Note: the decimal point or the "e"/"E" must be present in order to differentiate this value from an integer or unsigned literal constant. Here are some examples of legal literal floating point constants:

```
1.234     3.75e2    -1.0   1.1e-1    1e+4   0.1    -123.456e+789    +25e0
```

Notice that a floating point literal constant cannot begin with a decimal point; it must begin with a decimal digit so you must use "0.1" to represent ".1" in your programs.

HLA also allows you to place an underscore character ("_") between any two consecutive decimal digits in a floating point literal constant. You may use the underscore character in place of a comma (or other language-specific separator character) to help make your large floating point numbers easier to read. Here are some examples:

```
        1_234_837.25      1_000.00       789_934.99        9_999.99
```

To declare a floating point variable you use the *real32*, *real64*, or *real80* data types. Like their integer and unsigned brethren, the number at the end of these data type declarations specifies the number of bits used for each type's binary representation. Therefore, you use *real32* to declare single precision real values, *real64* to declare double precision floating point values, and *real80* to declare extended precision floating point values. Other than the fact that you use these types to declare floating point variables rather than integers, their use is nearly identical to that for *int8, int16, int32,* etc. The following examples demonstrate these declarations and their syntax:

```
static

        fltVar1:   real32;
        fltVar1a:  real32 := 2.7;
        pi:        real32 := 3.14159;
        DblVar:    real64;
        DblVar2:   real64 := 1.23456789e+10;
        XPVar:     real80;
        XPVar2:    real80 := -1.0e-104;
```

To output a floating point variable in ASCII form, you would use one of the *stdout.putr32, stdout.putr64*, or *stdout.putr80* routines. These procedures display a number in decimal notation, that is, a string of digits, an optional decimal point and a closing string of digits. Other than their names, these three routines use exactly the same calling sequence. Here are the calls and parameters for each of these routines:

```
stdout.putr80( r:real80; width:uns32; decpts:uns32 );
stdout.putr64( r:real64; width:uns32; decpts:uns32 );
stdout.putr32( r:real32; width:uns32; decpts:uns32 );
```

The first parameter to these procedures is the floating point value you wish to print. The size of this parameter must match the procedure's name (e.g., the *r* parameter must be an 80-bit extended precision floating point variable when calling the *stdout.putr80* routine). The second parameter specifies the field width for the output text; this is the number of print positions the number will require when the procedure displays it. Note that this width must include print positions for the sign of the number and the decimal point. The third parameter specifies the number of print positions after the decimal point. For example,

```
stdout.putr32( pi, 10, 4 );
```

displays the value

```
_ _ _ _ 3.1416
```

(the underscores represent leading spaces in this example).

Of course, if the number is very large or very small, you will want to use scientific notation rather than decimal notation for your floating point numeric output. The HLA Standard Library *stdout.pute32, stdout.pute64*, and *stdout.pute80* routines provide this facility. These routines use the following procedure prototypes:

```
stdout.pute80( r:real80; width:uns32 );
stdout.pute64( r:real64; width:uns32 );
stdout.pute32( r:real32; width:uns32 );
```

Unlike the decimal output routines, these scientific notation output routines do not require a third parameter specifying the number of digits after the decimal point to display. The width parameter, indirectly, specifies this value since all but one of the mantissa digits always appears to the right of the decimal point. These routines output their values in decimal notation, similar to the following:

```
1.23456789e+10    -1.0e-104     1e+2
```

You can also output floating point values using the HLA Standard Library *stdout.put* routine. If you specify the name of a floating point variable in the *stdout.put* parameter list, the *stdout.put* code will output the value using scientific notation. The actual field width varies depending on the size of the floating point variable (the *stdout.put* routine attempts to output as many significant digits as possible, in this case). Example:

```
stdout.put( "XPVar2 = ", XPVar2 );
```

If you specify a field width specification, by using a colon followed by a signed integer value, then the *stdout.put* routine will use the appropriate *stdout.puteXX* routine to display the value. That is, the number will still appear in scientific notation, but you get to control the field width of the output value. Like the field width for integer and unsigned values, a positive field width right justifies the number in the specified field, a negative number left justifies the value. Here is an example that prints the *XPVar2* variable using ten print positions:

```
stdout.put( "XPVar2 = ", XPVar2:10 );
```

If you wish to use *stdout.put* to print a floating point value in decimal notation, you need to use the following syntax:

Variable_Name : *Width* : *DecPts*

Note that the *DecPts* field must be a non-negative integer value.

When *stdout.put* contains a parameter of this form, it calls the corresponding *stdout.putrXX* routine to display the specified floating point value. As an example, consider the following call:

```
stdout.put( “Pi = “, pi:5:3 );
```

The corresponding output is

```
3.142
```

The HLA Standard Library provides several other useful routines you can use when outputting floating point values. Consult the HLA Standard Library reference manual for more information on these routines.

The HLA Standard Library provides several routines to let you display floating point values in a wide variety of formats. In contrast, the HLA Standard Library only provides two routines to support floating point input: *stdin.getf()* and *stdin.get()*. The *stdin.getf()* routine requires the use of the 80x86 FPU stack, a hardware component that this chapter is not going to cover. Therefore, this chapter will defer the discussion of the *stdin.getf()* routine until the chapter on arithmetic, later in this text. Since the *stdin.get()* routine provides all the capabilities of the *stdin.getf()* routine, this deference will not prove to be a problem.

You’ve already seen the syntax for the *stdin.get()* routine; its parameter list simply contains a list of variable names. *Stdin.get()* reads appropriate values for the user for each of the variables appearing in the parameter list. If you specify the name of a floating point variable, the *stdin.get()* routine automatically reads a floating point value from the user and stores the result into the specified variable. The following example demonstrates the use of this routine:

```
stdout.put( “Input a double precision floating point value: “ );
stdin.get( DblVar );
```

Warning: This section has discussed how you would declare floating point variables and how you would input and output them. It did not discuss arithmetic. Floating point arithmetic is different than integer arithmetic; you cannot use the 80x86 ADD and SUB instructions to operate on floating point values. Floating point arithmetic will be the subject of a later chapter in this text.

4.3 Binary Coded Decimal (BCD) Representation

Although the integer and floating point formats cover most of the numeric needs of an average program, there are some special cases where other numeric representations are convenient. In this section we’ll discuss the Binary Coded Decimal (BCD) format since the 80x86 CPU provides a small amount of hardware support for this data representation.

BCD values are a sequence of nibbles with each nibble representing a value in the range zero through nine. Of course you can represent values in the range 0..15 using a nibble; the BCD format, however, uses only 10 of the possible 16 different values for each nibble.

Each nibble in a BCD value represents a single decimal digit. Therefore, with a single byte (i.e., two digits) we can represent values containing two decimal digits, or values in the range 0..99. With a word, we can represent values having four decimal digits, or values in the range 0..9999. Likewise, with a double word we can represent values with up to eight decimal digits (since there are eight nibbles in a double word value).

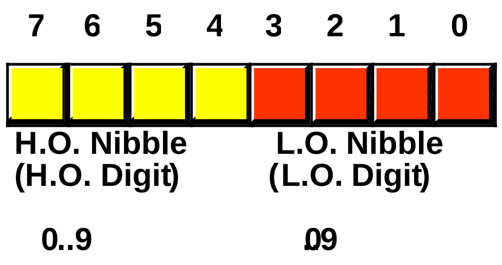

Figure 4.5 BCD Data Representation in Memory

As you can see, BCD storage isn't particularly memory efficient. For example, an eight-bit BCD variable can represent values in the range 0..99 while that same eight bits, when holding a binary value, can represent values in the range 0..255. Likewise, a 16-bit binary value can represent values in the range 0..65535 while a 16-bit BCD value can only represent about $^{1}/_{6}$ of those values (0..9999). Inefficient storage isn't the only problem. BCD calculations tend to be slower than binary calculations.

At this point, you're probably wondering why anyone would ever use the BCD format. The BCD format does have two saving graces: it's very easy to convert BCD values between the internal numeric representation and their string representation; also, its very easy to encode multi-digit decimal values in hardware (e.g., using a "thumb wheel" or dial) using BCD than it is using binary. For these two reasons, you're likely to see people using BCD in embedded systems (e.g., toaster ovens and alarm clocks) but rarely in general purpose computer software.

A few decades ago people mistakenly thought that calculations involving BCD (or just 'decimal') arithmetic was more accurate than binary calculations. Therefore, they would often perform 'important' calculations, like those involving dollars and cents (or other monetary units) using decimal-based arithmetic. While it is true that certain calculations can produce more accurate results in BCD, this statement is not true in general. Indeed, for most calculations (even those involving fixed point decimal arithmetic), the binary representation is more accurate. For this reason, most modern computer programs represent all values in a binary form. For example, the Intel x86 floating point unit (FPU) supports a pair of instructions for loading and storing BCD values. Internally, however, the FPU converts these BCD values to binary and performs all calculations in binary. It only uses BCD as an external data format (external to the FPU, that is). This generally produces more accurate results and requires far less silicon than having a separate coprocessor that supports decimal arithmetic.

This text will take up the subject of BCD arithmetic in a later chapter. Until then, you can safely ignore BCD unless you find yourself converting a COBOL program to assembly language (which is quite unlikely).

4.4 Characters

Perhaps the most important data type on a personal computer is the character data type. The term "character" refers to a human or machine readable symbol that is typically a non-numeric entity. In general, the term "character" refers to any symbol that you can normally type on a keyboard (including some symbols that may require multiple key presses to produce) or display on a video display. Many beginners often confuse the terms "character" and "alphabetic character." These terms are not the same. Punctuation symbols, numeric digits, spaces, tabs, carriage returns (enter), other control characters, and other special symbols are also characters. When this text uses the term "character" it refers to any of these characters, not just the alphabetic characters. When this text refers to alphabetic characters, it will use phrases like "alphabetic characters," "upper case characters," or "lower case characters."[6].

Another common problem beginners have when they first encounter the character data type is differentiating between numeric characters and numbers. The character ‘1’ is distinct and different from the value one. The computer (generally) uses two different internal, binary, representations for numeric characters (‘0’, ‘1’, ..., ‘9’) versus the numeric values zero through nine. You must take care not to confuse the two.

Most computer systems use a one or two byte sequence to encode the various characters in binary form. Windows and Linux certainly fall into this category, using either the ASCII or Unicode encodings for characters. This section will discuss the ASCII character set and the character declaration facilities that HLA provides.

4.4.1 The ASCII Character Encoding

The ASCII (American Standard Code for Information Interchange) Character set maps 128 textual characters to the unsigned integer values 0..127 ($0..$7F). Internally, of course, the computer represents everything using binary numbers; so it should come as no surprise that the computer also uses binary values to represent non-numeric entities such as characters. Although the exact mapping of characters to numeric values is arbitrary and unimportant, it is important to use a standardized code for this mapping since you will need to communicate with other programs and peripheral devices and you need to talk the same “language” as these other programs and devices. This is where the ASCII code comes into play; it is a standardized code that nearly everyone has agreed upon. Therefore, if you use the ASCII code 65 to represent the character “A” then you know that some peripheral device (such as a printer) will correctly interpret this value as the character “A” whenever you transmit data to that device.

You should not get the impression that ASCII is the only character set in use on computer systems. IBM uses the EBCDIC character set family on many of its mainframe computer systems. Another common character set in use is the Unicode character set. Unicode is an extension to the ASCII character set that uses 16 bits rather than seven to represent characters. This allows the use of 65,536 different characters in the character set, allowing the inclusion of most symbols in the world’s different languages into a single unified character set.

Since the ASCII character set provides only 128 different characters and a byte can represent 256 different values, an interesting question arises: “what do we do with the values 128..255 that one could store into a byte value when working with character data?” One answer is to ignore those extra values. That will be the primary approach of this text. Another possibility is to extend the ASCII character set and add an additional 128 characters to the character set. Of course, this would tend to defeat the whole purpose of having a standardized character set unless you could get everyone to agree upon the extensions. That is a difficult task.

When IBM first created their IBM-PC, they defined these extra 128 character codes to contain various non-English alphabetic characters, some line drawing graphics characters, some mathematical symbols, and several other special characters. Since IBM’s PC was the foundation for what we typically call a PC today, that character set has become a pseudo-standard on all IBM-PC compatible machines. Even on modern machines, which are not IBM-PC compatible and cannot run early PC software, the IBM extended character set still survives. Note, however, that this PC character set (an extension of the ASCII character set) is not universal. Most printers will not print the extended characters when using native fonts and many programs (particularly in non-English countries) do not use those characters for the upper 128 codes in an eight-bit value. For these reasons, this text will generally stick to the standard 128 character ASCII character set. However, a few examples and programs in this text will use the IBM PC extended character set, particularly the line drawing graphic characters (see Appendix B).

Should you need to exchange data with other machines which are not PC-compatible, you have only two alternatives: stick to standard ASCII or ensure that the target machine supports the extended IBM-PC character set. Some machines, like the Apple Macintosh, do not provide native support for the extended IBM-PC character set; however you may obtain a PC font which lets you display the extended character set.

6. Upper and lower case characters are always alphabetic characters within this text.

Other machines have similar capabilities. However, the 128 characters in the standard ASCII character set are the only ones you should count on transferring from system to system.

Despite the fact that it is a “standard”, simply encoding your data using standard ASCII characters does not guarantee compatibility across systems. While it’s true that an “A” on one machine is most likely an “A” on another machine, there is very little standardization across machines with respect to the use of the control characters. Indeed, of the 32 control codes plus delete, there are only four control codes commonly supported – backspace (BS), tab, carriage return (CR), and line feed (LF). Worse still, different machines often use these control codes in different ways. End of line is a particularly troublesome example. Windows, MS-DOS, CP/M, and other systems mark end of line by the two-character sequence CR/LF. Apple Macintosh, and many other systems mark the end of line by a single CR character. Linux, BeOS, and other UNIX systems mark the end of a line with a single LF character. Needless to say, attempting to exchange simple text files between such systems can be an experience in frustration. Even if you use standard ASCII characters in all your files on these systems, you will still need to convert the data when exchanging files between them. Fortunately, such conversions are rather simple.

Despite some major shortcomings, ASCII data is *the* standard for data interchange across computer systems and programs. Most programs can accept ASCII data; likewise most programs can produce ASCII data. Since you will be dealing with ASCII characters in assembly language, it would be wise to study the layout of the character set and memorize a few key ASCII codes (e.g., “0”, “A”, “a”, etc.).

The ASCII character set (excluding the extended characters defined by IBM) is divided into four groups of 32 characters. The first 32 characters, ASCII codes 0 through $1F (31), form a special set of non-printing characters called the control characters. We call them control characters because they perform various printer/display control operations rather than displaying symbols. Examples include *carriage return*, which positions the cursor to the left side of the current line of characters[7], line feed (which moves the cursor down one line on the output device), and back space (which moves the cursor back one position to the left). Unfortunately, different control characters perform different operations on different output devices. There is very little standardization among output devices. To find out exactly how a control character affects a particular device, you will need to consult its manual.

The second group of 32 ASCII character codes comprise various punctuation symbols, special characters, and the numeric digits. The most notable characters in this group include the space character (ASCII code $20) and the numeric digits (ASCII codes $30..$39). Note that the numeric digits differ from their numeric values only in the H.O. nibble. By subtracting $30 from the ASCII code for any particular digit you can obtain the numeric equivalent of that digit.

The third group of 32 ASCII characters contains the upper case alphabetic characters. The ASCII codes for the characters “A”..”Z” lie in the range $41..$5A (65..90). Since there are only 26 different alphabetic characters, the remaining six codes hold various special symbols.

The fourth, and final, group of 32 ASCII character codes represent the lower case alphabetic symbols, five additional special symbols, and another control character (delete). Note that the lower case character symbols use the ASCII codes $61..$7A. If you convert the codes for the upper and lower case characters to binary, you will notice that the upper case symbols differ from their lower case equivalents in exactly one bit position. For example, consider the character code for “E” and “e” in the following figure:

7. Historically, carriage return refers to the *paper carriage* used on typewriters. A carriage return consisted of physically moving the carriage all the way to the right so that the next character typed would appear at the left hand side of the paper.

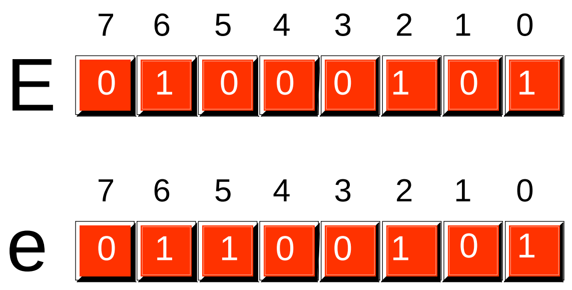

Figure 4.6 ASCII Codes for “E” and “e”

The only place these two codes differ is in bit five. Upper case characters always contain a zero in bit five; lower case alphabetic characters always contain a one in bit five. You can use this fact to quickly convert between upper and lower case. If you have an upper case character you can force it to lower case by setting bit five to one. If you have a lower case character and you wish to force it to upper case, you can do so by setting bit five to zero. You can toggle an alphabetic character between upper and lower case by simply inverting bit five.

Indeed, bits five and six determine which of the four groups in the ASCII character set you’re in:

Table 9: ASCII Groups

Bit 6	Bit 5	Group
0	0	Control Characters
0	1	Digits & Punctuation
1	0	Upper Case & Special
1	1	Lower Case & Special

So you could, for instance, convert any upper or lower case (or corresponding special) character to its equivalent control character by setting bits five and six to zero.

Consider, for a moment, the ASCII codes of the numeric digit characters:

Table 10: ASCII Codes for Numeric Digits

Character	Decimal	Hexadecimal
“0”	48	$30
“1”	49	$31
“2”	50	$32
“3”	51	$33

Table 10: ASCII Codes for Numeric Digits

Character	Decimal	Hexadecimal
“4”	52	$34
“5”	53	$35
“6”	54	$36
“7”	55	$37
“8”	56	$38
“9”	57	$39

The decimal representations of these ASCII codes are not very enlightening. However, the hexadecimal representation of these ASCII codes reveals something very important – the L.O. nibble of the ASCII code is the binary equivalent of the represented number. By stripping away (i.e., setting to zero) the H.O. nibble of a numeric character, you can convert that character code to the corresponding binary representation. Conversely, you can convert a binary value in the range 0..9 to its ASCII character representation by simply setting the H.O. nibble to three. Note that you can use the logical-AND operation to force the H.O. bits to zero; likewise, you can use the logical-OR operation to force the H.O. bits to %0011 (three).

Note that you *cannot* convert a string of numeric characters to their equivalent binary representation by simply stripping the H.O. nibble from each digit in the string. Converting 123 ($31 $32 $33) in this fashion yields three bytes: $010203, not the correct value which is $7B. Converting a string of digits to an integer requires more sophistication than this; the conversion above works only for single digits.

4.4.2 HLA Support for ASCII Characters

Although you could easily store character values in *byte* variables and use the corresponding numeric equivalent ASCII code when using a character literal in your program, such agony is unnecessary - HLA provides good support for character variables and literals in your assembly language programs.

Character literal constants in HLA take one of two forms: a single character surrounded by apostrophes or a pound symbol (“#”) followed by a numeric constant in the range 0..127 specifying the ASCII code of the character. Here are some examples:

```
‘A’        #65        #$41       #%0100_0001
```

Note that these examples all represent the same character (‘A’) since the ASCII code of ‘A’ is 65.

With a single exception, only a single character may appear between the apostrophes in a literal character constant. That single exception is the apostrophe character itself. If you wish to create an apostrophe literal constant, place four apostrophes in a row (i.e., double up the apostrophe inside the surrounding apostrophes), i.e.,

```
''''
```

The pound sign operator (“#”) must precede a legal HLA numeric constant (either decimal, hexadecimal or binary as the examples above indicate). In particular, the pound sign is not a generic character conversion function; it cannot precede registers or variable names, only constants. As a general rule, you should always use the apostrophe form of the character literal constant for graphic characters (that is, those that are printable or displayable). Use the pound sign form for control characters (that are invisible, or do funny things when you print them) or for extended ASCII characters that may not display or print properly within your source code.

Notice the difference between a character literal constant and a string literal constant in your programs. Strings are sequences of zero or more characters surrounded by quotation marks, characters are surrounded by apostrophes. It is especially important to realize that

```
‘A’ ≠ “A”
```

The character constant ‘A’ and the string containing the single character “A” have two completely different internal representations. If you attempt to use a string containing a single character where HLA expects a character constant, HLA will report an error. Strings and string constants will be the subject of a later chapter.

To declare a character variable in an HLA program, you use the *char* data type. The following declaration, for example, demonstrates how to declare a variable named *UserInput*:

```
static
    UserInput:    char;
```

This declaration reserves one byte of storage that you could use to store any character value (including eight-bit extended ASCII characters). You can also initialize character variables as the following example demonstrates:

```
static

    TheCharA:     char := ‘A’;
    ExtendedChar  char := #128;
```

Since character variables are eight-bit objects, you can manipulate them using eight-bit registers. You can move character variables into eight-bit registers and you can store the value of an eight-bit register into a character variable.

The HLA Standard Library provides a handful of routines that you can use for character I/O and manipulation; these include *stdout.putc*, *stdout.putcSize*, *stdout.put*, *stdin.getc*, and *stdin.get*.

The *stdout.putc* routine uses the following calling sequence:

```
                stdout.putc( chvar );
```

This procedure outputs the single character parameter passed to it as a character to the standard output device. The parameter may be any *char* constant or variable, or a *byte* variable or register[8].

The *stdout.putcSize* routine provides output width control when displaying character variables. The calling sequence for this procedure is

```
        stdout.putcSize( charvar, widthInt32, fillchar );
```

This routine prints the specified character (parameter *c*) using at least *width* print positions[9]. If the absolute value of *width* is greater than one, then *stdout.putcSize* prints the *fill* character as padding. If the value of *width* is positive, then *stdout.putcSize* prints the character right justified in the print field; if *width* is negative, then *stdout.putcSize* prints the character left justified in the print field. Since character output is usually left justified in a field, the *width* value will normally be negative for this call. The space character is the most common *fill* value.

You can also print character values using the generic *stdout.put* routine. If a character variable appears in the *stdout.put* parameter list, then *stdout.put* will automatically print it as a character value, e.g.,

```
    stdout.put( “Character c = ‘”, c, “‘”, nl );
```

You can read characters from the standard input using the *stdin.getc* and *stdin.get* routines. The *stdin.getc* routine does not have any parameters. It reads a single character from the standard input buffer and returns this character in the AL register. You may then store the character value away or otherwise

8. If you specify a byte variable or a byte-sized register as the parameter, the *stdout.putc* routine will output the character whose ASCII code appears in the variable or register.
9. The only time *stdout.putcSize* uses more print positions than you specify is when you specify zero as the width; then this routine uses exactly one print position.

manipulate the character in the AL register. The following program reads a single character from the user, converts it to upper case if it is a lower case character, and then displays the character:

```
program charInputDemo;
#include( “stdlib.hhf” );
static
    c:char;

begin charInputDemo;

    stdout.put( “Enter a character: “ );
    stdin.getc();
    if( al >= ‘a’ ) then

        if( al <= ‘z’ ) then

            and( $5f, al );

        endif;

    endif;
    stdout.put
    (
        “The character you entered, possibly “, nl,
        “converted to upper case, was ‘”
    );
    stdout.putc( al );
    stdout.put( “‘”, nl );

end charInputDemo;
```

Program 4.1 Character Input Sample

You can also use the generic *stdin.get* routine to read character variables from the user. If a *stdin.get* parameter is a character variable, then the *stdin.get* routine will read a character from the user and store the character value into the specified variable. Here is the program above rewritten to use the *stdin.get* routine:

```
program charInputDemo2;
#include( “stdlib.hhf” );
static
    c:char;

begin charInputDemo2;

    stdout.put( “Enter a character: “ );
    stdin.get(c);
    if( c >= ‘a’ ) then

        if( c <= ‘z’ ) then

            and( $5f, c );

        endif;
    endif;
    stdout.put
    (
        “The character you entered, possibly “, nl,
        “converted to upper case, was ‘”,
        c,
        “’”, nl
    );

end charInputDemo2;
```

Program 4.2 Stdin.get Character Input Sample

As you may recall from the last chapter, the HLA Standard Library buffers its input. Whenever you read a character from the standard input using *stdin.getc* or *stdin.get*, the library routines read the next available character from the buffer; if the buffer is empty, then the program reads a new line of text from the user and returns the first character from that line. If you want to guarantee that the program reads a new line of text from the user when you read a character variable, you should call the *stdin.flushInput* routine before attempting to read the character. This will flush the current input buffer and force the input of a new line of text on the next input (which should be your *stdin.getc* or *stdin.get* call).

The end of line is problematic. Different operating systems handle the end of line differently on output versus input. From the console device, pressing the ENTER key signals the end of a line; however, when reading data from a file you get an end of line sequence which is typically a line feed or a carriage return/line feed pair. To help solve this problem, HLA’s Standard Library provides an “end of line” function. This procedure returns true (one) in the AL register if all the current input characters have been exhausted, it returns false (zero) otherwise. The following sample program demonstrates the use of the *stdin.eoln* function.

```
program eolnDemo2;
#include( “stdlib.hhf” );
begin eolnDemo2;

    stdout.put( “Enter a short line of text: “ );
    stdin.flushInput();
    repeat

        stdin.getc();
        stdout.putc( al );
        stdout.put( “=$”, al, nl );

    until( stdin.eoln() );

end eolnDemo2;
```

Program 4.3 Testing for End of Line Using Stdin.eoln

The HLA language and the HLA Standard Library provide many other procedures and additional support for character objects. Later chapters in this textbook, as well as the HLA reference documentation, describe how to use these features.

4.4.3 The ASCII Character Set

The following table lists the binary, hexadecimal, and decimal representations for each of the 128 ASCII character codes.

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0000_0000	00	0	NULL
0000_0001	01	1	ctrl A
0000_0010	02	2	ctrl B
0000_0011	03	3	ctrl C
0000_0100	04	4	ctrl D
0000_0101	05	5	ctrl E
0000_0110	06	6	ctrl F
0000_0111	07	7	bell
0000_1000	08	8	backspace
0000_1001	09	9	tab
0000_1010	0A	10	line feed
0000_1011	0B	11	ctrl K
0000_1100	0C	12	form feed
0000_1101	0D	13	return
0000_1110	0E	14	ctrl N
0000_1111	0F	15	ctrl O
0001_0000	10	16	ctrl P
0001_0001	11	17	ctrl Q
0001_0010	12	18	ctrl R
0001_0011	13	19	ctrl S
0001_0100	14	20	ctrl T
0001_0101	15	21	ctrl U
0001_0110	16	22	ctrl V
0001_0111	17	23	ctrl W

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0001_1000	18	24	ctrl X
0001_1001	19	25	ctrl Y
0001_1010	1A	26	ctrl Z
0001_1011	1B	27	ctrl [
0001_1100	1C	28	ctrl \
0001_1101	1D	29	Esc
0001_1110	1E	30	ctrl ^
0001_1111	1F	31	ctrl _
0010_0000	20	32	space
0010_0001	21	33	!
0010_0010	22	34	"
0010_0011	23	35	#
0010_0100	24	36	$
0010_0101	25	37	%
0010_0110	26	38	&
0010_0111	27	39	'
0010_1000	28	40	(
0010_1001	29	41	)
0010_1010	2A	42	*
0010_1011	2B	43	+
0010_1100	2C	44	,
0010_1101	2D	45	-
0010_1110	2E	46	.
0010_1111	2F	47	/
0011_0000	30	48	0
0011_0001	31	49	1
0011_0010	32	50	2
0011_0011	33	51	3

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0011_0100	34	52	4
0011_0101	35	53	5
0011_0110	36	54	6
0011_0111	37	55	7
0011_1000	38	56	8
0011_1001	39	57	9
0011_1010	3A	58	:
0011_1011	3B	59	;
0011_1100	3C	60	<
0011_1101	3D	61	=
0011_1110	3E	62	>
0011_1111	3F	63	?
0100_0000	40	64	@
0100_0001	41	65	A
0100_0010	42	66	B
0100_0011	43	67	C
0100_0100	44	68	D
0100_0101	45	69	E
0100_0110	46	70	F
0100_0111	47	71	G
0100_1000	48	72	H
0100_1001	49	73	I
0100_1010	4A	74	J
0100_1011	4B	75	K
0100_1100	4C	76	L
0100_1101	4D	77	M
0100_1110	4E	78	N
0100_1111	4F	79	O

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0101_0000	50	80	P
0101_0001	51	81	Q
0101_0010	52	82	R
0101_0011	53	83	S
0101_0100	54	84	T
0101_0101	55	85	U
0101_0110	56	86	V
0101_0111	57	87	W
0101_1000	58	88	X
0101_1001	59	89	Y
0101_1010	5A	90	Z
0101_1011	5B	91	[
0101_1100	5C	92	\
0101_1101	5D	93	]
0101_1110	5E	94	^
0101_1111	5F	95	_
0110_0000	60	96	`
0110_0001	61	97	a
0110_0010	62	98	b
0110_0011	63	99	c
0110_0100	64	100	d
0110_0101	65	101	e
0110_0110	66	102	f
0110_0111	67	103	g
0110_1000	68	104	h
0110_1001	69	105	i
0110_1010	6A	106	j
0110_1011	6B	107	k

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0110_1100	6C	108	l
0110_1101	6D	109	m
0110_1110	6E	110	n
0110_1111	6F	111	o
0111_0000	70	112	p
0111_0001	71	113	q
0111_0010	72	114	r
0111_0011	73	115	s
0111_0100	74	116	t
0111_0101	75	117	u
0111_0110	76	118	v
0111_0111	77	119	w
0111_1000	78	120	x
0111_1001	79	121	y
0111_1010	7A	122	z
0111_1011	7B	123	{
0111_1100	7C	124	|
0111_1101	7D	125	}
0111_1110	7E	126	~
0111_1111	7F	127	

4.5 The UNICODE Character Set

Although the ASCII character set is, unquestionably, the most popular character representation on computers, it is certainly not the only format around. For example, IBM uses the EBCDIC code on many of its mainframe and minicomputer lines. Since EBCDIC appears mainly on IBM's big iron and you'll rarely encounter it on personal computer systems, we will not consider that character set in this text. Another character representation that is becoming popular on small computer systems (and large ones, for that matter) is the Unicode character set. Unicode overcomes two of ASCII's greatest limitations: the limited character space (i.e., a maximum of 128/256 characters in an eight-bit byte) and the lack of international (beyond the USA) characters.

Unicode uses a 16-bit word to represent a single character. Therefore, Unicode supports up to 65,536 different character codes. This is obviously a huge advance over the 256 possible codes we can represent with an eight-bit byte. Unicode is upwards compatible from ASCII. Specifically, if the H.O. 17 bits of a

Unicode character contain zero, then the L.O. seven bits represent the same character as the ASCII character with the same character code. If the H.O. 17 bits contain some non-zero value, then the character represents some other value. If you're wondering why so many different character codes are necessary, simply note that certain Asian character sets contain 4096 characters (at least, their Unicode subset).

This text will stick to the ASCII character set except for a few brief mentions of Unicode here and there. Eventually, this text may have to eliminate the discussion of ASCII in favor of Unicode since many new operating systems are using Unicode internally (and convert to ASCII as necessary). Unfortunately, many string algorithms are not as conveniently written for Unicode as for ASCII (especially character set functions) so we'll stick with ASCII in this text as long as possible.

4.6 Other Data Representations

Of course, we can represent many different objects other than numbers and characters in a computer system. The following subsections provide a brief description of the different real-world data types you might encounter.

4.6.1 Representing Colors on a Video Display

As you're probably aware, color images on a computer display are made up of a series of dots known as *pixels* (which is short for "picture elements."). Different display modes (depending on the capability of the display adapter) use different data representations for each of these pixels. The one thing in common between these data types is that they control the mixture of the three additive primary colors (red, green, and blue) to form a specific color on the display. The question, of course, is how much of each of these colors do they mix together?

Color depth is the term video card manufacturers use to describe how much red, green, and blue they mix together for each pixel. Modern video cards generally provide three color depths of eight, sixteen, or twenty-four bits, allowing 256, 65536, or over 16 million colors per pixel on the display. This produces images that are somewhat coarse and grainy (eight-bit images) to "Polaroid quality" (16-bit images), on up to "photographic quality" (24-bit images)[10].

One problem with these color depths is that two of the three formats do not contain a number of bits that is evenly divisible by three. Therefore, in each of these formats at least one of the three primary colors will have fewer bits than the others. For example, with an eight-bit color depth, two of the colors can have three bits (or eight different shades) associated with them while one of the colors must have only two bits (or four shades). Therefore, when distributing the bits there are three formats possible: 2-3-3 (two bits red, three bits green, and three bits blue), 3-2-3, or 3-3-2. Likewise, with a 16 bit color depth, two of the three colors can have five bits while the third color can have six bits. This lets us generate three different palettes using the bit values 5-5-6, 5-6-5, or 6-5-5. For 24-bit displays, each primary color can have eight bits, so there is an even distribution of the colors for each pixel.

A 24-bit display produces amazingly good results. A 16-bit display produces okay images. Eight-bit displays, to put it bluntly, produce horrible photographic images (they do produce good synthetic images like those you would manipulate with a draw program). To produce better images when using an eight-bit display, most cards provide a hardware *palette*. A palette is nothing more than an array of 24-bit values containing 256 elements[11]. The system uses the eight-bit pixel value as an index into this array of 256 values and displays the color associated with the 24-bit entry in the palette table. Although the display can still dis-

10. Some graphic artists would argue that 24 bit images are not of a sufficient quality. There are some display/printer./scanner devices capable of working with 32-bit, 36-bit, and even 48-bit images; if, of course, you're willing to pay for them.

11. Actually, the color depth of each palette entry is not necessarily fixed at 24 bits. Some display devices, for example, use 18-bit entries in their palette.

play only 256 different colors at one time, the palette mechanism lets users select exactly which colors they want to display. For example, they could display 250 shades of blue and six shades of purple if such a mixture produces a better image for them.

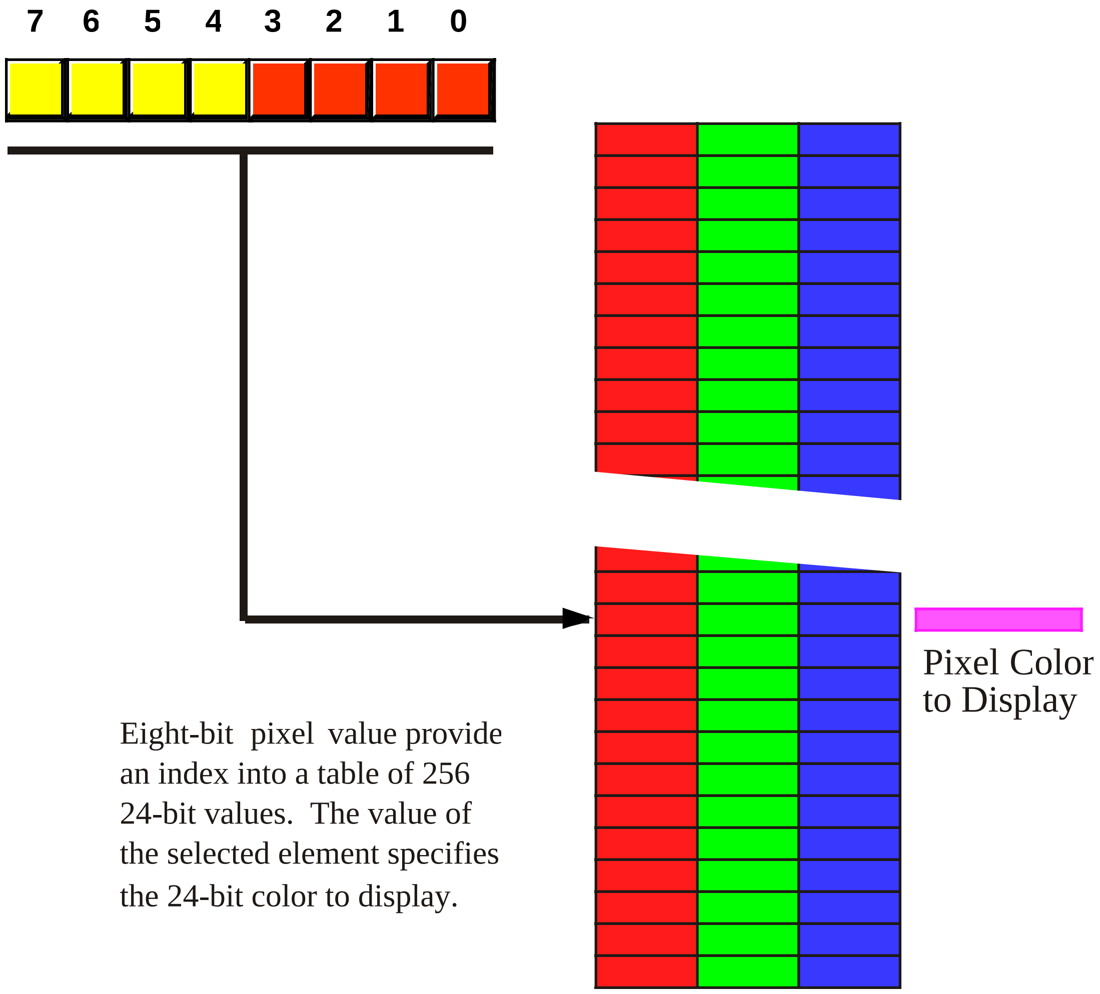
7 6 5 4 3 2 1 0
Pixel Color to Display
Eight-bit pixel value provide an index into a table of 256 24-bit values. The value of the selected element specifies the 24-bit color to display.

Figure 4.7 Extending the Number of Colors Using a Palette

Unfortunately, the palette scheme only works for displays with minimal color depths. For example, attempting to use a palette with 16-bit images would require a lookup table with 65,536 different three-byte entries – a bit much for today's operating systems (since they may have to reload the palette every time you select a window on the display). Fortunately, the higher bit depths don't require the palette concept as much as the eight-bit color depth.

Obviously, we could dream up other schemes for representing pixel color on the display. Some display systems, for example, use the subtractive primary colors (Cyan, Yellow, and Magenta, plus Black, the so-called CYMK color space). Other display system use fewer or more bits to represent the values. Some distribute the bits between various shades. Monochrome displays typically use one, four, or eight bit pixels to display various gray scales (e.g., two, sixteen, or 256 shades of gray). However, the bit organizations of this section are among the more popular in use by display adapters.

4.6.2 Representing Audio Information

Another real-world quantity you'll often find in digital form on a computer is audio information. WAV files, MP3 files, and other audio formats are quite popular on personal computers. An interesting question is "how do we represent audio information inside the computer?" While many sound formats are far too complex to discuss here (e.g., the MP3 format), it is relatively easy to represent sound using a simple sound data format (something similar to the WAV file format). In this section we'll explore a couple of possible ways to represent audio information; but before we take a look at the digital format, perhaps it's a wise idea to study the analog format first.

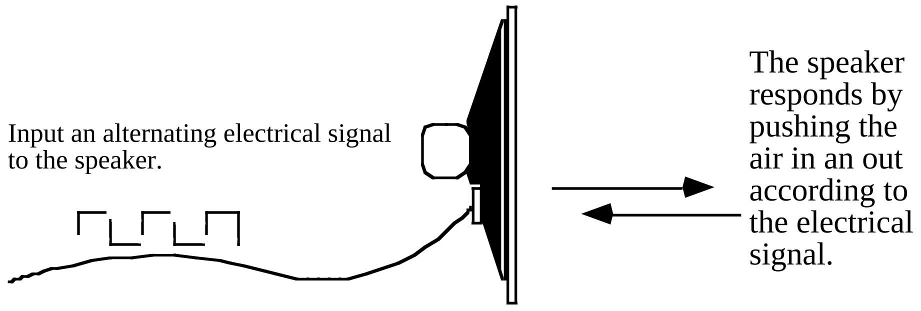
Input an alternating electrical signal to the speaker.
The speaker responds by pushing the air in an out according to the electrical signal.

Figure 4.8 Operation of a Speaker

Sounds you hear are the result of vibrating air molecules. When air molecules quickly vibrate back and forth between 20 and 20,000 times per second, we interpret this as some sort of sound. A speaker (see Figure 4.8) is a device which vibrates air in response to an electrical signal. That is, it converts an electric signal which alternates between 20 and 20,000 times per second (Hz) to an audible tone. Alternating a signal is very easy on a computer, all you have to do is apply a logic one to an output port for some period of time and then write a logic zero to the output port for a short period. Then repeat this over and over again. A plot of this activity over time appears in Figure 4.9.

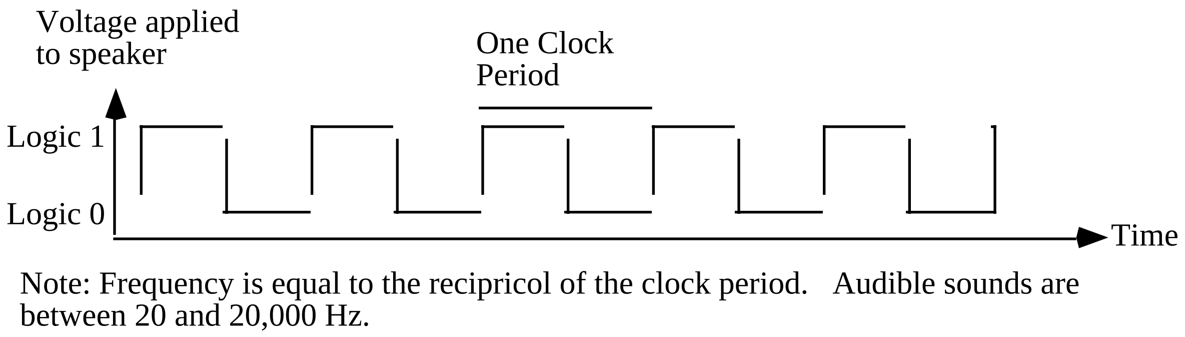
Voltage applied to speaker
One Clock Period
Logic 1
Logic 0
Time
Note: Frequency is equal to the recipricol of the clock period. Audible sounds are between 20 and 20,000 Hz.

Figure 4.9 An Audible Sound Wave

Although many humans are capable of hearing tones in the range 20-20Khz, the PC's speaker is not capable of faithfully reproducing the tones in this range. It works pretty good for sounds in the range 100-10Khz, but the volume drops off dramatically outside this range. Fortunately, most modern PCs contain a sound card that is quite capable (with appropriate external speakers) of faithfully representing "CD-Quality" sound. Of course, a good question might be "what is CD-Quality sound, anyway?" Well, to answer that question, we've got to decide how we're going to represent sound information in a binary format (see "What is "Digital Audio" Anyway?" on page 112).

Take another look at Figure 4.9. This is a graph of amplitude (volume level) over time. If logic one corresponds to a fully extended speaker cone and logic zero corresponds to a fully retracted speaker cone, then the graph in Figure 4.9 suggests that we are constantly pushing the speaker cone in an out as time progresses. This analog data, by the way, produces what is known as a "square wave" which tends to be a very bright sound at high frequencies and a very buzzy sound at low frequencies. One advantage of a square wave tone is that we only need to alternate a single bit of data over time in order to produce a tone. This is very easy to do and very inexpensive. These two reasons are why the PC's built-in speaker (not the sound card) uses exactly this technique for producing beeps and squawks.

To produce different tones with a square wave sound system is very easy. All you've got to do is write a one and a zero to some bit connected to the speaker somewhere between 20 and 20,000 times per second. You can even produce "warbling" sounds by varying the frequency at which you write those zeros and ones to the speaker.

One easy data format we can develop to represent digitized (or, should we say, "binarized") audio data is to create a stream of bits that we feed to the speaker every $^1/_{40,000}$ seconds. By alternating ones and zeros in this bit stream, we get a 20 KHz tone (remember, it takes a high and a low section to give us one clock period, hence it will take two bits to produce a single cycle on the output). To get a 20 Hz tone, you would create a bit stream that alternates between 1,000 zeros and 1,000 ones. With 1,000 zeros, the speaker will remain in the retracted position for $^1/_{40}$ seconds, following that with 1,000 ones leaves the speaker in the fully extended position for $^1/_{40}$ seconds. The end result is that the speaker moves in and out 20 times a second (giving us our 20 Hz frequency). Of course, you don't have to emit a regular pattern of zeros and ones. By varying the positions of the ones and zeros in your data stream you can dramatically affect the type of sound the system will produce.

The length of your data stream will determine how long the sound plays. With 40,000 bits, the sound will play for one second (assuming each bit's duration is $^1/_{40,000}$ seconds). As you can see, this sound format will consume 5,000 bytes per second. This may seem like a lot, but it's relatively modest by digital audio standards.

Unfortunately, square waves are very limited with respect to the sounds they can produce and are not very high fidelity (certainly not "CD-Quality"). Real analog audio signals are much more complex and you cannot represent them with two different voltage levels on a speaker. Figure 4.10 provides a typical example

What is "Digital Audio" Anyway?

"Digital Audio" or "digitized audio" is the conventional term the consumer electronics industry uses to describe audio information encoded for use on a computer. What exactly does the term "digital" mean in this case. Historically, the term "digit" refers to a finger. A digital numbering system is one based on counting one's fingers. Traditionally, then, a "digital number" was a base ten number (since the numbering system we most commonly use is based on the ten digits with which God endowed us). In the early days of computer systems the terms "digital computer" and "binary computer" were quite prevalent, with digital computers describing decimal computer systems (i.e., BCD-based systems). Binary computers, of course, were those based on the binary numbering system. Although BCD computers are mainly an artifact in the historical dust bin, the name "digital computer" lives on and is the common term to describe all computer systems, binary or otherwise. Therefore, when people talk about the logic gates computer designers use to create computer systems, they call them "digital logic." Likewise, when they refer to computerized data (like audio data), they refer to it as "digital." Technically, the term "digital" should mean base ten, not base two. Therefore, we should really refer to "digital audio" as "binary audio" to be technically correct. However, it's a little late in the game to change this term, so "digital XXXXX" lives on. Just keep in mind that the two terms "digital audio" and "binary audio" really do mean the same thing, even though they shouldn't.

of an audio waveform. Notice that the frequency and the amplitude (the height of the signal) varies considerably over time. To capture the height of the waveform at any given point in time we will need more than two values; hence, we'll need more than a single bit.

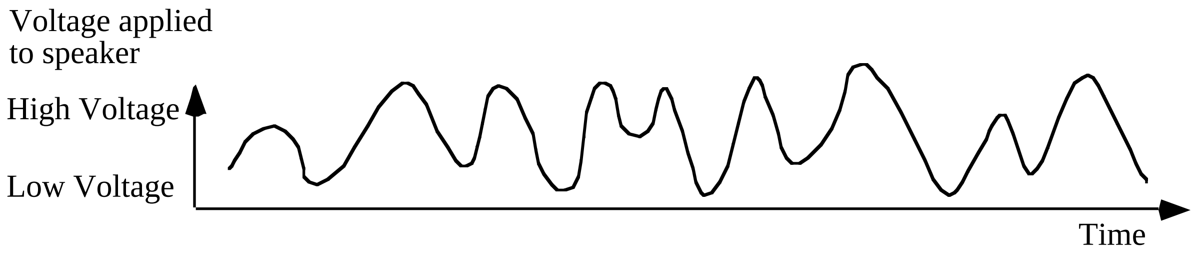

Figure 4.10 A Typical Audio Waveform

An obvious first approximation is to use a byte, rather than a single bit, to represent each point in time on our waveform. We can convert this byte data to an analog signal using a "digital to analog converter" (how obvious) or DAC. This accepts some binary number as input and produces an analog voltage on its output. This allows us to represent an impressive 256 different voltage levels in the waveform. By using eight bits, we can produce a far wider range of sounds than are possible with a single bit. Of course, our data stream now consumes 40,000 bytes per second; quite a big step up from the 5,000 bytes/second in the previous example, but still relatively modest in terms of digital audio data rates.

You might think that 256 levels would be sufficient to produce some impressive audio. Unfortunately, our hearing is logarithmic in nature and it takes an order of magnitude difference in signal for a sound to appear just a little bit louder. Therefore, our 256 different analog levels aren't as impressive to our ears. Although you can produce some decent sounds with an eight-bit data stream, it's still not high fidelity and certainly not "CD-Quality" audio.

The next obvious step up the ladder is a 16-bit value for each point of our digital audio stream. With 65,536 different analog levels we finally reach the realm of "CD-Quality" audio. Of course, we're now consuming 80,000 bytes per second to achieve this! For technical reasons, the Compact Disc format actually requires 44,100 16-bit samples per second. For a stereo (rather than monaural) data stream, you need two 16-bit values each $^{1}/_{44,100}$ seconds. This produces a whopping data rate of over 160,000 bytes per second. Now you understand the claim a littler earlier that 5,000 bytes per second is a relatively modest data rate.

Some very high quality digital audio systems use 20 or 24 bits of information and record the data at a higher frequency than 44.1 KHz (48 KHz is popular, for example). Such data formats record a better signal at the expense of a higher data rate. Some sound systems don't require anywhere near the fidelity levels of even a CD-Quality recording. Telephone conversations, for example, require only about 5,000 eight-bit samples per second (this, by the way, is why phone modems are limited to approximately 56,000 bits per second, which is about 5,000 bytes per second plus some overhead). Some common "digitizing" rates for audio include the following:

- Eight-bit samples at 11 KHz
- Eight-bit samples at 22 KHz
- Eight-bit samples at 44.1 KHz
- 16-bit samples at 32 KHz
- 16-bit samples at 44.1 KHz
- 16-bit samples at 48 KHz
- 24-bit samples at 44.1 KHz (generally in professional recording systems)
- 24-bit samples at 48 KHz (generally in professional recording systems)

The fidelity increases as you move down this list.

The exact format for various audio file formats is way beyond the scope of this text since many of the formats incorporate data compression. Some simple audio file formats like WAV and AIFF consist of little more than the digitized byte stream, but other formats are nearly indecipherable in their complexity. The exact nature of a sound data type is highly dependent upon the sound hardware in your system, so we won't delve any farther into this subject. There are several books available on computer audio and sound file formats if you're interested in pursuing this subject farther.

4.6.3 Representing Musical Information

Although it is possible to compress an audio data stream somewhat, high-quality audio will consume a large amount of data. CD-Quality audio consumes just over 160 Kilobytes per second, so a CD at 650 Megabytes holds enough data for just over an hour of audio (in stereo). Earlier, you saw that we could use a palette to allow higher quality color images on an eight-bit display. An interesting question is "can we create a sound palette to let us encode higher quality audio?" Unfortunately, the general answer is no because audio information is much less redundant than video information and you cannot produce good results with rough approximation (which using a sound palette would require). However, if you're trying to produce a specific sound, rather than trying to faithfully reproduce some recording, there are some possibilities open to you.

The advantage to the digitized audio format is that it records *everything*. In a music track, for example, the digital information records all the instruments, the vocalists, the background noise, and, well, *everything*. Sometimes you might not need to retain all this information. For example, if all you want to record is a keyboard player's synthesizer, the ability to record all the other audio information simultaneously is not necessary. In fact, with an appropriate interface to the computer, recording the audio signal from the keyboard is completely unnecessary. A far more cost-effective approach (from a memory usage point of view) is to simply record the notes the keyboardist plays (along with the duration of each note and the velocity at which the keyboardist plays the note) and then simply feed this keyboard information back to the synthesizer to play the music at a later time. Since it only takes a few bytes to record each note the keyboardist plays, and the keyboardist generally plays fewer than 100 notes per second, the amount of data needed to record a complex piece of music is tiny compared to a digitized audio recording of the same performance.

One very popular format for recording musical information in this fashion is the MIDI format (MIDI stands for Musical Instrument Digital Interface and it specifies how to connect musical instruments, computers, and other equipment together). The MIDI protocol uses multi-byte values to record information about a series of instruments (a simple MIDI file can actually control up to 16 or more instruments simultaneously).

Although the internal data format of the MIDI protocol is beyond the scope of this chapter, it is interesting to note that a MIDI command is effectively equivalent to a "palette look-up" for an audio signal. When a musical instrument receives a MIDI command telling it to play back some note, that instrument generally plays back some waveform stored in the synthesizer.

Note that you don't actually need an external keyboard/synthesizer to play back MIDI files. Most sound cards contain software that will interpret MIDI commands and play the accompany notes. These cards definitely use the MIDI command as an index into a "wave table" (short for waveform lookup table) to play the accompanying sound. Although the quality of the sound these cards reproduce is often inferior to that a professional synthesizer produces, they do let you play MIDI files without purchasing an expensive synthesizer module[12].

If you're interested in the actual data format that MIDI uses, there are dozens of texts available on the MIDI format. Any local music store should carry several of these. You should also be able to find lots of information on MIDI on the Internet (try Roland's web site as a good starting point).

12. For those who would like a better MIDI experience using a sound card, some synthesizer manufacturers produce sound cards with an integrated synthesizer on-board.

4.6.4 Representing Video Information

Recent increases in disk space, computer speed, and network access have allowed an explosion in the popularity of multimedia on personal computers. Although the term “multimedia” suggests that the data format deals with many different types of media, most people use this term to describe digital video recording and playback on a computer system. In fact, most multimedia formats support at least two mediums: video and audio. The more popular formats like Apple’s Quicktime support other concurrent media streams as well (e.g., a separate subtitle track, time codes, and device control). To simplify matters, we limit the discussion in this section to digital video streams.

Fundamentally, a video image is nothing more than a succession of still pictures that the system displays at some rate like 30 images per second. Therefore, if we want to create a digitized video image format, all we really need to do is store 30 or so pictures for each second of video we wish to view. This may not seem like a big deal, but consider that a typical “full screen” video display has 640x480 pixels or a total of 307,200 pixels. If we use a 24-bit RGB color space, then each pixel will require three bytes, raising the total to 921,600 bytes per image. Displaying 30 of these images per second means our video format will consume 27,648,000 bytes per second. Digital audio, at 160 Kilobytes per second is virtually nothing compared to the data requirements for digital video.

Although computer systems and hard disk systems have advanced tremendously over the past decade, maintaining a 30 MByte/second data rate from disk to display is a little too much to expect from all but the most expensive workstations currently available (at least, in the year 2000 as this was written). Therefore, most multimedia systems use various techniques (or combinations of these techniques) to get the data rate down to something more reasonable. In stock computer systems, a common technique is to display a 320x240 quarter screen image rather than a full-screen 640x480 image. This reduces the data rate to about seven megabytes per second.

Another technique digital video formats use is to *compress* the video data. Video data tends to contain lots of redundant information that the system can eliminate through the use of compression. The popular DV format for digital video camcorders, for example, compresses the data stream by almost 90%, requiring only a 3.3 MByte/second data rate for full-screen video. This type of compression is not without cost. There is a detectable, though slight, loss in image quality when employing DV compression on a video image. Nevertheless, this compression makes it possible to deal with digital video data streams on a contemporary computer system. Compressed data formats are a little beyond the scope of this chapter; however, by the time you finish this text you should be well-prepared to deal with compressed data formats. Programmers writing video data compression algorithms often use assembly language because compression and decompression algorithms need to be very fast to process a video stream in real time. Therefore, keep reading this text if you’re interested in working on these types of algorithms.

4.6.5 Where to Get More Information About Data Types

Since there are many ways to represent a particular real-world object inside the computer, and nearly an infinite variety of real-world objects, this text cannot even begin to cover all the possibilities. In fact, one of the most important steps in writing a piece of computer software is to carefully consider what objects the software needs to represent and then choose an appropriate internal representation for that object. For some objects or processes, an internal representation is fairly obvious; for other objects or processes, developing an appropriate data type representation is a difficult task. Although we will continue to look at different data representations throughout this text, if you’re really interested in learning more about data representation of real world objects, activities, and processes, you should consult a good “Data Structures and Algorithms” textbook. This text does not have the space to treat these subjects properly (since it still has to teach assembly language). Most texts on data structures present their material in a high level language. Adopting this material to assembly language is not difficult, especially once you’ve digested a large percentage of this text. For something a little closer to home, you might consider reading Knuth’s “The Art of Computer Programming” that describes data structures and algorithms using a synthetic assembly language called *MIX*. Although MIX isn’t the same as HLA or even x86 assembly language, you will probably find it easier to convert algorithms in this text to x86 than it would be to convert algorithms written in Pascal, Java, or C++ to assembly language.

4.7 Putting It All Together

Perhaps the most important fact this chapter and the last chapter present is that computer programs all use strings of binary bits to represent data internally. It is up to an application program to distinguish between the possible representations. For example, the bit string %0100_0001 could represent the numeric value 65, an ASCII character (‘A’), or the mantissa portion of a floating point value ($41). The CPU cannot and does not distinguish between these different representations, it simply processes this eight-bit value as a bit string and leaves the interpretation of the data to the application.

Beginning assembly language programmers often have trouble comprehending that they are responsible for interpreting the type of data found in memory; after all, one of the most important abstractions that high level languages provide is to associate a data type with a bit string in memory. This allows the compiler to do the interpretation of data representation rather than the programmer. Therefore, an important point this chapter makes is that assembly language programmers must handle this interpretation themselves. The HLA language provides built-in data types that seem to provide these abstractions, but keep in mind that once you’ve loaded a value into a register, HLA can no longer interpret that data for you, it is your responsibility to use the appropriate machine instructions that operate on the specified data.

One small amount of checking that HLA and the CPU does enforce is size checking - HLA will not allow you to mix sizes of operands within most instructions[13]. That is, you cannot specify a byte operand and a word operand in the same instruction that expects its two operands to be the same size. However, as the following program indicates, you can easily write a program that treats the same value as completely different types.

```
program dataInterpretation;
#include( “stdlib.hhf” );
static
    r:  real32 := -1.0;

begin dataInterpretation;


    stdout.put( “‘r’ interpreted as a real32 value: “, r:5:2, nl );

    stdout.put( “‘r’ interpreted as an uns32 value: “ );
    mov( r, eax );
    stdout.putu32( eax );
    stdout.newln();

    stdout.put( “‘r’ interpreted as an int32 value: “ );
    mov( r, eax );
    stdout.puti32( eax );
    stdout.newln();

    stdout.put( “‘r’ interpreted as a dword value: $” );
    mov( r, eax );
    stdout.putd( eax );
    stdout.newln();

end dataInterpretation;
```

13. The sign and zero extension instructions are an obvious exception, though HLA still checks the operand sizes to ensure they are appropriate.

Program 4.4 Interpreting a Single Value as Several Different Data Types

As this sample program demonstrates, you can get completely different results by interpreting your data differently during your program's execution. So always remember, it is your responsibility to interpret the data in your program. HLA helps a little by allowing you to declare data types that are slightly more abstract than bytes, words, or double words; HLA also provides certain support routines, like stdout.put, that will automatically interpret these abstract data types for you; however, it is generally your responsibility to use the appropriate machine instructions to consistently manipulate memory objects according to their data type.