

# 1

## HELLO, WORLD OF ASSEMBLY LANGUAGE



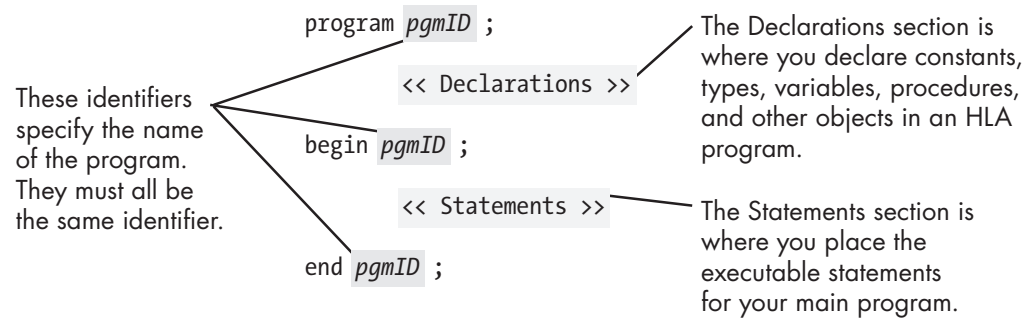
This chapter is a “quick-start” chapter that lets you start writing basic assembly language programs as rapidly as possible. This chapter does the following:

- Presents the basic syntax of an HLA (High Level Assembly) program
- Introduces you to the Intel CPU architecture
- Provides a handful of data declarations, machine instructions, and high-level control statements
- Describes some utility routines you can call in the HLA Standard Library
- Shows you how to write some simple assembly language programs

By the conclusion of this chapter, you should understand the basic syntax of an HLA program and should understand the prerequisites that are needed to start learning new assembly language features in the chapters that follow.

## 1.1 The Anatomy of an HLA Program

A typical HLA program takes the form shown in Figure 1-1.



`program`, `begin`, and `end` are HLA reserved words that delineate the program. Note the placement of the semicolons in this program.

Figure 1-1: Basic HLA program

`pgmID` in the template above is a user-defined program identifier. You must pick an appropriate descriptive name for your program. In particular, `pgmID` would be a horrible choice for any real program. If you are writing programs as part of a course assignment, your instructor will probably give you the name to use for your main program. If you are writing your own HLA program, you will have to choose an appropriate name for your project.

Identifiers in HLA are very similar to identifiers in most high-level languages. HLA identifiers may begin with an underscore or an alphabetic character and may be followed by zero or more alphanumeric or underscore characters. HLA's identifiers are *case neutral*. This means that the identifiers are case sensitive insofar as you must always spell an identifier exactly the same way in your program (even with respect to upper- and lowercase). However, unlike in case-sensitive languages such as C/C++, you may not declare two identifiers in the program whose name differs only by alphabetic case.

A traditional first program people write, popularized by Kernighan and Ritchie's *The C Programming Language*, is the "Hello, world!" program. This program makes an excellent concrete example for someone who is learning a new language. Listing 1-1 presents the HLA *helloWorld* program.

---

```
program helloWorld;
#include( "stdlib.hhf" );

begin helloWorld;

    stdout.put( "Hello, World of Assembly Language", nl );

end helloWorld;
```

---

Listing 1-1: The *helloWorld* program

The `#include` statement in this program tells the HLA compiler to include a set of declarations from the *stdlib.hhf* (standard library, HLA Header File). Among other things, this file contains the declaration of the `stdout.put` code that this program uses.

The `stdout.put` statement is the print statement for the HLA language. You use it to write data to the standard output device (generally the console). To anyone familiar with I/O statements in a high-level language, it should be obvious that this statement prints the phrase `Hello, World of Assembly Language`. The `nl` appearing at the end of this statement is a constant, also defined in *stdlib.hhf*, that corresponds to the newline sequence.

Note that semicolons follow the program, `begin`, `stdout.put`, and `end` statements. Technically speaking, a semicolon does not follow the `#include` statement. It is possible to create include files that generate an error if a semicolon follows the `#include` statement, so you may want to get in the habit of not putting a semicolon here.

The `#include` is your first introduction to HLA declarations. The `#include` itself isn't actually a declaration, but it does tell the HLA compiler to substitute the file *stdlib.hhf* in place of the `#include` directive, thus inserting several declarations at this point in your program. Most HLA programs you will write will need to include one or more of the HLA Standard Library header files (*stdlib.hhf* actually includes all the standard library definitions into your program).

Compiling this program produces a *console* application. Running this program in a command window prints the specified string, and then control returns to the command-line interpreter (or *shell* in Unix terminology).

HLA is a free-format language. Therefore, you may split statements across multiple lines if this helps to make your programs more readable. For example, you could write the `stdout.put` statement in the *helloWorld* program as follows:

---

```
stdout.put
(
    "Hello, World of Assembly Language",
    nl
);
```

---

Another construction you'll see appearing in example code throughout this text is that HLA automatically concatenates any adjacent string constants it finds in your source file. Therefore, the statement above is also equivalent to

---

```
stdout.put
(
    "Hello, "
    "World of Assembly Language",
    nl
);
```

---

Indeed, `\n` (the newline) is really nothing more than a string constant, so (technically) the comma between the `\n` and the preceding string isn't necessary. You'll often see the above written as

---

```
stdout.put( "Hello, World of Assembly Language" \n );
```

---

Notice the lack of a comma between the string constant and `\n`; this turns out to be legal in HLA, though it applies only to certain constants; you may not, in general, drop the comma. Chapter 4 explains in detail how this works. This discussion appears here because you'll probably see this "trick" employed by sample code prior to the formal explanation.

## 1.2 Running Your First HLA Program

The whole purpose of the "Hello, world!" program is to provide a simple example by which someone who is learning a new programming language can figure out how to use the tools needed to compile and run programs in that language. True, the *helloWorld* program in Section 1.1 helps demonstrate the format and syntax of a simple HLA program, but the real purpose behind a program like *helloWorld* is to learn how to create and run a program from beginning to end. Although the previous section presents the layout of an HLA program, it did not discuss how to edit, compile, and run that program. This section will briefly cover those details.

All of the software you need to compile and run HLA programs can be found at <http://www.artofasm.com/> or at <http://webster.cs.ucr.edu/>. Select **High Level Assembly** from the Quick Navigation Panel and then the Download HLA link from that page. HLA is currently available for Windows, Mac OS X, Linux, and FreeBSD. Download the appropriate version of the HLA software for your system. From the Download HLA web page, you will also be able to download all the software associated with this book. If the HLA download doesn't include them, you will probably want to download the HLA reference manual and the HLA Standard Library reference manual along with HLA and the software for this book. This text does not describe the entire HLA language, nor does it describe the entire HLA Standard Library. You'll want to have these reference manuals handy as you learn assembly language using HLA.

This section will not describe how to install and set up the HLA system because those instructions change over time. The HLA download page for each of the operating systems describes how to install and use HLA. Please consult those instructions for the exact installation procedure.

Creating, compiling, and running an HLA program is very similar to the process you'd use when creating, compiling, or running a program in any computer language. First, because HLA is not an *integrated development environment (IDE)* that allows you to edit, compile, test and debug, and run your application all from within the same program, you'll create and edit HLA programs using a text editor.<sup>1</sup>

---

<sup>1</sup> HIDE (HLA Integrated Development Environment) is an IDE available for Windows users. See the High Level Assembly web page for details on downloading HIDE.

Windows, Mac OS X, Linux, and FreeBSD offer many text editor options. You can even use the text editor provided with other IDEs to create and edit HLA programs (such as those found in Visual C++, Borland's Delphi, Apple's Xcode, and similar languages). The only restriction is that HLA expects ASCII text files, so the editor you use must be capable of manipulating and saving text files. Under Windows you can always use Notepad to create HLA programs. If you're working under Linux and FreeBSD you can use joe, vi, or emacs. Under Mac OS X you can use XCode or Text Wrangler or another editor of your preference.

The HLA compiler<sup>2</sup> is a traditional *command-line compiler*, which means that you need to run it from a Windows *command-line prompt* or a Linux/FreeBSD/Mac OS X *shell*. To do so, enter something like the following into the command-line prompt or shell window:

---

```
hla hw.hla
```

---

This command tells HLA to compile the *hw.hla* (*helloWorld*) program to an executable file. Assuming there are no errors, you can run the resulting program by typing the following command into your command prompt window (Windows):

---

```
hw
```

---

or into the shell interpreter window (Linux/FreeBSD/Mac OS X):

---

```
./hw
```

---

If you're having problems getting the program to compile and run properly, please see the HLA installation instructions on the HLA download page. These instructions describe in great detail how to install, set up, and use HLA.

## 1.3 Some Basic HLA Data Declarations

HLA provides a wide variety of constant, type, and data declaration statements. Later chapters will cover the declaration sections in more detail, but it's important to know how to declare a few simple variables in an HLA program.

HLA predefines several different signed integer types including `int8`, `int16`, and `int32`, corresponding to 8-bit (1-byte) signed integers, 16-bit (2-byte) signed integers, and 32-bit (4-byte) signed integers, respectively.<sup>3</sup> Typical variable declarations occur in the HLA *static variable section*. A typical set of variable declarations takes the form shown in Figure 1-2.

---

<sup>2</sup> Traditionally, programmers have always called translators for assembly languages *assemblers* rather than *compilers*. However, because of HLA's high-level features, it is more proper to call HLA a compiler rather than an assembler.

<sup>3</sup> A discussion of bits and bytes will appear in Chapter 2 for those who are unfamiliar with these terms.

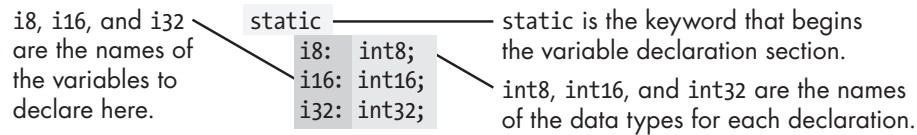


Figure 1-2: Static variable declarations

Those who are familiar with the Pascal language should be comfortable with this declaration syntax. This example demonstrates how to declare three separate integers: `i8`, `i16`, and `i32`. Of course, in a real program you should use variable names that are more descriptive. While names like `i8` and `i32` describe the type of the object, they do not describe its purpose. Variable names should describe the purpose of the object.

In the *static declaration section*, you can also give a variable an initial value that the operating system will assign to the variable when it loads the program into memory. Figure 1-3 provides the syntax for this.

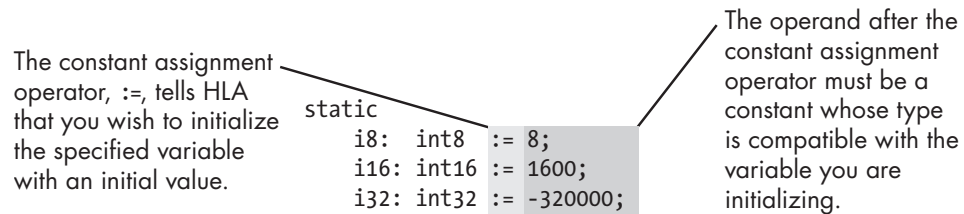


Figure 1-3: Static variable initialization

It is important to realize that the expression following the assignment operator (`:=`) must be a constant expression. You cannot assign the values of other variables within a static variable declaration.

Those familiar with other high-level languages (especially Pascal) should note that you can declare only one variable per statement. That is, HLA does not allow a comma-delimited list of variable names followed by a colon and a type identifier. Each variable declaration consists of a single identifier, a colon, a type ID, and a semicolon.

Listing 1-2 provides a simple HLA program that demonstrates the use of variables within an HLA program.

---

```
Program DemoVars;
#include( "stdlib.hhf" )

static
  InitDemo:      int32 := 5;
  NotInitialized: int32;

begin DemoVars;

  // Display the value of the pre-initialized variable:

  stdout.put( "InitDemo's value is ", InitDemo, nl );

  // Input an integer value from the user and display that value:
```

```
    stdout.put( "Enter an integer value: " );
    stdin.get( NotInitialized );
    stdout.put( "You entered: ", NotInitialized, nl );

end DemoVars;
```

---

*Listing 1-2: Variable declaration and use*

In addition to static variable declarations, this example introduces three new concepts. First, the `stdout.put` statement allows multiple parameters. If you specify an integer value, `stdout.put` will convert that value to its string representation on output.

The second new feature introduced in Listing 1-2 is the `stdin.get` statement. This statement reads a value from the standard input device (usually the keyboard), converts the value to an integer, and stores the integer value into the `NotInitialized` variable. Finally, Listing 1-2 also introduces the syntax for (one form of) HLA comments. The HLA compiler ignores all text from the `//` sequence to the end of the current line. (Those familiar with Java, C++, and Delphi should recognize these comments.)

## 1.4 Boolean Values

HLA and the HLA Standard Library provide limited support for boolean objects. You can declare boolean variables, use boolean literal constants, use boolean variables in boolean expressions, and you can print the values of boolean variables.

Boolean literal constants consist of the two predefined identifiers `true` and `false`. Internally, HLA represents the value `true` using the numeric value 1; HLA represents `false` using the value 0. Most programs treat 0 as `false` and anything else as `true`, so HLA's representations for `true` and `false` should prove sufficient.

To declare a boolean variable, you use the `boolean` data type. HLA uses a single byte (the least amount of memory it can allocate) to represent boolean values. The following example demonstrates some typical declarations:

---

```
static
    BoolVar:    boolean;
    HasClass:   boolean := false;
    IsClear:    boolean := true;
```

---

As this example demonstrates, you can initialize boolean variables if you desire.

Because boolean variables are byte objects, you can manipulate them using any instructions that operate directly on 8-bit values. Furthermore, as long as you ensure that your boolean variables only contain 0 and 1 (for `false` and `true`, respectively), you can use the 80x86 `and`, `or`, `xor`, and `not` instructions to manipulate these boolean values (these instructions are covered in Chapter 2).

You can print boolean values by making a call to the `stdout.put` routine. For example:

---

```
stdout.put( BoolVar )
```

---

This routine prints the text `true` or `false` depending upon the value of the boolean parameter (0 is false; anything else is true). Note that the HLA Standard Library does not allow you to read boolean values via `stdin.get`.

## 1.5 Character Values

HLA lets you declare 1-byte ASCII character objects using the `char` data type. You may initialize character variables with a literal character value by surrounding the character with a pair of apostrophes. The following example demonstrates how to declare and initialize character variables in HLA:

---

```
static
    c: char;
    LetterA: char := 'A';
```

---

You can print character variables use the `stdout.put` routine, and you can read character variables using the `stdin.get` procedure call.

## 1.6 An Introduction to the Intel 80x86 CPU Family

Thus far, you've seen a couple of HLA programs that will actually compile and run. However, all the statements appearing in programs to this point have been either data declarations or calls to HLA Standard Library routines. There hasn't been any *real* assembly language. Before we can progress any further and learn some real assembly language, a detour is necessary; unless you understand the basic structure of the Intel 80x86 CPU family, the machine instructions will make little sense.

The Intel CPU family is generally classified as a *Von Neumann Architecture Machine*. Von Neumann computer systems contain three main building blocks: the *central processing unit (CPU)*, *memory*, and *input/output (I/O) devices*. These three components are interconnected using the *system bus* (consisting of the address, data, and control buses). The block diagram in Figure 1-4 shows this relationship.

The CPU communicates with memory and I/O devices by placing a numeric value on the address bus to select one of the memory locations or I/O device port locations, each of which has a unique binary numeric *address*. Then the CPU, memory, and I/O devices pass data among themselves by placing the data on the data bus. The control bus contains signals that determine the direction of the data transfer (to/from memory and to/from an I/O device).



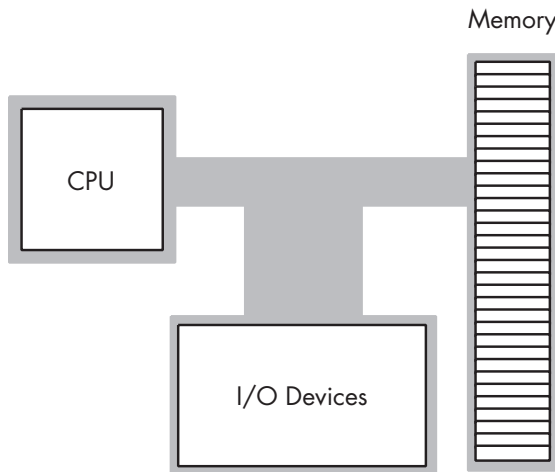


Figure 1-4: Von Neumann computer system block diagram

The 80x86 CPU registers can be broken down into four categories: general-purpose registers, special-purpose application-accessible registers, segment registers, and special-purpose kernel-mode registers. Because the segment registers aren't used much in modern 32-bit operating systems (such as Windows, Mac OS X, FreeBSD, and Linux) and because this text is geared to writing programs written for 32-bit operating systems, there is little need to discuss the segment registers. The special-purpose kernel-mode registers are intended for writing operating systems, debuggers, and other system-level tools. Such software construction is well beyond the scope of this text.

The 80x86 (Intel family) CPUs provide several general-purpose registers for application use. These include eight 32-bit registers that have the following names: EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP.

The *E* prefix on each name stands for *extended*. This prefix differentiates the 32-bit registers from the eight 16-bit registers that have the following names: AX, BX, CX, DX, SI, DI, BP, and SP.

Finally, the 80x86 CPUs provide eight 8-bit registers that have the following names: AL, AH, BL, BH, CL, CH, DL, and DH.

Unfortunately, these are not all separate registers. That is, the 80x86 does not provide 24 independent registers. Instead, the 80x86 overlays the 32-bit registers with the 16-bit registers, and it overlays the 16-bit registers with the 8-bit registers. Figure 1-5 shows this relationship.

The most important thing to note about the general-purpose registers is that they are not independent. Modifying one register may modify as many as three other registers. For example, modification of the EAX register may very well modify the AL, AH, and AX registers. This fact cannot be overemphasized here. A very common mistake in programs written by beginning assembly language programmers is register value corruption because the programmer did not completely understand the ramifications of the relationship shown in Figure 1-5.

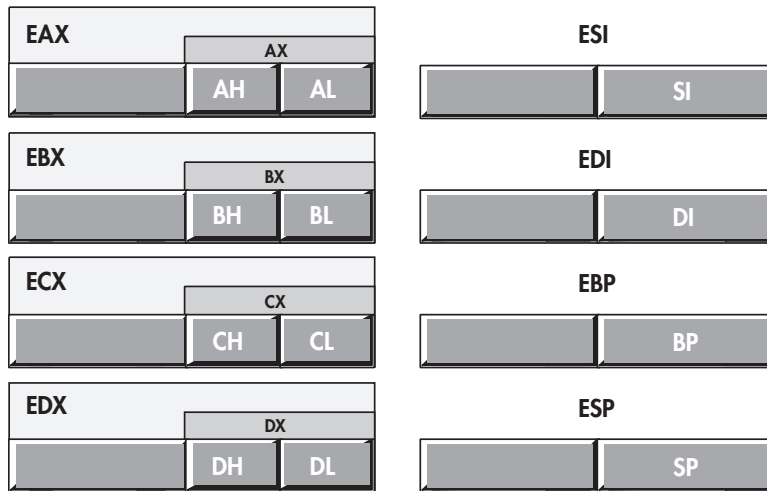


Figure 1-5: 80x86 (Intel CPU) general-purpose registers

The EFLAGS register is a 32-bit register that encapsulates several single-bit boolean (true/false) values. Most of the bits in the EFLAGS register are either reserved for kernel mode (operating system) functions or are of little interest to the application programmer. Eight of these bits (or *flags*) are of interest to application programmers writing assembly language programs. These are the overflow, direction, interrupt disable,<sup>4</sup> sign, zero, auxiliary carry, parity, and carry flags. Figure 1-6 shows the layout of the flags within the lower 16 bits of the EFLAGS register.

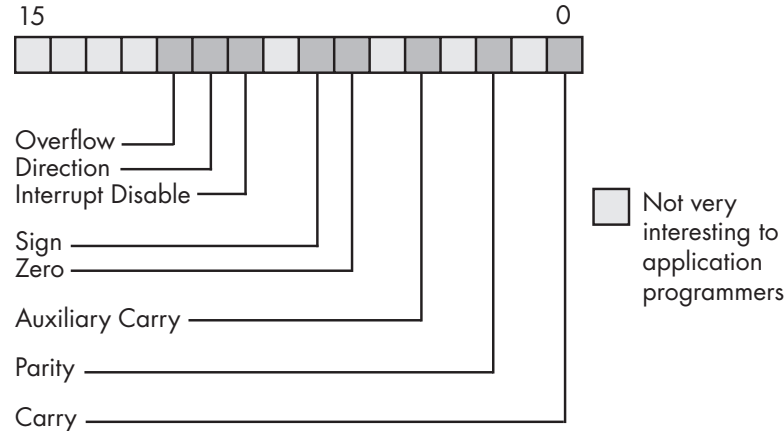


Figure 1-6: Layout of the FLAGS register (lower 16 bits of EFLAGS)

Of the eight flags that are of interest to application programmers, four flags in particular are extremely valuable: the overflow, carry, sign, and zero flags. Collectively, we will call these four flags the *condition codes*.<sup>5</sup> The state of these flags lets you test the result of previous computations. For example, after comparing two values, the condition code flags will tell you whether one value is less than, equal to, or greater than a second value.

<sup>4</sup> Application programs cannot modify the interrupt flag, but we'll look at this flag in Chapter 2; hence the discussion of this flag here.

<sup>5</sup> Technically the parity flag is also a condition code, but we will not use that flag in this text.

One important fact that comes as a surprise to those just learning assembly language is that almost all calculations on the 80x86 CPU involve a register. For example, to add two variables together, storing the sum into a third variable, you must load one of the variables into a register, add the second operand to the value in the register, and then store the register away in the destination variable. Registers are a middleman in nearly every calculation. Therefore, registers are very important in 80x86 assembly language programs.

Another thing you should be aware of is that although the registers have the name “general purpose,” you should not infer that you can use any register for any purpose. All the 80x86 registers have their own special purposes that limit their use in certain contexts. The SP/ESP register pair, for example, has a very special purpose that effectively prevents you from using it for anything else (it’s the *stack pointer*). Likewise, the BP/EBP register has a special purpose that limits its usefulness as a general-purpose register. For the time being, you should avoid the use of the ESP and EBP registers for generic calculations; also, keep in mind that the remaining registers are not completely interchangeable in your programs.

## 1.7 The Memory Subsystem

A typical 80x86 processor running a modern 32-bit OS can access a maximum of  $2^{32}$  different memory locations, or just over 4 billion bytes. A few years ago, 4 gigabytes of memory would have seemed like infinity; modern machines, however, exceed this limit. Nevertheless, because the 80x86 architecture supports a maximum 4GB address space when using a 32-bit operating system like Windows, Mac OS X, FreeBSD, or Linux, the following discussion will assume the 4GB limit.

Of course, the first question you should ask is, “What exactly is a memory location?” The 80x86 supports *byte-addressable memory*. Therefore, the basic memory unit is a byte, which is sufficient to hold a single character or a (very) small integer value (we’ll talk more about that in Chapter 2).

Think of memory as a linear array of bytes. The address of the first byte is 0 and the address of the last byte is  $2^{32}-1$ . For an 80x86 processor, the following pseudo-Pascal array declaration is a good approximation of memory:

---

```
Memory: array [0..4294967295] of byte;
```

---

C/C++ and Java users might prefer the following syntax:

---

```
byte Memory[4294967296];
```

---

To execute the equivalent of the Pascal statement `Memory [125] := 0;` the CPU places the value 0 on the data bus, places the address 125 on the address bus, and asserts the write line (this generally involves setting that line to 0), as shown in Figure 1-7.

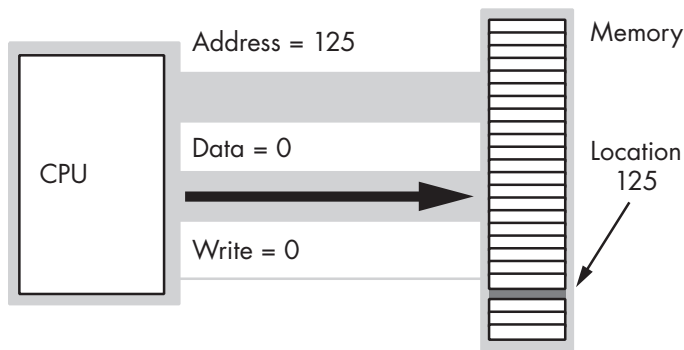


Figure 1-7: Memory write operation

To execute the equivalent of `CPU := Memory [125]`; the CPU places the address 125 on the address bus, asserts the read line (because the CPU is reading data from memory), and then reads the resulting data from the data bus (see Figure 1-8).

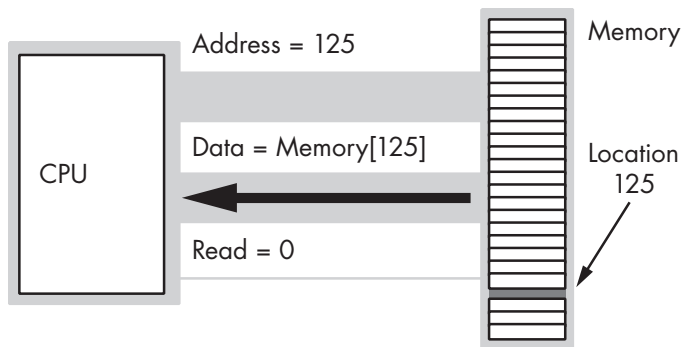


Figure 1-8: Memory read operation

This discussion applies *only* when accessing a single byte in memory. So what happens when the processor accesses a word or a double word? Because memory consists of an array of bytes, how can we possibly deal with values larger than a single byte? Easy—to store larger values, the 80x86 uses a sequence of consecutive memory locations. Figure 1-9 shows how the 80x86 stores bytes, words (2 bytes), and double words (4 bytes) in memory. The memory address of each of these objects is the address of the first byte of each object (that is, the lowest address).

Modern 80x86 processors don't actually connect directly to memory. Instead, there is a special memory buffer on the CPU known as the *cache* (pronounced “cash”) that acts as a high-speed intermediary between the CPU and main memory. Although the cache handles the details automatically for you, one fact you should know is that accessing data objects in memory is sometimes more efficient if the address of the object is an even multiple of the object's size. Therefore, it's a good idea to *align* 4-byte objects (double words) on addresses that are multiples of 4. Likewise, it's most

efficient to align 2-byte objects on even addresses. You can efficiently access single-byte objects at any address. You'll see how to set the alignment of memory objects in Section 3.4.

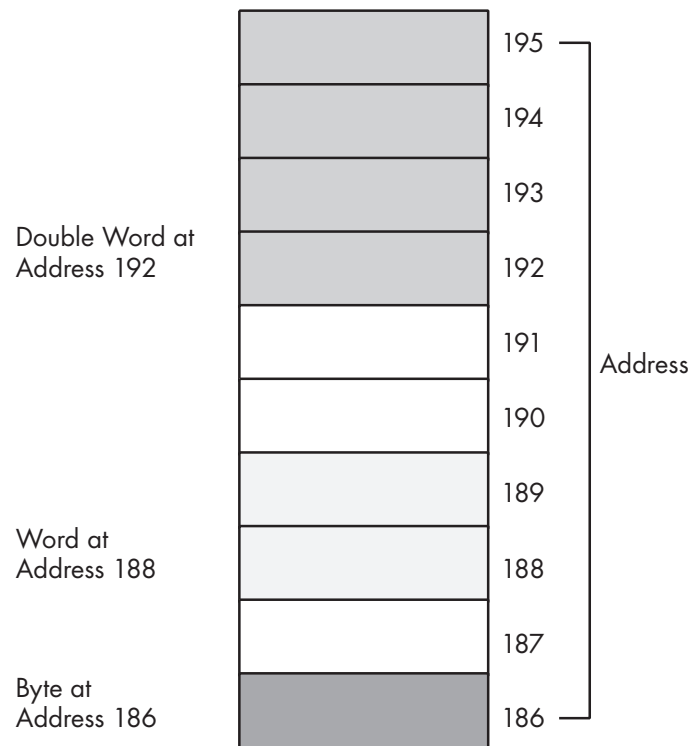


Figure 1-9: Byte, word, and double-word storage in memory

Before leaving this discussion of memory objects, it's important to understand the correspondence between memory and HLA variables. One of the nice things about using an assembler/compiler like HLA is that you don't have to worry about numeric memory addresses. All you need to do is declare a variable in HLA, and HLA takes care of associating that variable with some unique set of memory addresses. For example, if you have the following declaration section:

---

```
static
    i8      :int8;
    i16     :int16;
    i32     :int32;
```

---

HLA will find some unused 8-bit byte in memory and associate it with the `i8` variable; it will find a pair of consecutive unused bytes and associate `i16` with them; finally, HLA will find 4 consecutive unused bytes and associate the value of `i32` with those 4 bytes (32 bits). You'll always refer to these variables by their name. You generally don't have to concern yourself with their numeric address. Still, you should be aware that HLA is doing this for you behind your back.

## 1.8 Some Basic Machine Instructions

The 80x86 CPU family provides from just over a hundred to many thousands of different machine instructions, depending on how you define a machine instruction. Even at the low end of the count (greater than 100), it appears as though there are far too many machine instructions to learn in a short time. Fortunately, you don't need to know all the machine instructions. In fact, most assembly language programs probably use around 30 different machine instructions.<sup>6</sup> Indeed, you can certainly write several meaningful programs with only a few machine instructions. The purpose of this section is to provide a small handful of machine instructions so you can start writing simple HLA assembly language programs right away.

Without question, the `mov` instruction is the most oft-used assembly language statement. In a typical program, anywhere from 25 percent to 40 percent of the instructions are `mov` instructions. As its name suggests, this instruction moves data from one location to another.<sup>7</sup> The HLA syntax for this instruction is:

---

```
mov( source_operand, destination_operand );
```

---

The *source\_operand* can be a register, a memory variable, or a constant. The *destination\_operand* may be a register or a memory variable. Technically the 80x86 instruction set does not allow both operands to be memory variables. HLA, however, will automatically translate a `mov` instruction with two-word or double-word memory operands into a pair of instructions that will copy the data from one location to another. In a high-level language like Pascal or C/C++, the `mov` instruction is roughly equivalent to the following assignment statement:

---

```
destination_operand = source_operand ;
```

---

Perhaps the major restriction on the `mov` instruction's operands is that they must both be the same size. That is, you can move data between a pair of byte (8-bit) objects, word (16-bit) objects, or double-word (32-bit) objects; you may not, however, mix the sizes of the operands. Table 1-1 lists all the legal combinations for the `mov` instruction.

You should study this table carefully because most of the general-purpose 80x86 instructions use this syntax.

---

<sup>6</sup> Different programs may use a different set of 30 instructions, but few programs use more than 30 distinct instructions.

<sup>7</sup> Technically, `mov` actually copies data from one location to another. It does not destroy the original data in the source operand. Perhaps a better name for this instruction would have been `copy`. Alas, it's too late to change it now.

**Table 1-1:** Legal 80x86 mov Instruction Operands

Source	Destination
Reg <sub>8</sub> <sup>*</sup>	Reg <sub>8</sub>
Reg <sub>8</sub>	Mem <sub>8</sub>
Mem <sub>8</sub>	Reg <sub>8</sub>
Constant <sup>†</sup>	Reg <sub>8</sub>
Constant	Mem <sub>8</sub>
Reg <sub>16</sub>	Reg <sub>16</sub>
Reg <sub>16</sub>	Mem <sub>16</sub>
Mem <sub>16</sub>	Reg <sub>16</sub>
Constant	Reg <sub>16</sub>
Constant	Mem <sub>16</sub>
Reg <sub>32</sub>	Reg <sub>32</sub>
Reg <sub>32</sub>	Mem <sub>32</sub>
Mem <sub>32</sub>	Reg <sub>32</sub>
Constant	Reg <sub>32</sub>
Constant	Mem <sub>32</sub>

<sup>\*</sup> The suffix denotes the size of the register or memory location.

<sup>†</sup> The constant must be small enough to fit in the specified destination operand.

The 80x86 add and sub instructions let you add and subtract two operands. Their syntax is nearly identical to the mov instruction:

---

```
add( source_operand, destination_operand );
sub( source_operand, destination_operand );
```

---

The add and sub operands take the same form as the mov instruction.<sup>8</sup> The add instruction does the following:

---

```
destination_operand = destination_operand + source_operand ;
destination_operand += source_operand; // For those who prefer C syntax.
```

---

The sub instruction does the calculation:

---

```
destination_operand = destination_operand - source_operand ;
destination_operand -= source_operand ; // For C fans.
```

---

With nothing more than these three instructions, plus the HLA control structures that the next section discusses, you can actually write some sophisticated programs. Listing 1-3 provides a sample HLA program that demonstrates these three instructions.

<sup>8</sup> Remember, though, that add and sub do not support memory-to-memory operations.

---

```

program DemoMOVaddSUB;

#include( "stdlib.hhf" )

static
    i8:      int8      := -8;
    i16:     int16     := -16;
    i32:     int32     := -32;

begin DemoMOVaddSUB;

    // First, print the initial values
    // of our variables.

    stdout.put
    (
        nl,
        "Initialized values: i8=", i8,
        ", i16=", i16,
        ", i32=", i32,
        nl
    );

    // Compute the absolute value of the
    // three different variables and
    // print the result.
    // Note: Because all the numbers are
    // negative, we have to negate them.
    // Using only the mov, add, and sub
    // instructions, we can negate a value
    // by subtracting it from zero.

    mov( 0, al ); // Compute i8 := -i8;
    sub( i8, al );
    mov( al, i8 );

    mov( 0, ax ); // Compute i16 := -i16;
    sub( i16, ax );
    mov( ax, i16 );

    mov( 0, eax ); // Compute i32 := -i32;
    sub( i32, eax );
    mov( eax, i32 );

    // Display the absolute values:

    stdout.put
    (
        nl,
        "After negation: i8=", i8,
        ", i16=", i16,
        ", i32=", i32,
        nl
    )

```



```

    );

    // Demonstrate add and constant-to-memory
    // operations:

    add( 32323200, i32 );
    stdout.put( nl, "After add: i32=", i32, nl );

end DemoMOVaddSUB;

```

---

*Listing 1-3: Demonstration of the mov, add, and sub instructions*

## 1.9 Some Basic HLA Control Structures

The `mov`, `add`, and `sub` instructions, while valuable, aren't sufficient to let you write meaningful programs. You will need to complement these instructions with the ability to make decisions and create loops in your HLA programs before you can write anything other than a simple program. HLA provides several high-level control structures that are very similar to control structures found in high-level languages. These include `if..then..elseif..else..endif`, `while..endwhile`, `repeat..until`, and so on. By learning these statements you will be armed and ready to write some real programs.

Before discussing these high-level control structures, it's important to point out that these are not real 80x86 assembly language statements. HLA compiles these statements into a sequence of one or more real assembly language statements for you. In Chapter 7, you'll learn how HLA compiles the statements, and you'll learn how to write pure assembly language code that doesn't use them. However, there is a lot to learn before you get to that point, so we'll stick with these high-level language statements for now.

Another important fact to mention is that HLA's high-level control structures are *not* as high level as they first appear. The purpose behind HLA's high-level control structures is to let you start writing assembly language programs as quickly as possible, not to let you avoid the use of assembly language altogether. You will soon discover that these statements have some severe restrictions associated with them, and you will quickly outgrow their capabilities. This is intentional. Once you reach a certain level of comfort with HLA's high-level control structures and decide you need more power than they have to offer, it's time to move on and learn the real 80x86 instructions behind these statements.

Do not let the presence of high-level-like statements in HLA confuse you. Many people, after learning about the presence of these statements in the HLA language, erroneously come to the conclusion that HLA is just some special high-level language and not a true assembly language. This isn't true. HLA is a full low-level assembly language. HLA supports all the same machine instructions as any other 80x86 assembler. The difference is that HLA has some *extra* statements that allow you to do *more* than is possible with those other 80x86 assemblers. Once you learn 80x86 assembly

language with HLA, you may elect to ignore all these extra (high-level) statements and write only low-level 80x86 assembly language code if this is your desire.

The following sections assume that you're familiar with at least one high-level language. They present the HLA control statements from that perspective without bothering to explain how you actually use these statements to accomplish something in a program. One prerequisite this text assumes is that you already know how to use these generic control statements in a high-level language; you'll use them in HLA programs in an identical manner.

### 1.9.1 Boolean Expressions in HLA Statements

Several HLA statements require a boolean (true or false) expression to control their execution. Examples include the `if`, `while`, and `repeat...until` statements. The syntax for these boolean expressions represents the greatest limitation of the HLA high-level control structures. This is one area where your familiarity with a high-level language will work against you—you'll want to use the fancy expressions you use in a high-level language, yet HLA supports only some basic forms.

HLA boolean expressions take the following forms:<sup>9</sup>

---

`flag_specification`  
`!flag_specification`  
`register`  
`!register`  
`Boolean_variable`  
`!Boolean_variable`  
`mem_reg relop mem_reg_const`  
`register in LowConst..HiConst`  
`register not in LowConst..HiConst`

---

A `flag_specification` may be one of the symbols that are described in Table 1-2.

**Table 1-2:** Symbols for `flag_specification`

Symbol	Meaning	Explanation
@c	Carry	True if the carry is set (1); false if the carry is clear (0).
@nc	No carry	True if the carry is clear (0); false if the carry is set (1).
@z	Zero	True if the zero flag is set; false if it is clear.
@nz	Not zero	True if the zero flag is clear; false if it is set.
@o	Overflow	True if the overflow flag is set; false if it is clear.
@no	No overflow	True if the overflow flag is clear; false if it is set.
@s	Sign	True if the sign flag is set; false if it is clear.
@ns	No sign	True if the sign flag is clear; false if it is set.

<sup>9</sup> There are a few additional forms that we'll cover in Chapter 6.

The use of the flag values in a boolean expression is somewhat advanced. You will begin to see how to use these boolean expression operands in the next chapter.

A register operand can be any of the 8-bit, 16-bit, or 32-bit general-purpose registers. The expression evaluates false if the register contains a zero; it evaluates true if the register contains a nonzero value.

If you specify a boolean variable as the expression, the program tests it for zero (false) or nonzero (true). Because HLA uses the values zero and one to represent false and true, respectively, the test works in an intuitive fashion. Note that HLA requires such variables be of type `boolean`. HLA rejects other data types. If you want to test some other type against zero/not zero, then use the general boolean expression discussed next.

The most general form of an HLA boolean expression has two operands and a relational operator. Table 1-3 lists the legal combinations.

**Table 1-3:** Legal Boolean Expressions

Left Operand	Relational Operator	Right Operand
Memory variable or register	= or ==	Variable, register, or constant
	<> or !=	
	<	
	<=	
	>	
	>=	

Note that both operands cannot be memory operands. In fact, if you think of the *right operand* as the source operand and the *left operand* as the destination operand, then the two operands must be the same that `add` and `sub` allow.

Also like the `add` and `sub` instructions, the two operands must be the same size. That is, they must both be byte operands, they must both be word operands, or they must both be double-word operands. If the right operand is a constant, its value must be in the range that is compatible with the left operand.

There is one other issue: if the left operand is a register and the right operand is a positive constant or another register, HLA uses an *unsigned* comparison. The next chapter will discuss the ramifications of this; for the time being, do not compare negative values in a register against a constant or another register. You may not get an intuitive result.

The `in` and `not in` operators let you test a register to see if it is within a specified range. For example, the expression `eax in 2000..2099` evaluates true if the value in the EAX register is between 2,000 and 2,099 (inclusive). The `not in` (two words) operator checks to see if the value in a register is outside the specified range. For example, `al not in 'a'..'z'` evaluates true if the character in the AL register is not a lowercase alphabetic character.

Here are some examples of legal boolean expressions in HLA:

---

```
@c
Bool_var
al
esi
eax < ebx
ebx > 5
i32 < -2
i8 > 128
al < i8
eax in 1..100
ch not in 'a'..'z'
```

---

### 1.9.2 The HLA *if..then..elseif..else..endif* Statement

The HLA if statement uses the syntax shown in Figure 1-10.

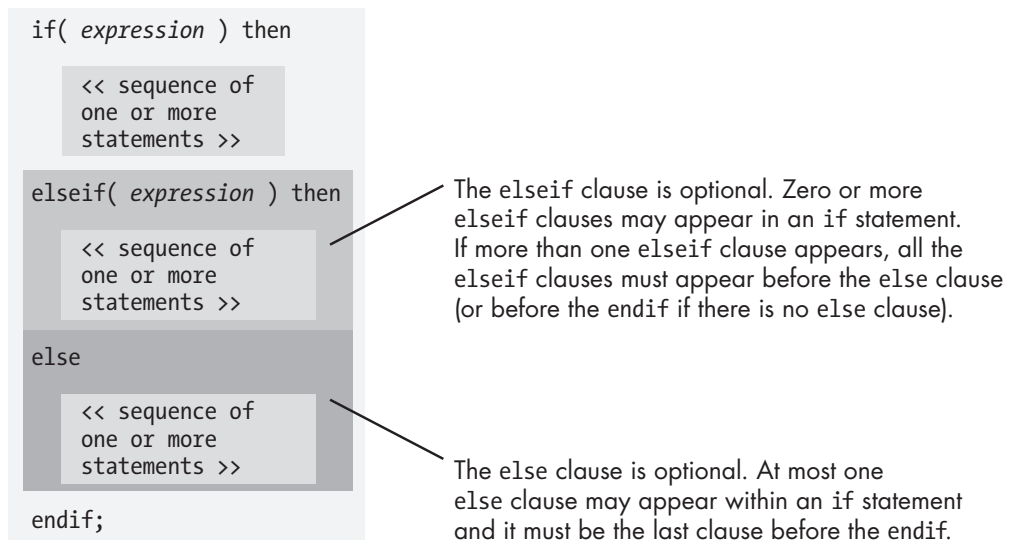


Figure 1-10: HLA if statement syntax

The expressions appearing in an if statement must take one of the forms from the previous section. If the boolean expression is true, the code after the then executes; otherwise control transfers to the next elseif or else clause in the statement.

Because the elseif and else clauses are optional, an if statement could take the form of a single if..then clause, followed by a sequence of statements and a closing endif clause. The following is such a statement:

---

```
if( eax = 0 ) then

    stdout.put( "error: NULL value", nl );

endif;
```

---

If, during program execution, the expression evaluates true, then the code between the then and the endif executes. If the expression evaluates false, then the program skips over the code between the then and the endif.

Another common form of the if statement has a single else clause. The following is an example of an if statement with an optional else clause:

---

```
if( eax = 0 ) then

    stdout.put( "error: NULL pointer encountered", nl );

else

    stdout.put( "Pointer is valid", nl );

endif;
```

---

If the expression evaluates true, the code between the then and the else executes; otherwise the code between the else and the endif clauses executes.

You can create sophisticated decision-making logic by incorporating the elseif clause into an if statement. For example, if the CH register contains a character value, you can select from a menu of items using code like the following:

---

```
if( ch = 'a' ) then

    stdout.put( "You selected the 'a' menu item", nl );

elseif( ch = 'b' ) then

    stdout.put( "You selected the 'b' menu item", nl );

elseif( ch = 'c' ) then

    stdout.put( "You selected the 'c' menu item", nl );

else

    stdout.put( "Error: illegal menu item selection", nl );

endif;
```

---

Although this simple example doesn't demonstrate it, HLA does not require an else clause at the end of a sequence of elseif clauses. However, when making multiway decisions, it's always a good idea to provide an else clause just in case an error arises. Even if you think it's impossible for the else clause to execute, just keep in mind that future modifications to the code could void this assertion, so it's a good idea to have error-reporting statements in your code.

### 1.9.3 Conjunction, Disjunction, and Negation in Boolean Expressions

Some obvious omissions in the list of operators in the previous sections are the conjunction (logical and), disjunction (logical or), and negation (logical not) operators. This section describes their use in boolean expressions (the discussion had to wait until after describing the if statement in order to present realistic examples).

HLA uses the `&&` operator to denote logical and in a runtime boolean expression. This is a dyadic (two-operand) operator, and the two operands must be legal runtime boolean expressions. This operator evaluates to true if both operands evaluate to true. For example:

---

```
if( eax > 0 && ch = 'a' ) then

    mov( eax, ebx );
    mov( ' ', ch );

endif;
```

---

The two `mov` statements above execute only if `EAX` is greater than zero *and* `CH` is equal to the character *a*. If either of these conditions is false, then program execution skips over these `mov` instructions.

Note that the expressions on either side of the `&&` operator may be any legal boolean expressions; these expressions don't have to be comparisons using the relational operators. For example, the following are all legal expressions:

---

```
@z && al in 5..10
al in 'a'..'z' && ebx
boolVar && !eax
```

---

HLA uses *short-circuit evaluation* when compiling the `&&` operator. If the leftmost operand evaluates false, then the code that HLA generates does not bother evaluating the second operand (because the whole expression must be false at that point). Therefore, in the last expression above, the code will not check `EAX` against zero if `boolVar` evaluates false.

Note that an expression like `eax < 10 && ebx <> eax` is itself a legal boolean expression and, therefore, may appear as the left or right operand of the `&&` operator. Therefore, expressions like the following are perfectly legal:

---

```
eax < 0 && ebx <> eax && !ecx
```

---

The `&&` operator is left associative, so the code that HLA generates evaluates the expression above in a left-to-right fashion. If `EAX` is less than zero, the CPU will not test either of the remaining expressions. Likewise, if `EAX` is not less than zero but `EBX` is equal to `EAX`, this code will not evaluate the third expression because the whole expression is false regardless of `ECX`'s value.

HLA uses the `||` operator to denote disjunction (logical or) in a runtime boolean expression. Like the `&&` operator, this operator expects two legal runtime boolean expressions as operands. This operator evaluates true if either (or both) operands evaluate true. Like the `&&` operator, the disjunction operator uses short-circuit evaluation. If the left operand evaluates true, then the code that HLA generates doesn't bother to test the value of the second operand. Instead, the code will transfer to the location that handles the situation when the boolean expression evaluates true. Here are some examples of legal expressions using the `||` operator:

---

```
@z || al = 10
al in 'a'..'z' || ebx
!boolVar || eax
```

---

Like the `&&` operator, the disjunction operator is left associative, so multiple instances of the `||` operator may appear within the same expression. Should this be the case, the code that HLA generates will evaluate the expressions from left to right. For example:

---

```
eax < 0 || ebx <> eax || !ecx
```

---

The code above evaluates to true if EAX is less than zero, EBX does not equal EAX, or ECX is zero. Note that if the first comparison is true, the code doesn't bother testing the other conditions. Likewise, if the first comparison is false and the second is true, the code doesn't bother checking to see if ECX is zero. The check for ECX equal to zero occurs only if the first two comparisons are false.

If both the conjunction and disjunction operators appear in the same expression, then the `&&` operator takes precedence over the `||` operator. Consider the following expression:

---

```
eax < 0 || ebx <> eax && !ecx
```

---

The machine code HLA generates evaluates this as

---

```
eax < 0 || (ebx <> eax && !ecx)
```

---

If EAX is less than zero, then the code HLA generates does not bother to check the remainder of the expression, and the entire expression evaluates true. However, if EAX is not less than zero, then both of the following conditions must evaluate true in order for the overall expression to evaluate true.

HLA allows you to use parentheses to surround subexpressions involving `&&` and `||` if you need to adjust the precedence of the operators. Consider the following expression:

---

```
(eax < 0 || ebx <> eax) && !ecx
```

---

For this expression to evaluate true, ECX must contain zero and either EAX must be less than zero or EBX must not equal EAX. Contrast this to the result the expression produces without the parentheses.

HLA uses the `!` operator to denote logical negation. However, the `!` operator may only prefix a register or boolean variable; you may not use it as part of a larger expression (e.g., `!eax < 0`). To achieve logical negative of an existing boolean expression, you must surround that expression with parentheses and prefix the parentheses with the `!` operator. For example:

---

```
!( eax < 0 )
```

---

This expression evaluates true if EAX is not less than zero.

The logical not operator is primarily useful for surrounding complex expressions involving the conjunction and disjunction operators. While it is occasionally useful for short expressions like the one above, it's usually easier (and more readable) to simply state the logic directly rather than convolute it with the logical not operator.

Note that HLA also provides the `|` and `&` operators, but they are distinct from `||` and `&&` and have completely different meanings. See the HLA reference manual for more details on these (compile-time) operators.

### 1.9.4 The *while..endwhile* Statement

The while statement uses the basic syntax shown in Figure 1-11.

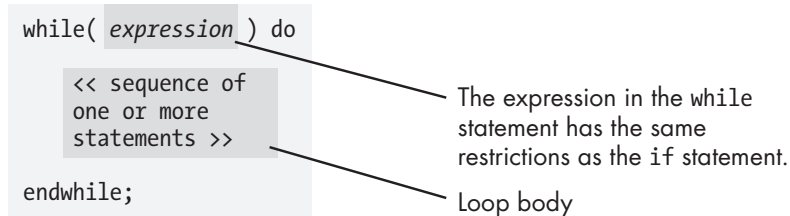


Figure 1-11: HLA *while* statement syntax

This statement evaluates the boolean expression. If it is false, control immediately transfers to the first statement following the `endwhile` clause. If the value of the expression is true, then the CPU executes the body of the loop. After the loop body executes, control transfers back to the top of the loop, where the `while` statement retests the loop control expression. This process repeats until the expression evaluates false.

Note that the `while` loop, like its high-level-language counterpart, tests for loop termination at the top of the loop. Therefore, it is quite possible that the statements in the body of the loop will not execute (if the expression is false when the code first executes the `while` statement). Also note that the body of the `while` loop must, at some point, modify the value of the boolean expression or an infinite loop will result.



Here's an example of an HLA while loop:

---

```
mov( 0, i );
while( i < 10 ) do

    stdout.put( "i=", i, nl );
    add( 1, i );

endwhile;
```

---

### **1.9.5 The *for..endfor* Statement**

The HLA for loop takes the following general form:

---

```
for( Initial_Smt; Termination_Expression; Post_Body_Statement ) do

    << Loop body >>

endfor;
```

---

This is equivalent to the following while statement:

---

```
Initial_Smt;
while( Termination_Expression ) do

    << Loop body >>

    Post_Body_Statement;

endwhile;
```

---

*Initial\_Smt* can be any single HLA/80x86 instruction. Generally this statement initializes a register or memory location (the loop counter) with zero or some other initial value. *Termination\_Expression* is an HLA boolean expression (same format that while allows). This expression determines whether the loop body executes. *Post\_Body\_Statement* executes at the bottom of the loop (as shown in the while example above). This is a single HLA statement. Usually an instruction like add modifies the value of the loop control variable.

The following gives a complete example:

---

```
for( mov( 0, i ); i < 10; add(1, i ) ) do

    stdout.put( "i=", i, nl );

endfor;
```

---

The above, rewritten as a while loop, becomes:

---

```
mov( 0, i );
while( i < 10 ) do

    stdout.put( "i=", i, nl );

    add( 1, i );

endwhile;
```

---

### 1.9.6 The repeat..until Statement

The HLA repeat..until statement uses the syntax shown in Figure 1-12. C/C++/C# and Java users should note that the repeat..until statement is very similar to the do..while statement.

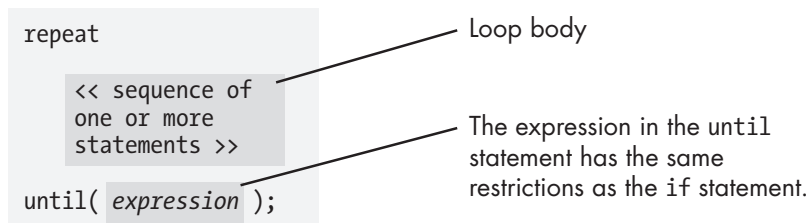


Figure 1-12: HLA repeat..until statement syntax

The HLA repeat..until statement tests for loop termination at the bottom of the loop. Therefore, the statements in the loop body always execute at least once. Upon encountering the until clause, the program will evaluate the expression and repeat the loop if the expression is false (that is, it repeats while false). If the expression evaluates true, the control transfers to the first statement following the until clause.

The following simple example demonstrates the repeat..until statement:

---

```
mov( 10, ecx );
repeat

    stdout.put( "ecx = ", ecx, nl );
    sub( 1, ecx );

until( ecx = 0 );
```

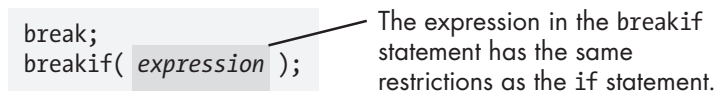
---

If the loop body will always execute at least once, then it is usually more efficient to use a repeat..until loop rather than a while loop.

## 1.9.7 The break and breakif Statements

The break and breakif statements provide the ability to prematurely exit from a loop. Figure 1-13 shows the syntax for these two statements.

```
break;  
breakif( expression );
```



The expression in the breakif statement has the same restrictions as the if statement.

Figure 1-13: HLA break and breakif syntax

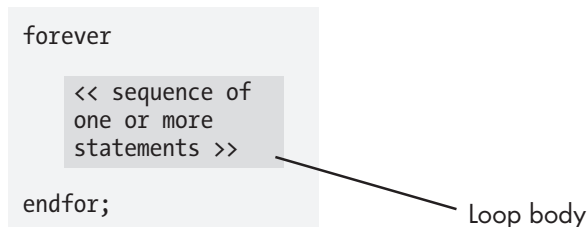
The break statement exits the loop that immediately contains the break. The breakif statement evaluates the boolean expression and exits the containing loop if the expression evaluates true.

Note that the break and breakif statements do not allow you to break out of more than one nested loop. HLA does provide statements that do this, the begin..end block and the exit/exitif statements. Please consult the HLA reference manual for more details. HLA also provides the continue/continueif pair that lets you repeat a loop body. Again, see the HLA reference manual for more details.

## 1.9.8 The forever..endfor Statement

Figure 1-14 shows the syntax for the forever statement.

```
forever  
    << sequence of  
    one or more  
    statements >>  
endfor;
```



Loop body

Figure 1-14: HLA forever loop syntax

This statement creates an infinite loop. You may also use the break and breakif statements along with forever..endfor to create a loop that tests for loop termination in the middle of the loop. Indeed, this is probably the most common use of this loop, as the following example demonstrates:

---

```
forever  
  
    stdout.put( "Enter an integer less than 10: ");  
    stdin.get( i );  
    breakif( i < 10 );  
    stdout.put( "The value needs to be less than 10!", nl );  
  
endfor;
```

---

### 1.9.9 The `try..exception..endtry` Statement

The HLA `try..exception..endtry` statement provides very powerful *exception handling* capabilities. The syntax for this statement appears in Figure 1-15.

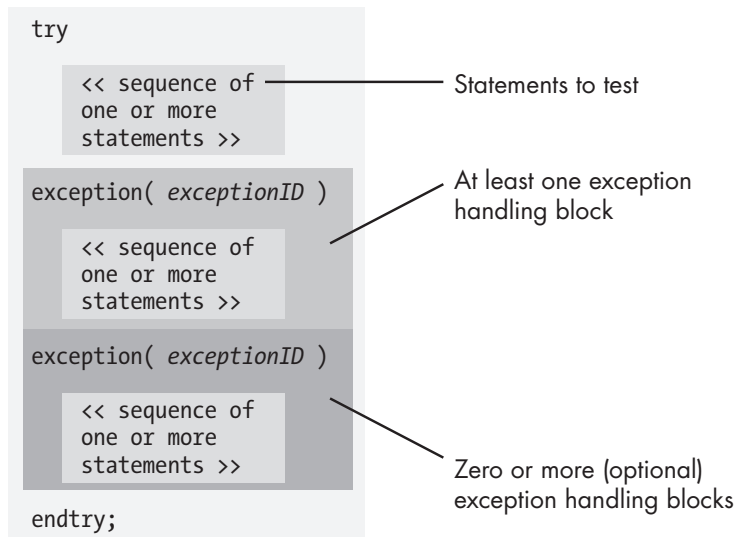


Figure 1-15: HLA `try..exception..endtry` statement syntax

The `try..endtry` statement protects a block of statements during execution. If the statements between the `try` clause and the first `exception` clause (the *protected block*), execute without incident, control transfers to the first statement after the `endtry` immediately after executing the last statement in the protected block. If an error (exception) occurs, then the program interrupts control at the point of the exception (that is, the program *raises* an exception). Each exception has an unsigned integer constant associated with it, known as the *exception ID*. The `excepts.hhf` header file in the HLA Standard Library predefines several exception IDs, although you may create new ones for your own purposes. When an exception occurs, the system compares the exception ID against the values appearing in each of the exception clauses following the protected code. If the current exception ID matches one of the exception values, control continues with the block of statements immediately following that exception. After the exception-handling code completes execution, control transfers to the first statement following the `endtry`.

If an exception occurs and there is no active `try..endtry` statement, or the active `try..endtry` statements do not handle the specific exception, the program will abort with an error message.

The following code fragment demonstrates how to use the `try..endtry` statement to protect the program from bad user input:

---

```
repeat

    mov( false, GoodInteger );    // Note: GoodInteger must be a boolean var.
    try

        stdout.put( "Enter an integer: " );
```

```

        stdin.get( i );
        mov( true, GoodInteger );

    exception( ex.ConversionError );

        stdout.put( "Illegal numeric value, please re-enter", nl );

    exception( ex.ValueOutOfRange );

        stdout.put( "Value is out of range, please re-enter", nl );

    endtry;

until( GoodInteger );

```

---

The repeat..until loop repeats this code as long as there is an error during input. Should an exception occur because of bad input, control transfers to the exception clauses to see if a conversion error (e.g., illegal characters in the number) or a numeric overflow occurs. If either of these exceptions occur, then they print the appropriate message, control falls out of the try..endtry statement, and the repeat..until loop repeats because the code will not have set GoodInteger to true. If a different exception occurs (one that is not handled in this code), then the program aborts with the specified error message.<sup>10</sup>

Table 1-4 lists the exceptions provided in the *excepts.hhf* header file at the time this was being written. See the *excepts.hhf* header file provided with HLA for the most current list of exceptions.

**Table 1-4:** Exceptions Provided in *excepts.hhf*

Exception	Description
ex.StringOverflow	Attempt to store a string that is too large into a string variable.
ex.StringIndexError	Attempt to access a character that is not present in a string.
ex.StringOverlap	Attempt to copy a string onto itself.
ex.StringMetaData	Corrupted string value.
ex.StringAlignment	Attempt to store a string at an unaligned address.
ex.StringUnderflow	Attempt to extract "negative" characters from a string.
ex.IllegalStringOperation	Operation not permitted on string data.
ex.ValueOutOfRange	Value is too large for the current operation.
ex.IllegalChar	Operation encountered a character code whose ASCII code is not in the range 0..127.
ex.TooManyCmdLnParms	Command line contains too many program parameters.
ex.BadObjPtr	Pointer to class object is illegal.

(continued)

---

<sup>10</sup> An experienced programmer may wonder why this code uses a boolean variable rather than a breakif statement to exit the repeat..until loop. There are some technical reasons for this that you will learn about in Section 1.11.

**Table 1-4:** Exceptions Provided in *excepts.hhf* (continued)

Exception	Description
<code>ex.InvalidAlignment</code>	Argument was not aligned on a proper memory address.
<code>ex.InvalidArgument</code>	Function call (generally OS API call) contains an invalid argument value.
<code>ex.BufferOverflow</code>	Buffer or blob object exceeded declared size.
<code>ex.BufferUnderflow</code>	Attempt to retrieve nonexistent data from a blob or buffer.
<code>ex.IllegalSize</code>	Argument's data size is incorrect.
<code>ex.ConversionError</code>	String-to-numeric conversion operation contains illegal (nonnumeric) characters.
<code>ex.BadFileHandle</code>	Program attempted a file access using an invalid file handle value.
<code>ex.FileNotFound</code>	Program attempted to access a nonexistent file.
<code>ex.FileOpenFailure</code>	Operating system could not open the file (file not found).
<code>ex.FileCloseError</code>	Operating system could not close the file.
<code>ex.FileWriteError</code>	Error writing data to a file.
<code>ex.FileReadError</code>	Error reading data from a file.
<code>ex.FileSeekError</code>	Attempted to seek to a nonexistent position in a file.
<code>ex.DiskFullError</code>	Attempted to write data to a full disk.
<code>ex.AccessDenied</code>	User does not have sufficient privileges to access file data.
<code>ex.EndOfFile</code>	Program attempted to read beyond the end of file.
<code>ex.CannotCreateDir</code>	Attempt to create a directory failed.
<code>ex.CannotRemoveDir</code>	Attempt to delete a directory failed.
<code>ex.CannotRemoveFile</code>	Attempt to delete a file failed.
<code>ex.CDFailed</code>	Attempt to change to a new directory failed.
<code>ex.CannotRenameFile</code>	Attempt to rename a file failed.
<code>ex.MemoryAllocationFailure</code>	Insufficient system memory for allocation request.
<code>ex.MemoryFreeFailure</code>	Could not free the specified memory block (corrupted memory management system).
<code>ex.MemoryAllocationCorruption</code>	Corrupted memory management system.
<code>ex.AttemptToFreeNULL</code>	Caller attempted to free a NULL pointer.
<code>ex.AttemptToDerefNULL</code>	Program attempted to access data indirectly using a NULL pointer.
<code>ex.BlockAlreadyFree</code>	Caller attempted to free a block that was already freed.
<code>ex.CannotFreeMemory</code>	Memory free operation failure.
<code>ex.PointerNotInHeap</code>	Caller attempted to free a block of memory that was not allocated on the heap.
<code>ex.WidthTooBig</code>	Format width for numeric to string conversion was too large.
<code>ex.FractionTooBig</code>	Format size for fractional portion in floating-point-to-string conversion was too large.
<code>ex.ArrayShapeViolation</code>	Attempted operation on two arrays whose dimensions don't match.

**Table 1-4:** Exceptions Provided in *excepts.hhf* (continued)

Exception	Description
ex.ArrayBounds	Attempted to access an element of an array, but the index was out of bounds.
ex.InvalidDate	Attempted date operation with an illegal date.
ex.InvalidDateFormat	Conversion from string to date contains illegal characters.
ex.TimeOverflow	Overflow during time arithmetic.
ex.InvalidTime	Attempted time operation with an illegal time.
ex.InvalidTimeFormat	Conversion from string to time contains illegal characters.
ex.SocketError	Network communication failure.
ex.ThreadError	Generic thread (multitasking) error.
ex.AssertionFailed	assert statement encountered a failed assertion.
ex.ExecutedAbstract	Attempt to execute an abstract class method.
ex.AccessViolation	Attempt to access an illegal memory location.
ex.InPageError	OS memory access error.
ex.NoMemory	OS memory failure.
ex.InvalidHandle	Bad handle passed to OS API call.
ex.ControlC	CTRL-C was pressed on system console (functionality is OS specific).
ex.Breakpoint	Program executed a breakpoint instruction (INT 3).
ex.SingleStep	Program is operating with the trace flag set.
ex.PrivInstr	Program attempted to execute a kernel-only instruction.
ex.IllegalInstr	Program attempted to execute an illegal machine instruction.
ex.BoundInstr	Bound instruction execution with “out of bounds” value.
ex.IntoInstr	Into instruction execution with the overflow flag set.
ex.DivideError	Program attempted division by zero or other divide error.
ex.fDenormal	Floating point exception (see Chapter 6).
ex.fDivByZero	Floating point exception (see Chapter 6).
ex.fInexactResult	Floating point exception (see Chapter 6).
ex.fInvalidOperation	Floating point exception (see Chapter 6).
ex.fOverflow	Floating point exception (see Chapter 6).
ex.fStackCheck	Floating point exception (see Chapter 6).
ex.fUnderflow	Floating point exception (see Chapter 6).
ex.InvalidHandle	OS reported an invalid handle for some operation.

Most of these exceptions occur in situations that are well beyond the scope of this chapter. Their appearance here is strictly for completeness. See the HLA reference manual, the HLA Standard Library documentation, and the HLA Standard Library source code for more details concerning these exceptions. The `ex.ConversionError`, `ex.ValueOutOfRange`, and `ex.StringOverflow` exceptions are the ones you’ll most commonly use.

We’ll return to the discussion of the `try...endtry` statement in Section 1.11. First, however, we need to cover a little more material.

## 1.10 Introduction to the HLA Standard Library

There are two reasons HLA is much easier to learn and use than standard assembly language. The first reason is HLA's high-level syntax for declarations and control structures. This leverages your high-level language knowledge, allowing you to learn assembly language more efficiently. The other half of the equation is the HLA Standard Library. The HLA Standard Library provides many common, easy-to-use, assembly language routines that you can call without having to write this code yourself (and, more importantly, having to learn how to write yourself). This eliminates one of the larger stumbling blocks many people have when learning assembly language: the need for sophisticated I/O and support code in order to write basic statements. Prior to the advent of a standardized assembly language library, it often took considerable study before a new assembly language programmer could do as much as print a string to the display. With the HLA Standard Library, this roadblock is removed, and you can concentrate on learning assembly language concepts rather than learning low-level I/O details that are specific to a given operating system.

A wide variety of library routines is only part of HLA's support. After all, assembly language libraries have been around for quite some time.<sup>11</sup> HLA's Standard Library complements HLA by providing a high-level language interface to these routines. Indeed, the HLA language itself was originally designed specifically to allow the creation of a high-level set of library routines. This high-level interface, combined with the high-level nature of many of the routines in the library, packs a surprising amount of power in an easy-to-use package.

The HLA Standard Library consists of several modules organized by category. Table 1-5 lists many of the modules that are available.<sup>12</sup>

**Table 1-5:** HLA Standard Library Modules

Name	Description
args	Command-line parameter-parsing support routines.
arrays	Array declarations and operations.
bits	Bit-manipulation functions.
blobs	Binary large objects—operations on large blocks of binary data.
bsd	OS API calls for FreeBSD (HLA FreeBSD version only).
chars	Operations on character data.
console	Portable console (text screen) operations (cursor movement, screen clears, etc.).
conv	Various conversions between strings and other values.
coroutines	Support for coroutines ("cooperative multitasking").
cset	Character set functions.
DateTime	Calendar, date, and time functions.

<sup>11</sup> For example, see the UCR Standard Library for 80x86 Assembly Language Programmers.

<sup>12</sup> Because the HLA Standard Library is expanding, this list is probably out of date. See the HLA documentation for a current list of Standard Library modules.



**Table 1-5:** HLA Standard Library Modules (continued)

Name	Description
env	Access to OS environment variables.
excepts	Exception-handling routines.
fileclass	Object-oriented file input and output.
fileio	File input and output routines.
filesys	Access to the OS file system.
hla	Special HLA constants and other values.
linux	Linux system calls (HLA Linux version only).
lists	An HLA class for manipulating linked lists.
mac	OS API calls for Mac OS X (HLA Mac OS X version only).
math	Extended-precision arithmetic, transcendental functions, and other mathematical functions.
memmap	Memory-mapped file operations.
memory	Memory allocation, deallocation, and support code.
patterns	The HLA pattern-matching library.
random	Pseudo-random number generators and support code.
sockets	A set of network communication functions and classes.
stderr	Provides user output and several other support functions.
stdin	User input routines.
stdio	A support module for stderr, stdin, and stdout.
stdout	Provides user output and several other support routines.
strings	HLA's powerful string library.
tables	Table (associative array) support routines.
threads	Support for multithreaded applications and process synchronization.
timers	Support for timing events in an application.
win32	Constants used in Windows calls (HLA Windows version only).
x86	Constants and other items specific to the 80x86 CPU.

Later sections of this text will explain many of these modules in greater detail. This section will concentrate on the most important routines (at least to beginning HLA programmers), the `stdio` library.

### 1.10.1 *Predefined Constants in the `stdio` Module*

Perhaps the first place to start is with a description of some common constants that the `stdio` module defines for you. Consider the following (typical) example:

---

```
stdout.put( "Hello World", nl );
```

---

The `nl` appearing at the end of this statement stands for *newline*. The `nl` identifier is not a special HLA reserved word, nor is it specific to the `stdout.put` statement. Instead, it's simply a predefined constant that corresponds to the

string containing the standard end-of-line sequence (a carriage return/line feed pair under Windows or just a line feed under Linux, FreeBSD, and Mac OS X).

In addition to the `nl` constant, the HLA standard I/O library module defines several other useful character constants, as listed in Table 1-6.

**Table 1-6:** Character Constants Defined by the HLA Standard I/O Library

Character	Definition
<code>stdio.bell</code>	The ASCII bell character; beeps the speaker when printed
<code>stdio.bs</code>	The ASCII backspace character
<code>stdio.tab</code>	The ASCII tab character
<code>stdio.lf</code>	The ASCII linefeed character
<code>stdio.cr</code>	The ASCII carriage return character

Except for `nl`, these characters appear in the `stdio` namespace<sup>13</sup> (and therefore require the `stdio.` prefix). The placement of these ASCII constants within the `stdio` namespace helps avoid naming conflicts with your own variables. The `nl` name does not appear within a namespace because you will use it very often, and typing `stdio.nl` would get tiresome very quickly.

### 1.10.2 *Standard In and Standard Out*

Many of the HLA I/O routines have a `stdin` or `stdout` prefix. Technically, this means that the standard library defines these names in a namespace. In practice, this prefix suggests where the input is coming from (the standard input device) or going to (the standard output device). By default, the standard input device is the system keyboard. Likewise, the default standard output device is the console display. So, in general, statements that have `stdin` or `stdout` prefixes will read and write data on the console device.

When you run a program from the command-line window (or shell), you have the option of *redirecting* the standard input and/or standard output devices. A command-line parameter of the form `>outfile` redirects the standard output device to the specified file (outfile). A command-line parameter of the form `<infile` redirects the standard input so that its data comes from the specified input file (infile). The following examples demonstrate how to use these parameters when running a program named *testpgm* in the command window:<sup>14</sup>

---

```
testpgm <input.data
testpgm >output.txt
testpgm <in.txt >output.txt
```

---

<sup>13</sup>Namespaces are the subject of Chapter 5.

<sup>14</sup>For Linux, FreeBSD, and Mac OS X users, depending on how your system is set up, you may need to type `./` in front of the program's name to actually execute the program (e.g., `./testpgm <input.data`).

### 1.10.3 The *stdout.newln* Routine

The `stdout.newln` procedure prints a newline sequence to the standard output device. This is functionally equivalent to saying `stdout.put( nl );`. The call to `stdout.newln` is sometimes a little more convenient. For example:

---

```
stdout.newln();
```

---

### 1.10.4 The *stdout.putiX* Routines

The `stdout.puti8`, `stdout.puti16`, and `stdout.puti32` library routines print a single parameter (one byte, two bytes, or four bytes, respectively) as a signed integer value. The parameter may be a constant, a register, or a memory variable, as long as the size of the actual parameter is the same as the size of the formal parameter.

These routines print the value of their specified parameter to the standard output device. These routines will print the value using the minimum number of print positions possible. If the number is negative, these routines will print a leading minus sign. Here are some examples of calls to these routines:

---

```
stdout.puti8( 123 );  
stdout.puti16( dx );  
stdout.puti32( i32Var );
```

---

### 1.10.5 The *stdout.putiXSize* Routines

The `stdout.puti8Size`, `stdout.puti16Size`, and `stdout.puti32Size` routines output signed integer values to the standard output, just like the `stdout.putiX` routines. These routines, however, provide more control over the output; they let you specify the (minimum) number of print positions the value will require on output. These routines also let you specify a padding character should the print field be larger than the minimum needed to display the value. These routines require the following parameters:

---

```
stdout.puti8Size( Value8, width, padchar );  
stdout.puti16Size( Value16, width, padchar );  
stdout.puti32Size( Value32, width, padchar );
```

---

The *Value\** parameter can be a constant, a register, or a memory location of the specified size. The *width* parameter can be any signed integer constant that is between -256 and +256; this parameter may be a constant, register (32-bit), or memory location (32-bit). The *padchar* parameter should be a single-character value.

Like the `stdout.putiX` routines, these routines print the specified value as a signed integer constant to the standard output device. These routines, however, let you specify the *field width* for the value. The field width is the minimum number of print positions these routines will use when printing the value. The *width* parameter specifies the minimum field width. If the

number would require more print positions (e.g., if you attempt to print 1234 with a field width of 2), then these routines will print however many characters are necessary to properly display the value. On the other hand, if the *width* parameter is greater than the number of character positions required to display the value, then these routines will print some extra padding characters to ensure that the output has at least *width* character positions. If the *width* value is negative, the number is left justified in the print field; if the *width* value is positive, the number is right justified in the print field.

If the absolute value of the *width* parameter is greater than the minimum number of print positions, then these `stdout.putiXSize` routines will print a padding character before or after the number. The *padchar* parameter specifies which character these routines will print. Most of the time you would specify a space as the pad character; for special cases, you might specify some other character. Remember, the *padchar* parameter is a character value; in HLA character constants are surrounded by apostrophes, not quotation marks. You may also specify an 8-bit register as this parameter.

Listing 1-4 provides a short HLA program that demonstrates the use of the `stdout.puti32Size` routine to display a list of values in tabular form.

---

```
program NumsInColumns;

#include( "stdlib.hhf" )

var
    i32:    int32;
    ColCnt: int8;

begin NumsInColumns;

    mov( 96, i32 );
    mov( 0, ColCnt );
    while( i32 > 0 ) do

        if( ColCnt = 8 ) then

            stdout.newLn();
            mov( 0, ColCnt );

        endif;
        stdout.puti32Size( i32, 5, ' ' );
        sub( 1, i32 );
        add( 1, ColCnt );

    endwhile;
    stdout.newLn();

end NumsInColumns;
```

---

*Listing 1-4: Tabular output demonstration using `stdio.Puti32Size`*

### 1.10.6 The *stdout.put* Routine

The *stdout.put* routine<sup>15</sup> is the one of the most flexible output routines in the standard output library module. It combines most of the other output routines into a single, easy-to-use procedure.

The generic form for the *stdout.put* routine is the following:

---

```
stdout.put( list_of_values_to_output );
```

---

The *stdout.put* parameter list consists of one or more constants, registers, or memory variables, each separated by a comma. This routine displays the value associated with each parameter appearing in the list. Because we've already been using this routine throughout this chapter, you've already seen many examples of this routine's basic form. It is worth pointing out that this routine has several additional features not apparent in the examples appearing in this chapter. In particular, each parameter can take one of the following two forms:

---

```
value  
value:width
```

---

The *value* may be any legal constant, register, or memory variable object. In this chapter, you've seen string constants and memory variables appearing in the *stdout.put* parameter list. These parameters correspond to the first form above. The second parameter form above lets you specify a minimum field width, similar to the *stdout.putiXSize* routines.<sup>16</sup> The program in Listing 1-5 produces the same output as the program in Listing 1-4; however, Listing 1-5 uses *stdout.put* rather than *stdout.puti32Size*.

---

```
program NumsInColumns2;  
  
#include( "stdlib.hhf" )  
  
var  
    i32:    int32;  
    ColCnt: int8;  
  
begin NumsInColumns2;  
  
    mov( 96, i32 );  
    mov( 0, ColCnt );  
    while( i32 > 0 ) do  
  
        if( ColCnt = 8 ) then
```

---

<sup>15</sup> *stdout.put* is actually a macro, not a procedure. The distinction between the two is beyond the scope of this chapter. Chapter 9 describes their differences.

<sup>16</sup> Note that you cannot specify a padding character when using the *stdout.put* routine; the padding character defaults to the space character. If you need to use a different padding character, call the *stdout.putiXSize* routines.

```

        stdout.newln();
        mov( 0, ColCnt );

    endif;
    stdout.put( i32:5 );
    sub( 1, i32 );
    add( 1, ColCnt );

endwhile;
stdout.put( nl );

end NumsInColumns2;

```

---

*Listing 1-5: Demonstration of the stdout.put field width specification*

The stdout.put routine is capable of much more than the few attributes this section describes. This text will introduce those additional capabilities as appropriate.

### **1.10.7 The stdin.getc Routine**

The stdin.getc routine reads the next available character from the standard input device's input buffer.<sup>17</sup> It returns this character in the CPU's AL register. The program in Listing 1-6 demonstrates a simple use of this routine.

---

```

program charInput;

#include( "stdlib.hhf" )

var
    counter: int32;

begin charInput;

    // The following repeats as long as the user
    // confirms the repetition.

    repeat

        // Print out 14 values.

        mov( 14, counter );
        while( counter > 0 ) do

            stdout.put( counter:3 );
            sub( 1, counter );

        endwhile;

        // Wait until the user enters 'y' or 'n'.
    
```

---

<sup>17</sup> *Buffer* is just a fancy term for an array.

```

        stdout.put( nl, nl, "Do you wish to see it again? (y/n):" );
        forever

            stdin.readLn();
            stdin.getc();
            breakif( al = 'n' );
            breakif( al = 'y' );
            stdout.put( "Error, please enter only 'y' or 'n': " );

        endfor;
        stdout.newLn();

    until( al = 'n' );

end charInput;

```

---

*Listing 1-6: Demonstration of the stdin.getc() routine*

This program uses the `stdin.ReadLn` routine to force a new line of input from the user. A description of `stdin.ReadLn` appears in Section 1.10.9.

### **1.10.8 The stdin.getiX Routines**

The `stdin.geti8`, `stdin.geti16`, and `stdin.geti32` routines read 8-, 16-, and 32-bit signed integer values from the standard input device. These routines return their values in the AL, AX, or EAX register, respectively. They provide the standard mechanism for reading signed integer values from the user in HLA.

Like the `stdin.getc` routine, these routines read a sequence of characters from the standard input buffer. They begin by skipping over any whitespace characters (spaces, tabs, and so on) and then convert the following stream of decimal digits (with an optional leading minus sign) into the corresponding integer. These routines raise an exception (that you can trap with the `try..endtry` statement) if the input sequence is not a valid integer string or if the user input is too large to fit in the specified integer size. Note that values read by `stdin.geti8` must be in the range  $-128..+127$ ; values read by `stdin.geti16` must be in the range  $-32,768..+32,767$ ; and values read by `stdin.geti32` must be in the range  $-2,147,483,648..+2,147,483,647$ .

The sample program in Listing 1-7 demonstrates the use of these routines.

---

```

program intInput;

#include( "stdlib.hhf" )

var
    i8:      int8;
    i16:     int16;
    i32:     int32;

begin intInput;

```

```

// Read integers of varying sizes from the user:

stdout.put( "Enter a small integer between -128 and +127: " );
stdin.geti8();
mov( al, i8 );

stdout.put( "Enter a small integer between -32768 and +32767: " );
stdin.geti16();
mov( ax, i16 );

stdout.put( "Enter an integer between +/- 2 billion: " );
stdin.geti32();
mov( eax, i32 );

// Display the input values.

stdout.put
(
    nl,
    "Here are the numbers you entered:", nl, nl,
    "Eight-bit integer: ", i8:12, nl,
    "16-bit integer:     ", i16:12, nl,
    "32-bit integer:     ", i32:12, nl
);

end intInput;

```

---

*Listing 1-7: stdin.getiX example code*

You should compile and run this program and then test what happens when you enter a value that is out of range or enter an illegal string of characters.

### ***1.10.9 The stdin.readLn and stdin.flushInput Routines***

Whenever you call an input routine like `stdin.getc` or `stdin.geti32`, the program does not necessarily read the value from the user at that moment. Instead, the HLA Standard Library buffers the input by reading a whole line of text from the user. Calls to input routines will fetch data from this input buffer until the buffer is empty. While this buffering scheme is efficient and convenient, sometimes it can be confusing. Consider the following code sequence:

---

```

stdout.put( "Enter a small integer between -128 and +127: " );
stdin.geti8();
mov( al, i8 );

stdout.put( "Enter a small integer between -32768 and +32767: " );
stdin.geti16();
mov( ax, i16 );

```

---



Intuitively, you would expect the program to print the first prompt message, wait for user input, print the second prompt message, and wait for the second user input. However, this isn't exactly what happens. For example, if you run this code (from the sample program in the previous section) and enter the text **123 456** in response to the first prompt, the program will not stop for additional user input at the second prompt. Instead, it will read the second integer (456) from the input buffer read during the execution of the `stdin.geti16` call.

In general, the `stdin` routines read text from the user only when the input buffer is empty. As long as the input buffer contains additional characters, the input routines will attempt to read their data from the buffer. You can take advantage of this behavior by writing code sequences such as the following:

---

```
stdout.put( "Enter two integer values: " );
stdin.geti32();
mov( eax, intval );
stdin.geti32();
mov( eax, AnotherIntVal );
```

---

This sequence allows the user to enter both values on the same line (separated by one or more whitespace characters), thus preserving space on the screen. So the input buffer behavior is desirable every now and then. The buffered behavior of the input routines can be counterintuitive at other times.

Fortunately, the HLA Standard Library provides two routines, `stdin.readLine` and `stdin.flushInput`, that let you control the standard input buffer. The `stdin.readLine` routine discards everything that is in the input buffer and immediately requires the user to enter a new line of text. The `stdin.flushInput` routine simply discards everything that is in the buffer. The next time an input routine executes, the system will require a new line of input from the user. You would typically call `stdin.readLine` immediately before some standard input routine; you would normally call `stdin.flushInput` immediately after a call to a standard input routine.

**NOTE** *If you are calling `stdin.readLine` and you find that you are having to input your data twice, this is a good indication that you should be calling `stdin.flushInput` rather than `stdin.readLine`. In general, you should always be able to call `stdin.flushInput` to flush the input buffer and read a new line of data on the next input call. The `stdin.readLine` routine is rarely necessary, so you should use `stdin.flushInput` unless you really need to immediately force the input of a new line of text.*

### **1.10.10 The `stdin.get` Routine**

The `stdin.get` routine combines many of the standard input routines into a single call, just as the `stdout.put` combines all of the output routines into a single call. Actually, `stdin.get` is a bit easier to use than `stdout.put` because the only parameters to this routine are a list of variable names.

Let's rewrite the example given in the previous section:

---

```
stdout.put( "Enter two integer values: " );
stdin.geti32();
mov( eax, intval );
stdin.geti32();
mov( eax, AnotherIntVal );
```

---

Using the `stdin.get` routine, we could rewrite this code as:

---

```
stdout.put( "Enter two integer values: " );
stdin.get( intval, AnotherIntVal );
```

---

As you can see, the `stdin.get` routine is a little more convenient to use.

Note that `stdin.get` stores the input values directly into the memory variables you specify in the parameter list; it does not return the values in a register unless you actually specify a register as a parameter. The `stdin.get` parameters must all be variables or registers.

## 1.11 Additional Details About `try..endtry`

As you may recall, the `try..endtry` statement surrounds a block of statements in order to capture any exceptions that occur during the execution of those statements. The system raises exceptions in one of three ways: through a hardware fault (such as a divide-by-zero error), through an operating system-generated exception, or through the execution of the HLA `raise` statement. You can write an exception handler to intercept specific exceptions using the exception clause. The program in Listing 1-8 provides a typical example of the use of this statement.

---

```
program testBadInput;
#include( "stdlib.hhf" )

static
    u:      int32;

begin testBadInput;

    try

        stdout.put( "Enter a signed integer:" );
        stdin.get( u );
        stdout.put( "You entered: ", u, nl );

        exception( ex.ConversionError )

            stdout.put( "Your input contained illegal characters" nl );

        exception( ex.ValueOutOfRange )
```

```

        stdout.put( "The value was too large" nl );

    endtry;

end testBadInput;

```

---

*Listing 1-8: try..endtry example*

HLA refers to the statements between the try clause and the first exception clause as the *protected* statements. If an exception occurs within the protected statements, then the program will scan through each of the exceptions and compare the value of the current exception against the value in the parentheses after each of the exception clauses.<sup>18</sup> This exception value is simply a 32-bit value. The value in the parentheses after each exception clause, therefore, must be a 32-bit value. The HLA *excepts.hhf* header file predefines several exception constants. Although it would be an incredibly bad style violation, you could substitute the numeric values for the two exception clauses above.

### **1.11.1 Nesting try..endtry Statements**

If the program scans through all the exception clauses in a try..endtry statement and does not match the current exception value, then the program searches through the exception clauses of a *dynamically nested* try..endtry block in an attempt to find an appropriate exception handler. For example, consider the code in Listing 1-9.

---

```

program testBadInput2;
#include( "stdlib.hhf" )

static
    u:      int32;

begin testBadInput2;

    try

        try

            stdout.put( "Enter a signed integer: " );
            stdin.get( u );
            stdout.put( "You entered: ", u, nl );

            exception( ex.ConversionError )

                stdout.put( "Your input contained illegal characters" nl );

        endtry;

    endtry;

```

---

<sup>18</sup> Note that HLA loads this value into the EAX register. So upon entry into an exception clause, EAX contains the exception number.

```

        stdout.put( "Input did not fail due to a value out of range" nl );

        exception( ex.ValueOutOfRangeException )

        stdout.put( "The value was too large" nl );

    endtry;

end testBadInput2;

```

---

*Listing 1-9: Nested try..endtry statements*

In Listing 1-9 one try statement is nested inside another. During the execution of the `stdin.get` statement, if the user enters a value greater than four billion and some change, then `stdin.get` will raise the `ex.ValueOutOfRangeException` exception. When the HLA runtime system receives this exception, it first searches through all the exception clauses in the `try..endtry` statement immediately surrounding the statement that raised the exception (this would be the nested `try..endtry` in the example above). If the HLA runtime system fails to locate an exception handler for `ex.ValueOutOfRangeException`, then it checks to see if the current `try..endtry` is nested inside another `try..endtry` (as is the case in Listing 1-9). If so, the HLA runtime system searches for the appropriate exception clause in the outer `try..endtry` statement. Within the `try..endtry` block appearing in Listing 1-9 the program finds an appropriate exception handler, so control transfers to the statements after the `exception( ex.ValueOutOfRangeException )` clause.

After leaving a `try..endtry` block, the HLA runtime system no longer considers that block active and will not search through its list of exceptions when the program raises an exception.<sup>19</sup> This allows you to handle the same exception differently in other parts of the program.

If two `try..endtry` statements handle the same exception, and one of the `try..endtry` blocks is nested inside the protected section of the other `try..endtry` statement, and the program raises an exception while executing in the innermost `try..endtry` sequence, then HLA transfers control directly to the exception handler provided by the innermost `try..endtry` block. HLA does not automatically transfer control to the exception handler provided by the outer `try..endtry` sequence.

In the previous example (Listing 1-9) the second `try..endtry` statement was statically nested inside the enclosing `try..endtry` statement.<sup>20</sup> As mentioned without comment earlier, if the most recently activated `try..endtry` statement does not handle a specific exception, the program will search through the exception clauses of any dynamically nesting `try..endtry` blocks. Dynamic nesting does not require the nested `try..endtry` block to physically appear within the enclosing `try..endtry` statement. Instead, control could transfer

<sup>19</sup> Unless, of course, the program re-enters the `try..endtry` block via a loop or other control structure.

<sup>20</sup> *Statically nested* means that one statement is physically nested within another in the source code. When we say one statement is nested within another, this typically means that the statement is statically nested within the other statement.

from inside the enclosing `try..endtry` protected block to some other point in the program. Execution of a `try..endtry` statement at that other point dynamically nests the two try statements. Although there are many ways to dynamically nest code, there is one method you are probably familiar with from your high-level language experience: the procedure call. In Chapter 5, when you learn how to write procedures (functions) in assembly language, you should keep in mind that any call to a procedure within the protected section of a `try..endtry` block can create a dynamically nested `try..endtry` if the program executes a `try..endtry` within that procedure.

### ***1.11.2 The unprotected Clause in a try..endtry Statement***

Whenever a program executes the try clause, it preserves the current exception environment and sets up the system to transfer control to the exception clauses within that `try..endtry` statement should an exception occur. If the program successfully completes the execution of a `try..endtry` protected block, the program restores the original exception environment and control transfers to the first statement beyond the `endtry` clause. This last step, restoring the execution environment, is very important. If the program skips this step, any future exceptions will transfer control to this `try..endtry` statement even though the program has already left the `try..endtry` block. Listing 1-10 demonstrates this problem.

---

```
program testBadInput3;
#include( "stdlib.hhf" )

static
    input: int32;

begin testBadInput3;

    // This forever loop repeats until the user enters
    // a good integer and the break statement below
    // exits the loop.

    forever

        try

            stdout.put( "Enter an integer value: " );
            stdin.get( input );
            stdout.put( "The first input value was: ", input, nl );
            break;

        exception( ex.ValueOutOfRange )

            stdout.put( "The value was too large, re-enter." nl );

        exception( ex.ConversionError )

            stdout.put( "The input contained illegal characters, re-enter." nl );
```

```

        endtry;

    endfor;

    // Note that the following code is outside the loop and there
    // is no try..endtry statement protecting this code.

    stdout.put( "Enter another number: " );
    stdin.get( input );
    stdout.put( "The new number is: ", input, nl );

end testBadInput3;

```

---

*Listing 1-10: Improperly exiting a try..endtry statement*

This example attempts to create a robust input system by putting a loop around the try..endtry statement and forcing the user to reenter the data if the stdin.get routine raises an exception (because of bad input data). While this is a good idea, there is a big problem with this implementation: the break statement immediately exits the forever..endfor loop without first restoring the exception environment. Therefore, when the program executes the second stdin.get statement, at the bottom of the program, the HLA exception-handling code still thinks that it's inside the try..endtry block. If an exception occurs, HLA transfers control back into the try..endtry statement looking for an appropriate exception handler. Assuming the exception was ex.ValueOutOfRange or ex.ConversionError, the program in Listing 1-10 will print an appropriate error message *and then force the user to re-enter the first value*. This isn't desirable.

Transferring control to the wrong try..endtry exception handlers is only part of the problem. Another big problem with the code in Listing 1-10 has to do with the way HLA preserves and restores the exception environment: specifically, HLA saves the old execution environment information in a special region of memory known as the *stack*. If you exit a try..endtry without restoring the exception environment, this leaves the old execution environment information on the stack, and this extra data on could cause your program to malfunction.

Although this discussion makes it quite clear that a program should not exit from a try..endtry statement in the manner that Listing 1-10 uses, it would be nice if you could use a loop around a try..endtry block to force the reentry of bad data as this program attempts to do. To allow for this, HLA's try..endtry statement provides an unprotected section. Consider the code in Listing 1-11.

---

```

program testBadInput4;
#include( "stdlib.hhf" )

static
    input: int32;

begin testBadInput4;

```

```

// This forever loop repeats until the user enters
// a good integer and the break statement below
// exits the loop. Note that the break statement
// appears in an unprotected section of the try..endtry
// statement.

forever

    try

        stdout.put( "Enter an integer value: " );
        stdin.get( input );
        stdout.put( "The first input value was: ", input, nl );

    unprotected

        break;

    exception( ex.ValueOutOfRangeException )

        stdout.put( "The value was too large, re-enter." nl );

    exception( ex.ConversionError )

        stdout.put( "The input contained illegal characters, re-enter." nl );

    endtry;

endfor;

// Note that the following code is outside the loop and there
// is no try..endtry statement protecting this code.

stdout.put( "Enter another number: " );
stdin.get( input );
stdout.put( "The new number is: ", input, nl );

end testBadInput4;

```

---

*Listing 1-11: The try..endtry unprotected section*

Whenever the try..endtry statement hits the unprotected clause, it immediately restores the exception environment. As the phrase suggests, the execution of statements in the unprotected section is no longer protected by that try..endtry block (note, however, that any dynamically nesting try..endtry statements will still be active; unprotected turns off only the exception handling of the try..endtry statement containing the unprotected clause). Because the break statement in Listing 1-11 appears inside the unprotected section, it can safely transfer control out of the try..endtry block without “executing” the endtry because the program has already restored the former exception environment.

Note that the unprotected keyword must appear in the `try..endtry` statement immediately after the protected block. That is, it must precede all exception keywords.

If an exception occurs during the execution of a `try..endtry` sequence, HLA automatically restores the execution environment. Therefore, you may execute a `break` statement (or any other instruction that transfers control out of the `try..endtry` block) within an exception clause.

Because the program restores the exception environment upon encountering an unprotected block or an exception block, an exception that occurs within one of these areas immediately transfers control to the previous (dynamically nesting) active `try..endtry` sequence. If there is no nesting `try..endtry` sequence, the program aborts with an appropriate error message.

### **1.11.3 The *anyexception* Clause in a *try..endtry* Statement**

In a typical situation, you will use a `try..endtry` statement with a set of exception clauses that will handle all possible exceptions that can occur in the protected section of the `try..endtry` sequence. Often, it is important to ensure that a `try..endtry` statement handles all possible exceptions to prevent the program from prematurely aborting due to an unhandled exception. If you have written all the code in the protected section, you will know the exceptions it can raise, so you can handle all possible exceptions. However, if you are calling a library routine (especially a third-party library routine), making a OS API call, or otherwise executing code that you have no control over, it may not be possible for you to anticipate all possible exceptions this code could raise (especially when considering past, present, and future versions of the code). If that code raises an exception for which you do not have an exception clause, this could cause your program to fail. Fortunately, HLA's `try..endtry` statement provides the *anyexception* clause that will automatically trap any exception the existing exception clauses do not handle.

The *anyexception* clause is similar to the exception clause except it does not require an exception number parameter (because it handles any exception). If the *anyexception* clause appears in a `try..endtry` statement with other exception sections, the *anyexception* section must be the last exception handler in the `try..endtry` statement. An *anyexception* section may be the only exception handler in a `try..endtry` statement.

If an otherwise unhandled exception transfers control to an *anyexception* section, the EAX register will contain the exception number. Your code in the *anyexception* block can test this value to determine the cause of the exception.

### **1.11.4 Registers and the *try..endtry* Statement**

The `try..endtry` statement preserves several bytes of data whenever you enter a `try..endtry` statement. Upon leaving the `try..endtry` block (or hitting the unprotected clause), the program restores the exception environment. As long as no exception occurs, the `try..endtry` statement does not affect the



values of any registers upon entry to or upon exit from the `try..endtry` statement. However, this claim is not true if an exception occurs during the execution of the protected statements.

Upon entry into an exception clause, the `EAX` register contains the exception number, but the values of all other general-purpose registers are undefined. Because the operating system may have raised the exception in response to a hardware error (and, therefore, has played around with the registers), you can't even assume that the general-purpose registers contain whatever values they happened to contain at the point of the exception. The underlying code that HLA generates for exceptions is subject to change in different versions of the compiler, and certainly it changes across operating systems, so it is never a good idea to experimentally determine what values registers contain in an exception handler and depend on those values in your code.

Because entry into an exception handler can scramble the register values, you must ensure that you reload important registers if the code following your `endtry` clause assumes that the registers contain certain values (i.e., values set in the protected section or values set prior to executing the `try..endtry` statement). Failure to do so will introduce some nasty defects into your program (and these defects may be very intermittent and difficult to detect because exceptions rarely occur and may not always destroy the value in a particular register). The following code fragment provides a typical example of this problem and its solution:

---

```
static
    sum: int32;
        .
        .
        .
    mov( 0, sum );
    for( mov( 0, ebx ); ebx < 8; inc( ebx )) do

        push( ebx ); // Must preserve ebx in case there is an exception.
        forever
            try

                stdin.geti32();
                unprotected break;

            exception( ex.ConversionError )

                stdout.put( "Illegal input, please re-enter value: " );

            endtry;
        endfor;
        pop( ebx ); // Restore ebx's value.
        add( ebx, eax );
        add( eax, sum );

    endfor;
```

---

Because the HLA exception-handling mechanism messes with the registers, and because exception handling is a relatively inefficient process, you should never use the `try..endtry` statement as a generic control structure (e.g., using it to simulate a `switch/case` statement by raising an integer exception value and using the exception clauses as the cases to process). Doing so will have a very negative impact on the performance of your program and may introduce subtle defects because exceptions scramble the registers.

For proper operation, the `try..endtry` statement assumes that you use the EBP register only to point at *activation records* (Chapter 5 discusses activation records). By default, HLA programs automatically use EBP for this purpose; as long as you do not modify the value in EBP, your programs will automatically use EBP to maintain a pointer to the current activation record. If you attempt to use the EBP register as a general-purpose register to hold values and compute arithmetic results, HLA's exception-handling capabilities will no longer function properly (along with other possible problems). Therefore, you should never use the EBP register as a general-purpose register. Of course, this same discussion applies to the ESP register.

## 1.12 High-Level Assembly Language vs. Low-Level Assembly Language

Before concluding this chapter, it's important to remind you that none of the control statements appearing in this chapter are "real" assembly language. The 80x86 CPU does not support machine instructions like `if`, `while`, `repeat`, `for`, `break`, `breakif`, and `try`. Whenever HLA encounters these statements, it *compiles* them into a sequence of one or more true machine instructions that do the operation as the high-level statements you've used. While these statements are convenient to use, and in many cases just as efficient as the sequence of low-level machine instructions into which HLA translates them, don't lose sight of the fact that they are not true machine instructions.

The purpose of this text is to teach you low-level assembly language programming; these high-level control structures are simply a means to that end. Remember, learning the HLA high-level control structures allows you to leverage your high-level language knowledge early on in the educational process so you don't have to learn everything about assembly language all at once. By using high-level control structures that you're already comfortable with, this text can put off the discussion of the actual machine instructions you'd normally use for control flow until much later. By doing so, this text can regulate how much material it presents, so, hopefully, you'll find learning assembly language to be much more pleasant. However, you must always remember that these high-level control statements are just a pedagogical tool to help you learn assembly language. Though you're free to use them in your assembly programs once you master the real control-flow statements, you really must learn the low-level control statements if you want to learn assembly language programming. Since, presumably, that's why you're reading this

book, don't allow the high-level control structures to become a crutch. When you get to the point where you learn how to really write low-level control statements, embrace and use them (exclusively). As you gain experience with the low-level control statements and learn their advantages and disadvantages, you'll be in a good position to decide whether a high-level or low-level code sequence is most appropriate for a given application. However, until you gain considerable experience with the low-level control structures, you'll not be able to make an educated decision. Remember, you can't really call yourself an assembly language programmer unless you've mastered the low-level statements.

Another thing to keep in mind is that the HLA Standard Library functions are not part of the assembly *language*. They're just some convenient functions that have been prewritten for you. Although there is nothing wrong with calling these functions, always remember that they are not machine instructions and that there is nothing special about these routines; as you gain experience writing assembly language code, you can write your own versions of each of these routines (and even write them more efficiently).

If you're learning assembly language because you want to write the most efficient programs possible (either the fastest or the smallest code), you need to understand that you won't achieve this goal completely if you're using high-level control statements and making a lot of calls to the HLA Standard Library. HLA's code generator and the HLA Standard Library aren't *horribly* inefficient, but the only true way to write efficient programs in assembly language is to *think* in assembly language. HLA's high-level control statements and many of the routines in the HLA Standard Library are great because they let you *avoid* thinking in assembly language. While this is great while you're first learning assembly, if your ultimate goal is to write efficient code, then you have to learn to think in assembly language. This text will get you to that point (and will do so much more rapidly because it uses HLA's high-level features), but don't forget that your ultimate goal is to give up these high-level features in favor of low-level coding.

## 1.13 For More Information

This chapter has covered a lot of ground! While you still have a lot to learn about assembly language programming, this chapter, combined with your knowledge of high-level languages, provides just enough information to let you start writing real assembly language programs.

Although this chapter has covered many different topics, the three primary topics of interest are the 80x86 CPU architecture, the syntax for simple HLA programs, and the HLA Standard Library. For additional topics on this subject, please consult the (unabridged) electronic version of this text, the HLA reference manual, and the HLA Standard Library manual. All three are available at <http://www.artofasm.com/> and <http://webster.cs.ucr.edu/>.