
Hello, World of Assembly Language

Chapter Two

2.1 Chapter Overview

This chapter is a “quick-start” chapter that lets you start writing basic assembly language programs right away. This chapter presents the basic syntax of an HLA (High Level Assembly) program, introduces you to the Intel CPU architecture, provides a handful of data declarations and machine instructions, describes some utility routines you can call in the HLA Standard Library, and then 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 be prepared to start learning new language features in subsequent chapters.

2.2 Installing the HLA Distribution Package

Before you can learn assembly language programming using HLA, you must first successfully install HLA on your system. Currently, HLA is available for the Linux and Windows operating systems. This section explains how to install HLA on these two systems. If HLA is already running on your system, you may skip to the next major section in this chapter.

The latest version of HLA is available from the Webster web server at

<http://webster.cs.ucr.edu>

Go to this web site and following the HLA links to the “HLA Download” page. From here you should select the latest version of HLA for download to your computer. The HLA distribution is provided in a “Zip File” compressed format. Under Windows, you will need a decompressor program like PKUNZIP or WinZip in order to extract the HLA files from this zipped archive file; under Linux, you will use the GZIP and TAR programs to decompress and extract HLA. A detailed description of the use of these decompression products is beyond the scope of this manual, please consult the software vendor’s documentation or their web page for information concerning the use of these products; this discussion will only briefly describe how to use them to extract important HLA files.

This text assumes that you will unzip the HLA distribution into the root directory of your C: drive under Windows, or to the “/usr/hla” directory under Linux. You can certainly install HLA anywhere you want, but you will have to adjust the following descriptions if you install HLA somewhere else. If possible, you should install HLA using root/administrator privileges; regardless, you should make sure the permissions are set properly on the files so everyone has read and execute access to the HLA files; if you are unsure how to do this, please consult your operating system’s documentation or consult a system administrator.

HLA is a console application. In order to run the HLA compiler you must run the command window program (this is “command.com” on Windows 95 and 98, or “cmd.exe” on Windows NT and Windows 2000; Linux users typically run “bash” or some other shell). This also means that you should be familiar with some simple “command line interface” (CLI) or “shell” commands.

Most Windows distributions let you run the command prompt windows from the Start menu or from a submenu hanging off the start menu (you may also select “RUN” from the Start menu and type “cmd” as the program name). This text assumes that you are familiar with the Windows command window and you know how to use some basic command window commands (e.g., dir, del, rename, etc.). If you have never before used the Windows command line interpreter, you should consult an appropriate text to learn a few basic commands.

Most Linux distributions run “bash” or some other shell program whenever you open up a terminal window (e.g., a GNOME or KDE terminal window or an X-TERM window). There are some minor differences between the shells running under Linux, this document assumes that you are using GNU’s “bash” shell. Again, this text assumes that you are comfortable with a few commands like ls, rm, and mv. If you have never used a Unix shell program before, you should consult an appropriate text or the on-line documentation to learn a few basic commands.

Before you can actually run the HLA compiler, you must set the system execution path and set up various environment variables. The following subsections explain how to do this under Windows and then Linux.

2.2.1 Installation Under Windows

HLA is not a stand alone program. It is a compiler that translates HLA source code into a lower-level assembly language. A separate assembler, such as MASM, then completes the processing of this low-level intermediate code to produce an object code file. Finally, you must link the object code output from the assembler using a linker program. Typically you will link the object code produced by one or more HLA source files with the HLA Standard Library (hlalib.lib) and, possibly, several operating system specific library files (e.g., kernel32.lib under Windows). Most of this activity takes place transparently whenever you ask HLA to compile your HLA source file(s). However, for the whole process to run smoothly, you must have installed HLA and all the support files correctly. This section will discuss how to set up HLA on your Windows system.

First, you will need an HLA distribution for Windows. The latest version of HLA is always available on Webster at <http://webster.cs.ucr.edu>. You should go there and download the latest version if you do not already possess it.

As noted earlier, HLA is not a stand alone assembler. The HLA package contains the HLA compiler, the HLA Standard Library, and a set of include files for the HLA Standard Library. If you write an HLA program with just this code, HLA will produce an "ASM" file and then stop. To produce an executable file you will need Microsoft's MASM and LINK programs, along with some Windows library files, to complete the process. The easiest way to get all the files you need is to download the "MASM32" package from <http://www.pdq.com.au/home/hutch/masm.htm> or any of the other places on the net where you can find the MASM32 package (Webster maintains a current link if this link is dead). Once you unzip this file, it's easy to install the MASM32 package using the install program it supplies. ***You must install MASM32 (or MASM/LINK/Win32 library files) before HLA will function properly.***

Here are the steps I went through to install MASM32 on my system:

- I downloaded masm32v6.zip from the URL above (later versions are probably okay too, although there is a slight chance that the installation will be different).
- I double-clicked on the masm32v6.zip file (which runs WinZip on my system).
- I choose to extract "install.exe". I told WinZip to extract this file to C:\.
- I double-clicked on the "install.exe" icon and selected the "C:" drive in the window that popped up. Then I hit the install button and waited while MASM32 extracted all the pertinent files. This produced a directory called "MASM32". MASM32 is a powerful assembly language development subsystem in its own right; but it uses the traditional MASM syntax rather than the HLA syntax. So we'll use MASM32 mainly for the assembler, linker, and library files. MASM32 also includes a simple editor/IDE and several other tools that may be useful to an HLA programmer. Feel free to check this software out and see if it is useful to you. For now, note that the executable files you will ultimately need are ML.EXE, ML.ERR, LINK.EXE, and a couple of DLLs. You can find them in the MASM32\BIN subdirectory. Leave them there for the time being. The MASM32\LIB directory also contains many Win32 library files you will need. Again, leave them alone for the time being.
- Next, if you haven't already done so, download the HLA executables file from Webster at <http://webster.cs.ucr.edu>. On Webster you can download several different ZIP files associated with HLA from the HLA download page. The "Executables" is the only one you'll absolutely need; however, you'll probably want to grab the documentation and examples files as well. If you're curious, or you want some more example code, you can download the source listings to the HLA Standard Library. If you're *really* curious (or masochistic), you can download the HLA compiler source listings to (this is *not* for casual browsing!).
- I downloaded the HLA1_32.zip file while writing this. Most likely, there is a much later version available as you're reading this. Be sure to get the latest version. I chose to download this file to my "C:\" root directory.

- After downloading HLA1_32.zip to my C: drive, I double-clicked on the icon to run WinZip. I selected "Extract" and told WinZip to extract all the files to my C:\ directory. This created an "HLA" subdirectory in my root on C: with two subdirectories (include and lib) and two EXE files (HLA.EXE and HLPARSE.EXE. The HLA program is a "shell" program that runs the HLA compiler (HLPARSE.EXE), MASM (ML.EXE), the linker (LINK.EXE), and other programs. You can think of HLA.EXE as the "HLA Compiler".
- Next, I created the following text file and named it "IHLA.BAT" (note that you may need to change the default drive letters if you want to install HLA on a drive other than "C:");

```
path=c:\hla;c:\masm32\bin;%path%
set lib=c:\masm32\lib;c:\hla\hlalib;%lib%
set include=c:\hla\include;c:\masm32\include;%include%
set hlainc=c:\hla\include
set hlalib=c:\hla\hlalib\hlalib.lib
```

- Be sure you've typed all the lines exactly as written or HLA will fail to run properly. You may use any reasonable TEXT editor (e.g., NOTEPAD.EXE) to create this file. Do not use a word processing program (since they generally don't save their data as a TEXT file). Be sure the file is named "IHLA.BAT" and not "IHLA.BAT.TXT" or some other variation.
- This batch file tells the system where to find all the files you will need when running HLA. Advanced Win32 users should note that you can set all these environment variables up inside the Windows system control panel in the "Advanced->Environment Variables" area. This is far more convenient (ultimately) than using this batch file (for reasons you'll soon see). However, you can mess up your system if you don't know what you're doing when playing with the system control panel, so only advanced users who've done this stuff before should attempt this.
- HLA is a Win32 Console Window program. To run HLA you must open up a console window. Under Windows 2000, Microsoft has hidden this away in Start->Programs->Accessories->Command Prompt. You might find it in another location. You can also start the command prompt processor by selecting Start->Run and entering "cmd".
- Once you've got the command prompt, ("C:>" or something similar), execute the IHLA.BAT file you've created by typing "IHLA" at the command line prompt. Hit the ENTER key to execute the command.
- At this point, HLA should be properly installed and ready to run. Try typing "hla -?" at the command line prompt and verify that you get the HLA help message. If not, go back and figure out what you've done wrong up to this point (it doesn't hurt to start over from the beginning if you're lost).
- Thus far, you've verified that HLA.EXE is operational. Now try the following command: "ML /?" This should run the Microsoft Macro Assembler (MASM) and display the help screen. You can ignore the information that appears; you will probably never need to know this stuff.
- Next, let's verify the correct operation of the linker. Type "link /?" and verify that the linker program runs. Again, you can ignore the help screen that appears. You don't need to know about this stuff.
- Now it's time to try your hand at writing an honest to goodness HLA program and verify that the whole system is working. Here's the canonical "Hello World" program written in HLA (we will revisit this program a little later in this chapter, don't worry about what it means just yet). Enter it into a text editor and save it using the filename "HW.HLA":

```
program HelloWorld;
#include( "stdlib.hhf" )
begin HelloWorld;

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

end HelloWorld;
```

- Make sure you're in the same directory containing the HW.HLA file and type the following command at the "C:>" prompt: "HLA -v HW". The "-v" option tells HLA to produce VERBOSE output during compilation. This is helpful for determining what went wrong if the system fails somewhere along the line. This command should produce the following output:

```
HLA (High Level Assembler)
Written by Randall Hyde and released to the public domain.
Version Version 1.32 build 4904 (prototype)

Files:
1: hw.hla

Compiling "hw.hla" to "hw.asm"

Assembling hw.asm via "ml /c /coff /Cp hw.asm"

Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997. All rights reserved.

Assembling: hw.asm
Linking via "link -subsystem:console /heap:0x1000000,0x1000000
/stack:0x1000000,0x1000000 /BASE:0x3000000 /machine:IX86 -entry:?HLAMain @hw.link
-out:hw.exe kernel32.lib user32.lib c:\hla\hlalib\hlalib.lib hw.obj"
Microsoft (R) Incremental Linker Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/section:.text,ER
/section:readonly,R
/section:.edata,R
/section:.data,RW
/section:.bss,RW
```

- If you get all of this output, you're in business. You can run the "HW" program using the following CLI (command line interpreter) command:

```
HW
```

- One thing to remember is that unless you set the environment variables permanently in the System control panel, you will have to run the IHLA.BAT file every time you open up a new command prompt window. Since this is a pain, here are some instructions I've taken from the Internet that describe how to set up the environment variables (*DO THIS AT YOUR OWN RISK!*)
- 1) Open System Properties (Winkey-Break is a convenient shortcut) and go to Advanced tab, then Environment Variables. Add "c:\hla" to the Path in SYSTEM VARIABLES, not in "User variables for <your win2k login name>". Click OK, but keep the Environment Variables window open, we're not done.
 - 2) Look at the contents of ihla.bat (ABOVE):
 - 3) In "User Variables for <your login name>", you must end up with each of these settings. For example, to create hlainc, you click the "New..." button, type "hlainc" as the name of the variable, and type "c:\hla\include" as the Variable value (all without quotes of course). If there is already a path set, and it already has some value, add this immediately to the end: ";c:\hla;%path%" and that will preserve your existing User and System paths as well as adding c:\hla.

For example, suppose you opened up your User Variables for <login name> and it already said "C:\Private

Files\PantiePix;c:\winnt\system32;c:\winnt;c:\winnt\System32\Wbem;d:\lcc\bin;D:\PROGRA~1\ULTRAE~1;D:\4NT300;C:\msoffice\Office;c:\hla",

you would click on Edit and type "C:\Private Files\PantiePix;c:\hla;%path%"

(Same advice for preserving existing lib and include settings)

- 4) Once you reboot the computer, you should be all set for "Hello world of assembly language"! (without having to run the IHLA.BAT file.)

Installing HLA is a complex and slightly involved process. Unfortunately, this is necessary because I don't have the rights to distribute MASM, LINK, and other Microsoft files. Fortunately, HUTCH has collected all of these files together so they are easy to download. If you are concerned about possible legal issues with the download, you may legally download MASM and LINK from Microsoft's site. A link on Webster (at the URL above) describes how to do this. At the time this was being written, work was progressing on HLA to produce TASM compatible output and plans were in the works to produce NASM and Gas versions as well. However, you will still have to obtain the Microsoft library files from some source if you intend to produce a Win32 application. Versions of HLA may appear for other Operating Systems as well. Check out Webster to see if any progress has been made in this direction.

The most common two problems people have running HLA involve the location of the Win32 library files and the choice of linker. During the linking phase, HLA (well, link.exe actually) requires the kernel32.lib, user32.lib, and gdi32.lib library files. These must be present in the pathname(s) specified by the LIB environment variable. If, during the linker phase, HLA complains about missing object modules, make sure that the LIB path specifies the directory containing these files. If you're a MS VC++ user, installation of VC++ should have set up the LIB path for you. If not, then locate these files (they are part of the MASM32 distribution) and copy them to the HLA\HLALIB directory (note that the ihla.bat file includes c:\hla\hlalib as part of the LIB path).

Another common problem with running HLA is the use of the wrong link.exe program. Microsoft has distributed several different versions of link.exe; in particular, there are 16-bit linkers and 32-bit linkers. You must use a 32-bit segmented linker with HLA. If you get complaints about "stack size exceeded" or other errors during the linker phase, this is a good indication that you're using a 16-bit version of the linker. Obtain and use a 32-bit version and things will work. Don't forget that the 32-bit linker must appear in the execution path (specified by the PATH environment variable) before the 16-bit linker.

2.2.2 Installation Under Linux

HLA is not a stand alone program. It is a compiler that translates HLA source code into a lower-level assembly language. A separate assembler, such as Gas (as), then completes the processing of this low-level intermediate code to produce an object code file. Finally, you must link the object code output from the assembler using a linker program. Typically you will link the object code produced by one or more HLA source files with the HLA Standard Library (hlalib.a). Most of this activity takes place transparently whenever you ask HLA to compile your HLA source file(s). However, for the whole process to run smoothly, you must have installed HLA and all the support files correctly. This section will discuss how to set up HLA on your system.

First, you will need an HLA distribution for Linux. The latest version of HLA is always available on Webster at <http://webster.cs.ucr.edu>. You should go there and download the latest version if you do not already possess it.

As noted earlier, HLA is not a stand alone assembler. The HLA package contains the HLA compiler, the HLA Standard Library, and a set of include files for the HLA Standard Library. If you write an HLA program with just this code, HLA will produce an "ASM" file and then stop. To produce an executable file

you will need GNU's `as` and `ld` programs (these come with any Linux distribution that supports compiling C/C++ programs). Note that HLA only works with `Gas` v2.10 or later. The `Gas` assembler is part of the `Binutils` package. If you don't have version 2.10 or later, download an appropriate `binutils` package from the internet. HLA will generate errors when it attempts to assemble its output via an invocation of the `as` (`Gas`) executable if you don't have `Gas` v2.10 or later installed in your system.

Here are the steps I went through to install HLA on my Linux system:

- First, if you haven't already done so, download the HLA executables file from Webster at <http://webster.cs.ucr.edu>. On Webster you can download several different ZIP files associated with HLA from the HLA download page. The "Linux Executables" is the only one you'll absolutely need; however, you'll probably want to grab the documentation and examples files as well. If you're curious, or you want some more example code, you can download the source listings to the HLA Standard Library. If you're *really* curious (or masochistic), you can download the HLA compiler source listings to (this is *not* for casual browsing!).
- I downloaded the `HLA1_39.tar.gz` file while writing this. Most likely, there is a much later version available as you're reading this. Be sure to get the latest version. I chose to download this file to my root directory; you can put the file wherever you like, though this documentation assumes that all HLA files wind up in the `/usr/hla/...` directory tree. If you do not already have a `/usr/hla` subdirectory, you can create one with the `"mkdir"` command (it's best to do this using the `"root"` or `"superuser"` account; if you do not have superuser privileges, you should have your system administrator do this for you).
- After downloading `HLA1_39.tar.gz` to my root directory, I executed the following shell command: `"gzip -d HLA1_39.tar.gz"`. Once decompression was complete, I extracted the individual files using the command `"tar xvf HLA1_39.tar"`. This extracted a couple of executable files (`"hla"` and `"hlaparse"`) along with two subdirectories (`include` and `hlalib`). The HLA program is a "shell" program that runs the HLA compiler (`hlaparse`), `Gas` (`as`), the linker (`ld`), and other programs. You can think of `"hla"` as the "HLA Compiler". It would be a real good idea, at this point, to set the permissions on `"hla"` and `"hlaparse"` so that everyone can read and execute them. You should also set read and execute permissions on the two subdirectories and read permissions on all the files within the directories (if this isn't the default state). Do a `"man chmod"` from the Linux command-line if you don't know how to change permissions.
- Next, (logged in as a plain user rather than root or the super-user), I edited the `".bashrc"` file in my home directory (`/home/rhyde` in my particular case, this will probably be different for you). I found the line that defined the `"path"` variable, it originally looked like this on my system

```
"PATH=$DBROOT/bin:$DBROOT/pgm:$PATH"
```

I edited this line to add the path to the HLA directory, producing the following:

```
"PATH=$DBROOT/bin:$DBROOT/pgm:/usr/hla:$PATH"
```

Without this modification, Linux will probably not find HLA when you attempt to execute it unless you type a full path (e.g., `/usr/hla/hla`) when running the program. Since this is a pain, you'll definitely want to add `/usr/hla` to your path.

- Next, I added the following four lines to `".bashrc"` (note that Linux filenames beginning with a period don't normally show up in directory listings unless you supply the `"-a"` option to `ls`):


```
hlalib=/usr/hla/hlalib/hlalib.a
export hlalib
hlainc=/usr/hla/include
export hlainc
```

These four lines define (and export) environment variables that HLA needs during compilation. Without these environment variables, HLA will probably complain about not being able to find include files, or the linker (`ld`) will complain about strange undefined symbols when you attempt to compile your programs.

After saving the `".bashrc"` shell, you can tell Linux to make the changes to the system by using the command:

```
source .bashrc
```

Note: this discussion only applies to users who run the BASH shell. If you are using a different shell (like the C-Shell or the Korn Shell), then the directions for setting the path and environment variables differs slightly. Please see the documentation for your particular shell if you don't know how to do this. Also note that Linux does not normally display files whose name begins with a period when you use the "ls" command; to see such files, use the "ls -a" shell command.

- At this point, HLA should be properly installed and ready to run. Try typing "hla -?" at the command line prompt and verify that you get the HLA help message. If not, go back and figure out what you've done wrong up to this point (it doesn't hurt to start over from the beginning if you're lost).
- Now it's time to try your hand at writing an honest to goodness HLA program and verify that the whole system is working. Here's the canonical "Hello World" program written in HLA (we'll discuss this program in detail a little later in this chapter). Enter it into a text editor and save it using the filename "hw.hla":

```
program HelloWorld;
#include( "stdlib.hhf" )
begin HelloWorld;

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

end HelloWorld;
```

- Make sure you're in the same directory containing the "hw.hla" file and type the following command at the prompt: "hla -v hw". The "-v" option tells HLA to produce VERBOSE output during compilation. This is helpful for determining what went wrong if the system fails somewhere along the line. This command should produce the following output:

```
HLA (High Level Assembler) Parser
Written by Randall Hyde and released to the public domain.
Version Version 1.39 build 6845 (prototype)
-t active
File: t.hla

Compiling "t.hla" to "t.asm"
HLA (High Level Assembler)
Copyright 1999, by Randall Hyde, all rights reserved.
Version Version 1.39 build 6845 (prototype)
ELF output
Using GAS assembler
GAS output
-test active

Files:
1: t.hla

Compiling 't.hla' to 't.asm'
using command line [hlaparse -v -sg -test "t.hla"]

Assembling "t.asm" via [as -o t.o "t.asm"]
Linking via [ld -o "t" "t.o" "/usr/hla/hlalib/hlalib.a"]
```

Installing HLA is a complex and slightly involved process; though take heart, it's a lot simpler to install HLA under Linux than Windows! (See the previous section if you need proof.) Versions of HLA may appear for other operating systems (beyond Windows and Linux) as well. Check out Webster to see if any progress has been made in this direction. Note a very unique thing about HLA: Carefully written (console) applications will compile and run on all supported operating systems without change. This is unheard of for

assembly language! So if you are using multiple operating systems supported by HLA, you'll probably want to download files for all supported OSes.

Note: to run the HelloWorld program, a Linux user would type "hw" (or possibly "./hw") at the command line prompt.

2.2.3 Installing "Art of Assembly" Related Files

Although HLA is relatively flexible about where you put it on your system, this text assumes you've installed HLA in the "hla" directory on your C: drive under a Win32 operating system or in "/usr/hla" under Linux. This text also assumes the standard directory placement for the HLA files, which has the following layout

- HLA directory
- AoA directory
- Doc directory
- Examples directory
- hlalib directory
- hlabsrc directory
- include directory
- Tests directory

The "Art of Assembly" (AoA) software distribution has the following directory tree structure:

- AoA directory
- volume1
- ch01 directory
- ch02 directory
- etc.
- volume2
- ch01 directory
- ch02 directory
- etc.
- etc.

The main *HLA* directory contains the executable code for the compiler. This consists of two files, *HLA.EXE/hla* and *HLAPARSE.EXE/hlaparse* (Windows/Linux). These two programs must be in the current execution path in order to run the compiler. Under Windows, it wouldn't hurt to put the *ml.exe*, *ml.err*, *link.exe*, *mspdbX0.dll* (*x*=5, 6, or greater), and *msvcrt.dll* files in this directory as well. Under Linux, the "as" and "ld" programs are already in the execution path, assuming your Linux system supports C/C++ development.

The *Doc* directory contains reference material for HLA in PDF and HTML formats. If you have a copy of Adobe Acrobat Reader, you will probably want to read the PDF versions since they are much nicer than the HTML versions. These documents contain the most up-to-date information about the HLA language; you should consult them if you have a question about the HLA language or the HLA Standard Library. Generally, material in this documentation supersedes information appearing in this text since the HLA document is electronic and is probably more up to date.

The *Examples* directory contains a large set of HLA programs that demonstrate various features in the HLA language. If you have a question about an HLA feature, you can probably find an example program that demonstrates that feature in the *Examples* directory. Such examples provide invaluable insight that is often superior to a written description of the feature. Note that some of these programs may be specific to Windows or Linux, not all will compile and run under either operating system.

The *hlalib* directory contains the object code for the HLA Standard Library. As you become more competent with HLA, you may want to take a look at how HLA implements various library functions by checking out the library source code in the *hlalibsrc* subdirectory.

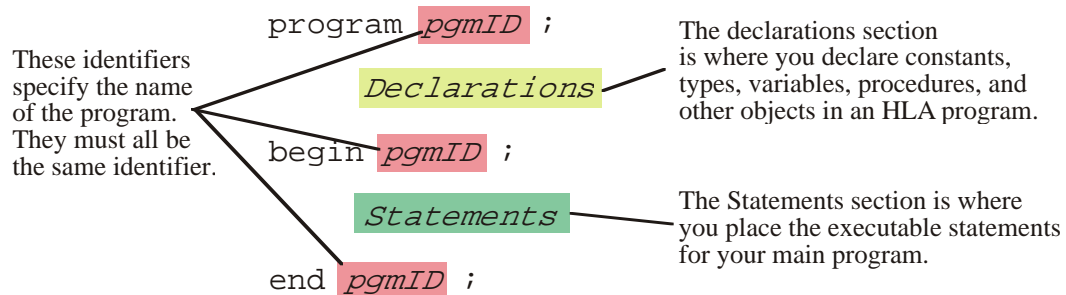
The *include* directory contains the HLA Standard Library include files. These special files (that end with a “.hhf” suffix, for “HLA Header File”) are needed during assembly to provide prototype and other information to your program. The example programs in this chapter all include the HLA header file “stdlib.hhf” that, in turn, includes all the other HLA header files in the standard library.

The *Tests* directory contains various test files that test the correct operation of the HLA system. HLA includes these files as part of the distribution package because they provide additional examples of HLA coding.

The *AoA* directory contains the code specific to this textbook. This directory contains all the source code to the (complete) programs appearing in this text. It also contains the programs appearing in the Laboratory Exercises section of each chapter. Therefore, this directory is very important to you. Within this subdirectory, the information is further divided up by volume and chapter. The material for Chapter One appears in the “ch01” subdirectory of the “volume1” directory in the AoA directory tree, the material for Chapter Two appears in the “ch02” subdirectory of the “volume1” directory, etc..

2.3 The Anatomy of an HLA Program

An HLA program typically takes the following form:



PROGRAM, BEGIN, and END are HLA reserved words that delineate the program. Note the placement of the semicolons in this program.

Figure 2.1 Basic HLA Program Layout

The *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 this name.

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 lower case). However, unlike other case sensitive languages, like C/C++, you may not declare two identifiers in the program whose name differs only by the case of alphabetic characters appearing in an identifier. Case neutrality enforces the good programming style of always spelling your names exactly the same

way (with respect to case) and never declaring two identifiers whose only difference is the case of certain alphabetic characters.

A traditional first program people write, popularized by K&R'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. Here's what the "Hello World" program looks like in HLA:

```

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

begin helloWorld;

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

end helloWorld;

```

Program 2.1 The Hello World 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` statements¹. Technically speaking, a semicolon is not necessary after 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 (note, however, that the HLA standard library include files always allow a semicolon after the corresponding `#INCLUDE` statement).

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 at least some of the HLA Standard Library header files ("stdlib.hhf" actually includes all the standard library definitions into your program; for more efficient compiles, you might want to be more selective about which files you include. You will see how to do this in a later chapter).

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

Note that HLA is a free-format language. Therefore, you may split statement across multiple lines (just like high level languages) if this helps to make your programs more readable. For example, the `stdout.put` statement in the HelloWorld program could also be written as follows:

```

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

```

1. Technically, from a language design point of view, these are not all statements. However, this chapter will not make that distinction.

Another item worth noting, since you'll see it cropping up 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, "nl" (the newline) is really nothing more than a string constant, so (technically) the comma between the *nl* and the preceding string isn't necessary. You'll often see the above written as:

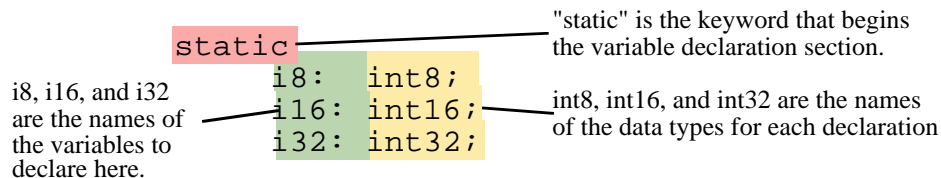
```
stdout.put( "Hello, World of Assembly Language" nl );
```

Notice the lack of a comma between the string constant and *nl*; this turns out to be perfectly legal in HLA, though it only applies to certain symbol string constants; you may not, in general, drop the comma. The chapter on Strings, later in this text, will explain in detail how this works. This discussion appears here because you'll probably see this "trick" employed by sample code prior to the formal discussion in the chapter on Strings.

2.4 Some Basic HLA Data Declarations

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

HLA predefines three different signed integer types: *int8*, *int16*, and *int32*, corresponding to eight-bit (one byte) signed integers, 16-bit (two byte) signed integers, and 32-bit (four byte) signed integers respectively². Typical variable declarations occur in the HLA *static variable section*. A typical set of variable declarations takes the following form



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

The diagram shows the code snippet above with three colored boxes and arrows pointing to explanatory text:

- A red box highlights the word `static`. An arrow points from it to the text: "static" is the keyword that begins the variable declaration section.
- A green box highlights the variable names `i8`, `i16`, and `i32`. An arrow points from it to the text: i8, i16, and i32 are the names of the variables to declare here.
- A yellow box highlights the data types `int8`, `int16`, and `int32`. An arrow points from it to the text: int8, int16, and int32 are the names of the data types for each declaration.

Figure 2.2 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 a little 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. The following figure demonstrates the syntax for this:

2. A discussion of bits and bytes will appear in the next chapter if you are unfamiliar with these terms.

```

static
  i8:   int8   := 8;
  i16:  int16  := 1600;
  i32:  int32  := -320000;

```

The constant assignment operator, "[:]" tells HLA that you wish to initialize the specified variable with an initial value.

The operand after the constant assignment operator must be a constant whose type is compatible with the variable you are initializing

Figure 2.3 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 may only declare 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.

Here is 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;

```

Program 2.2 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 the string representation of that integer's value on output. The second new feature this sample program introduces 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, this program 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 C++ and Delphi should recognize these comments.

2.5 Boolean Values

HLA and the HLA Standard Library provides limited support for boolean objects. You can declare boolean variables, use boolean literal constants, use boolean variables in boolean expressions (e.g., in an IF statement), 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 one; HLA represents false using the value zero. Most programs treat zero 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 you can see in this example, you may declare initialized as well as uninitialized variables.

Since boolean variables are byte objects, you can manipulate them using eight-bit registers and any instructions that operate directly on eight-bit values. Furthermore, as long as you ensure that your boolean variables only contain zero and one (for false and true, respectively), you can use the 80x86 AND, OR, XOR, and NOT instructions to manipulate these boolean values (we'll describe these instructions a little later).

You can print boolean values by making a call to the *stdout.put* routine, e.g.,

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

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

2.6 Character Values

HLA lets you declare one-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 using the *stdout.put* routine. We'll return to the subject of character constants a little later.

2.7 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 utilized to this point have been either data declarations or calls to HLA Standard Library routines. There hasn't been any *real* assembly language up to this point. Before we can progress any farther and learn

some real assembly language, a detour is necessary. For unless you understand the basic structure of the Intel 80x86 CPU family, the machine instructions will seem mysterious indeed.

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 devices* (I/O). These three components are connected together using the *system bus*. The following block diagram shows this relationship:

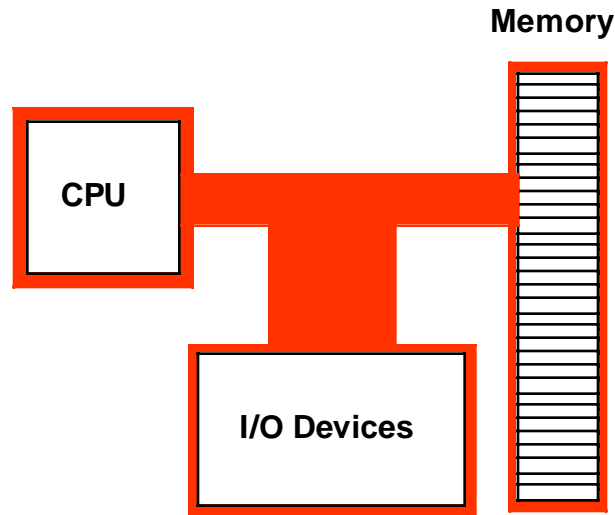


Figure 2.4 Von Neumann Computer System Block Diagram

Memory and I/O devices will be the subjects of later chapters; for now, let's take a look inside the CPU portion of the computer system, at least at the components that are visible to the assembly language programmer.

The most prominent items within the CPU are the registers. The Intel 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. This text will not consider the last two sets of registers. The segment registers are not used much in modern 32-bit operating systems (e.g., Windows, BeOS, and Linux); since this text is geared around 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 use by people who write operating systems, debuggers, and other system level tools. Such software construction is well beyond the scope of this text, so once again there is little need to discuss the special purpose kernel mode registers.

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. The following diagram shows this relationship:

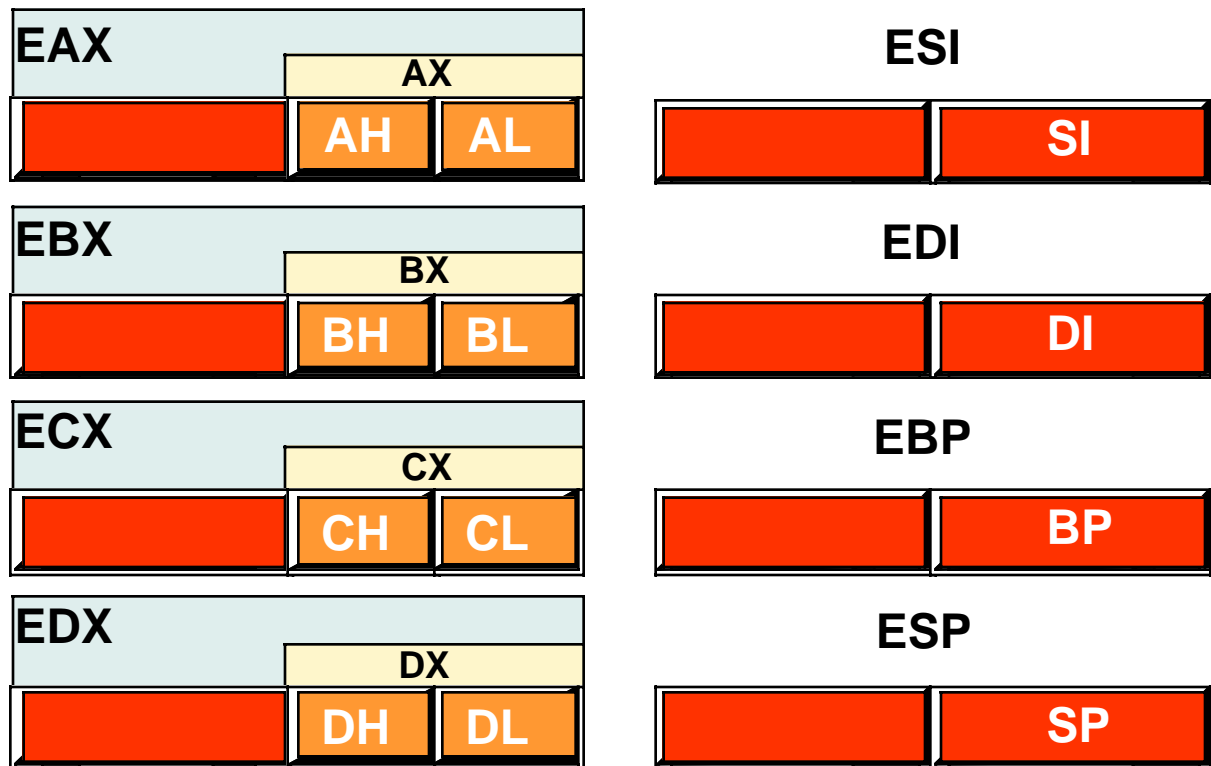


Figure 2.5 80x86 (Intel CPU) General Purpose Registers

The most important thing to note about the general purpose registers is that they are not independent. Modifying one register will modify at least one other register and 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 as well. 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 fully understand the ramifications of the above diagram.

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 disable³, sign, zero, auxiliary carry, parity, and carry flags. The following diagram shows their layout within the lower 16-bits of the EFLAGS register.

3. Application programs cannot modify the interrupt flag, but we'll look at this flag later in this text, hence the discussion of this flag here.

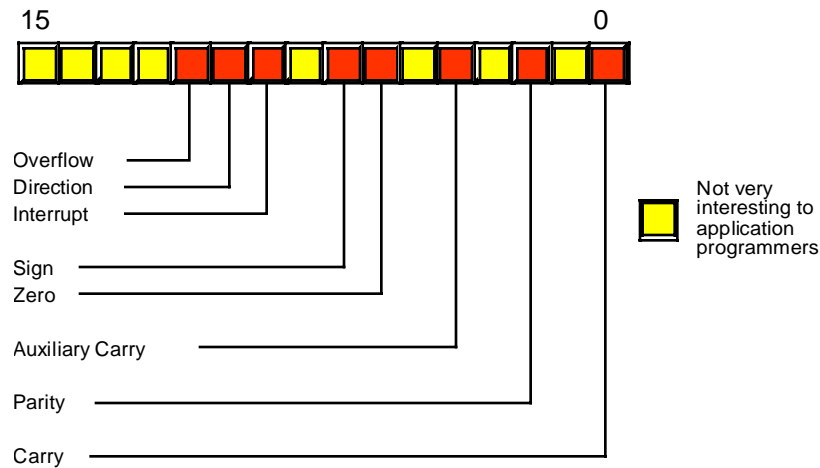


Figure 2.6 Layout of the FLAGS Register (Lower 16 bits of EFLAGS)

Of the eight flags that are usable by 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*⁴. The state of these flags (boolean variables) will let you test the results of previous computations and allow you to make decisions in your programs. For example, after comparing two values, the state of the condition code flags will tell you if one value is less than, equal to, or greater than a second value. The 80x86 CPUs provide special machine instructions that let you test the flags, alone or in various combinations.

The last register of interest is the EIP (instruction pointer) register. This 32-bit register contains the *memory address* of the next machine instruction to execute. Although you will manipulate this register directly in your programs, the instructions that modify its value treat this register as an implicit operand. Therefore, you will not need to remember much about this register since the 80x86 instruction set effectively hides it from you.

One important fact that comes as a surprise to those just learning assembly language is that almost all calculations on the 80x86 CPU must involve a register. For example, to add two (memory) variables together, storing the sum into a third location, you must load one of the memory operands into a register, add the second operand to the value in the register, and then store the register away in the destination memory location. 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 general purpose registers have the name “general purpose” you should not infer that you can use any register for any purpose. The SP/ESP register for example, has a very special purpose (it’s the *stack pointer*) that effectively prevents you from using it for any other purpose. Likewise, the BP/EBP register has a special purpose that limits its usefulness as a general purpose register. All the 80x86 registers have their own special purposes that limit their use in certain contexts. For the time being, you should simply avoid the use of the ESP and EBP registers for generic calculations and keep in mind that the remaining registers are not completely interchangeable in your programs.

2.8 Some Basic Machine Instructions

The 80x86 CPUs provide 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 period of time. Fortunately,

4. Technically the parity flag is also a condition code, but we will not use that flag in this text.

you don't need to know all the machine instructions. In fact, most assembly language programs probably use around 30 different machine instructions⁵. Indeed, you can certainly write several meaningful programs with only a small handful of 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 often-used assembly language statement. In a typical program, anywhere from 25-40% of the instructions are typically MOV instructions. As its name suggests, this instruction moves data from one location to another⁶. 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 16- or 32-bit 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 two eight-bit objects, between two 16-bit objects, or between two 32-bit objects; you may not, however, mix the sizes of the operands. The following table lists all the legal combinations:

Table 1: Legal 80x86 MOV Instruction Operands

Source	Destination
Reg ₈ ^a	Reg ₈
Reg ₈	Mem ₈
Mem ₈	Reg ₈
constant ^b	Reg ₈
constant	Mem ₈
Reg ₁₆	Reg ₁₆
Reg ₁₆	Mem ₁₆
Mem ₁₆	Reg ₁₆
constant	Reg ₁₆
constant	Mem ₁₆
Reg ₃₂	Reg ₃₂

5. Different programs may use a different set of 30 instructions, but few programs use more than 30 distinct instructions.

6. 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 should have been COPY. Alas, it's too late to change it now.

Table 1: Legal 80x86 MOV Instruction Operands

Reg ₃₂	Mem ₃₂
Mem ₃₂	Reg ₃₂
constant	Reg ₃₂
constant	Mem ₃₂

- a. The suffix denotes the size of the register or memory location.
- b. The constant must be small enough to fit in the specified destination operand

You should study this table carefully. Most of the general purpose 80x86 instructions use this same syntax. Note that in addition to the forms above, the HLA MOV instruction lets you specify two memory operands as the source and destination. However, this special translation that HLA provides only applies to the MOV instruction; it does not generalize to the other instructions.

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 must take the same form as the MOV instruction, listed in the table above⁷. The ADD instruction does the following:

```
destination_operand = destination_operand + source_operand ;
```

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

Similarly, 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. Here's a sample HLA program that demonstrates these three instructions:

```

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,

```

7. Remember, though, that ADD and SUB do not support memory-to-memory operations.

```

        "Initialized values: i8=", i8,
        ", i16=", i16,
        ", i32=", i32,
        nl
    );

    // Compute the absolute value of the
    // three different variables and
    // print the result.
    // Note, since all the numbers are
    // negative, we have to negate them.
    // Using only the MOV, ADD, and SUB
    // instruction, 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;

```

Program 2.3 Demonstration of MOV, ADD, and SUB Instructions

2.9 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 trivial 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. Later in this text, 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, you'll need to learn many new concepts before you get to that point, so we'll stick with these high level language statements for now since you're probably already familiar with statements like these from your exposure to high level languages.

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 real assembly language altogether. You will soon discover that these statements have some severe restrictions associated with them and you will quickly outgrow their capabilities (at least the restricted forms appearing in this section). 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.

2.9.1 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 same boolean expressions you use in a high level language and HLA only supports some basic forms.

HLA boolean expressions always take the following forms⁸:

```

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 following symbols:

- | | | |
|-------|--------------|--|
| • @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. |

8. Technically, there are a few more, advanced, forms, but you'll have to wait a few chapters before seeing these additional formats.

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 non-zero value.

If you specify a boolean variable as the expression, the program tests it for zero (false) or non-zero (true). Since HLA uses the values zero and one to represent false and true, respectively, the test works in an intuitive fashion. Note that HLA requires that stand-alone 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. The following table lists the legal combinations:

Table 2: Legal Boolean Expressions

Left Operand	Relational Operator	Right Operand
Memory Variable or Register	= or ==	Memory 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 as those allowed for the ADD and SUB instructions.

Also like the ADD and SUB instructions, the two operands must be the same size. That is, they must both be eight-bit operands, they must both be 16-bit operands, or they must both be 32-bit 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 of which you need to be aware. 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 2000 and 2099 (inclusive). The NOT IN (two words) operator lets you check 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 lower case 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'

```

2.9.2 The HLA IF..THEN..ELSEIF..ELSE..ENDIF Statement

The HLA IF statement uses the following syntax:

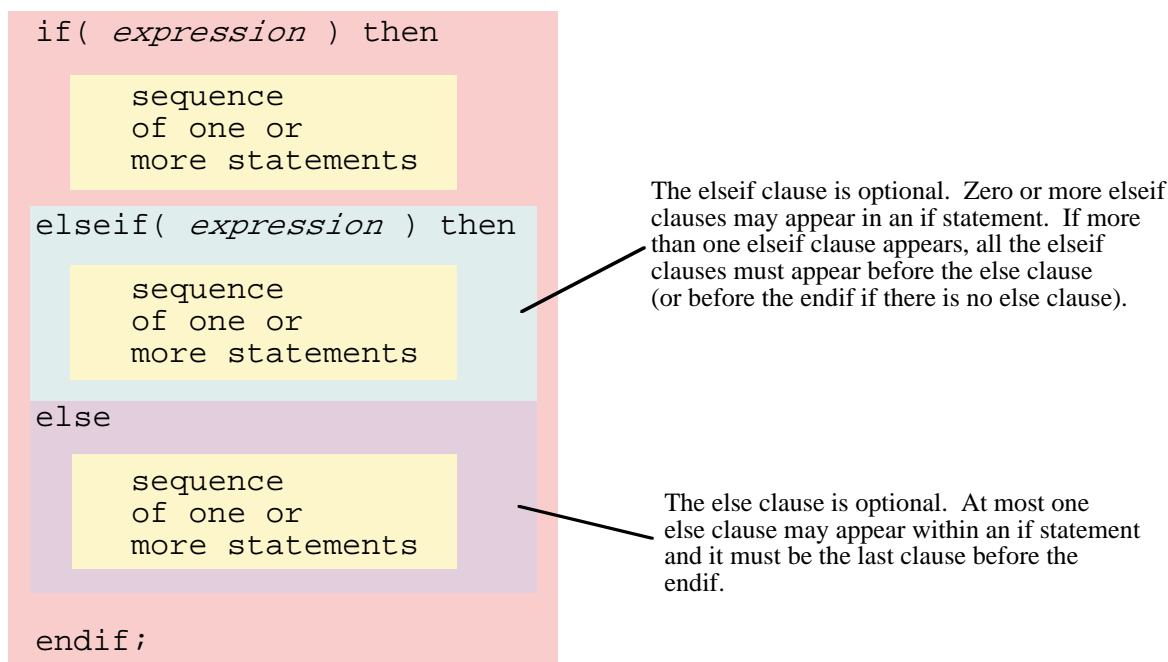


Figure 2.7 HLA IF Statement Syntax

The expressions appearing in this statement must take one of the forms from the previous section. If the associated expression is true, the code after the THEN executes, otherwise control transfers to the next ELSEIF or ELSE clause in the statement.

Since 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 an example of just 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:

```
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 multi-way 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 possibly void this assertion, so it's a good idea to have error reporting statements built into your code.

2.9.3 The WHILE..ENDWHILE Statement

The WHILE statement uses the following basic syntax:

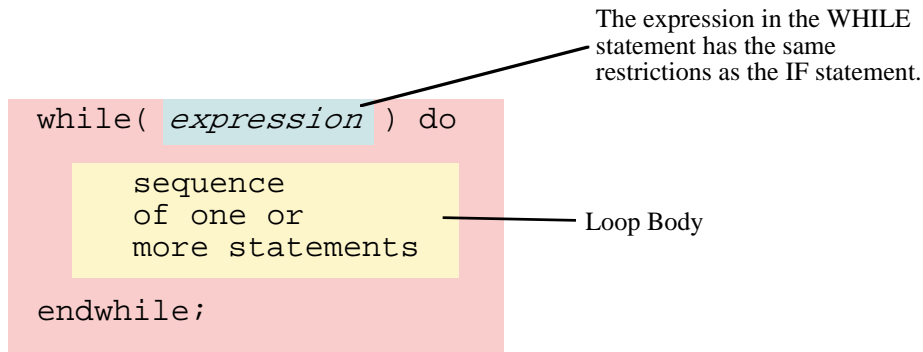


Figure 2.8 HLA While Statement Syntax

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 control falls through to 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 siblings, 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.

```

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

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

endwhile;

```

2.9.4 The FOR..ENDFOR Statement

The HLA FOR loop takes the following general form:

```

for( Initial_Stmt; Termination_Expression; Post_Body_Statement ) do

    << Loop Body >>

endfor;

```

This is equivalent to the following WHILE statement:

```

Initial_Stmt;
while( Termination_expression ) do

    << loop_body >>

    Post_Body_Statement;

endwhile;

```

Initial_Stmt 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 will execute. The *Post_Body_Statement* executes at the bottom of the loop (as shown in the WHILE example above). This is a single HLA statement. Usually it is an instruction like ADD that 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;
```

2.9.5 The REPEAT..UNTIL Statement

The HLA repeat..until statement uses the following syntax:

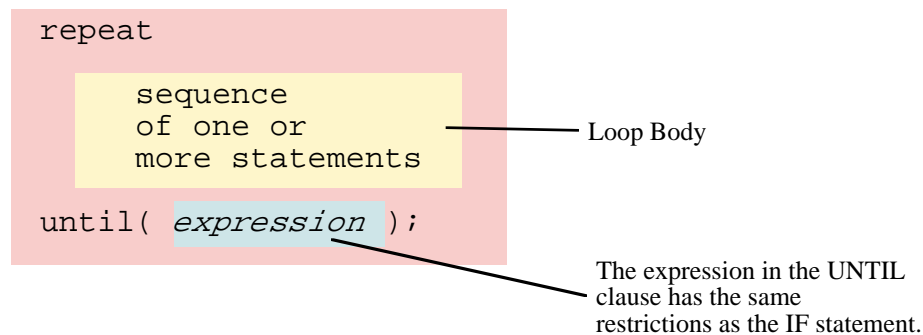


Figure 2.9 HLA Repeat..Until Statement Syntax

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 one use for the REPEAT..UNTIL statement:

```
mov( 10, ecx );
repeat

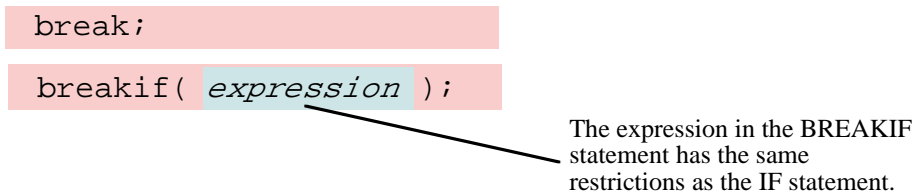
    stdout.put( "ecx = ", ecx, nl );
    sub( 1, ecx );

until( ecx = 0 );
```

If the loop body will always execute at least once, then it is more efficient to use a REPEAT..UNTIL loop rather than a WHILE loop.

2.9.6 The BREAK and BREAKIF Statements

The BREAK and BREAKIF statements provide the ability to prematurely exit from a loop. They use the following syntax:



```
break;
```

```
breakif( expression );
```

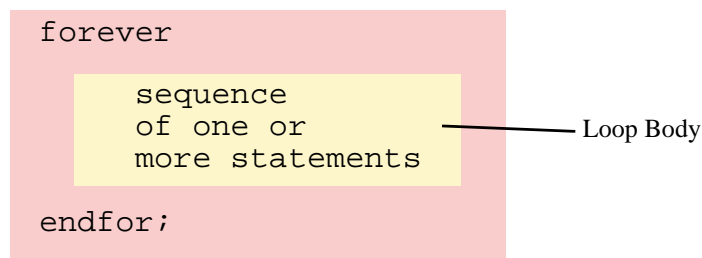
The expression in the BREAKIF statement has the same restrictions as the IF statement.

Figure 2.10 HLA Break and Breakif Syntax

The BREAK statement exits the loop that immediately contains the break; The BREAKIF statement evaluates the boolean expression and terminates the containing loop if the expression evaluates true.

2.9.7 The FOREVER..ENDFOR Statement

The FOREVER statement uses the following syntax:



```
forever
```

```
sequence
```

```
of one or
```

```
more statements
```

```
endfor;
```

Loop Body

Figure 2.11 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;
```


2.9.8 The TRY..EXCEPTION..ENDTRY Statement

The HLA TRY..EXCEPTION..ENDTRY statement provides very powerful *exception handling* capabilities. The syntax for this statement is the following:

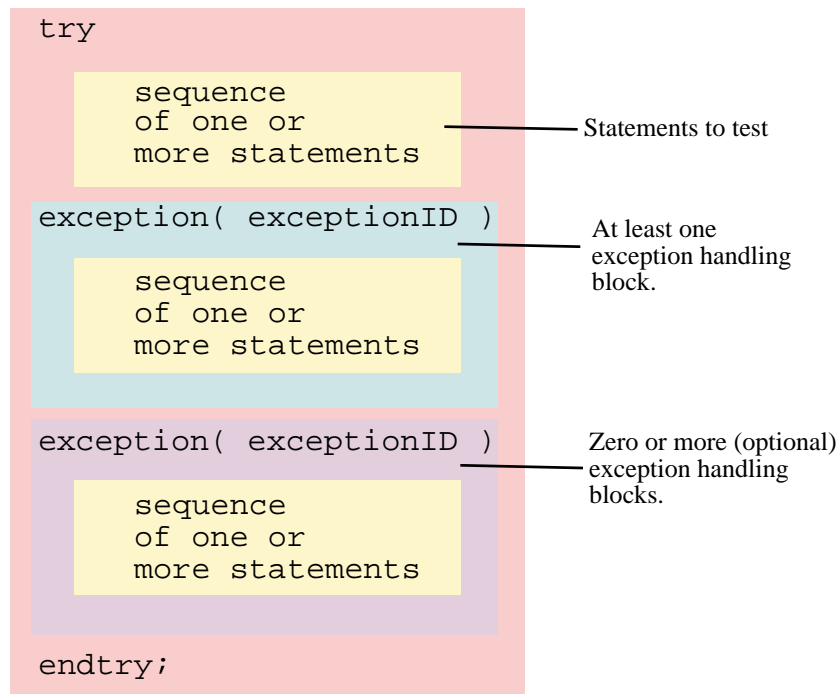


Figure 2.12 HLA Try..Except..Endtry Statement Syntax

The TRY..ENDTRY statement protects a block of statements during execution. If these statements, between the TRY clause and the first EXCEPTION clause, 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 one or more 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 sample program 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, 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 and control falls out of the TRY..ENDTRY statement and the REPEAT..UNTIL loop repeats since *GoodInteger* was never set to true. If a different exception occurs (one that is not handled in this code), then the program aborts with the specified error message⁹.

Please see the “excepts.hhf” header file that accompanies the HLA release for a complete list of all the exception ID codes. The HLA documentation will describe the purpose of each of these exception codes.

2.10 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 HLA feature leverages your high level language knowledge, reducing the need to learn arcane syntax, allowing you to learn assembly language more efficiently. The other half of the equation is the HLA Standard Library. The HLA Standard Library provides lot of commonly needed, easy to use, assembly language routines that you can call without having to write this code yourself (or even 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 weeks of 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 time¹⁰. HLA’s Standard Library continues the HLA tradition 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 accessible set of library routines¹¹. This high level interface,

9. 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 later in this text.

10. E.g., the UCR Standard Library for 80x86 Assembly Language Programmers.

11. HLA was created because MASM was insufficient to support the creation of the UCR StdLib v2.0.

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. The following table lists many of the modules that are available¹²:

Table 3: HLA Standard Library Modules

Name	Description
args	Command line parameter parsing support routines.
conv	Various conversions between strings and other values.
cset	Character set functions.
DateTime	Calendar, date, and time functions.
excepts	Exception handling routines.
fileio	File input and output routines
hla	Special HLA constants and other values.
Linux	Linux system calls (HLA Linux version only).
math	Transcendental and other mathematical functions.
memory	Memory allocation, deallocation, and support code.
misctypes	Miscellaneous data types.
patterns	The HLA pattern matching library.
rand	Pseudo-random number generators and support code.
stdin	User input routines
stdout	Provides user output and several other support routines.
stdlib	A special include file that links in all HLA standard library modules.
strings	HLA's powerful string library.
tables	Table (associative array) support routines.
win32	Constants used in Windows calls (HLA Win32 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.

12. Since the HLA Standard Library is expanding, this list is probably out of date. Please see the HLA documentation for a current list of Standard Library modules.

2.10.1 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. One constant you've seen already in code appearing in this chapter. 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 (this is a carriage return/line feed pair under Windows or just a line feed under Linux).

In addition to the *nl* constant, the HLA standard I/O library module defines several other useful character constants. They are

- `stdio.bell` The ASCII bell character. Beeps the speaker when printed.
- `stdio.bs` The ASCII backspace character.
- `stdio.tab` The ASCII tab character.
- `stdio.eoln` A linefeed character (even under Windows).
- `stdio.lf` The ASCII linefeed character.
- `stdio.cr` The ASCII carriage return character.

Except for *nl*, these characters appear in the *stdio* namespace (and, therefore, require the “stdio.” prefix). The placement of these ASCII constants within the *stdio* namespace is to help 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.

2.10.2 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*¹³. 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 window¹⁴:

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

¹³. Namespaces will be the subject of a later chapter.

¹⁴. Note for Linux 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”.

2.10.3 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);`” Of course, the call to `stdout.newln` is sometimes a little more convenient. Example of call:

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

2.10.4 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 );
```

2.10.5 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 *ValueX* 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 two), 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 eight-bit register as this parameter.

Here is a short HLA program that demonstrates the use of the `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;

```

Program 2.4 Columnar Output Demonstration Using `stdio.Puti32Size`

2.10.6 The `stdout.put` Routine

The `stdout.put` routine¹⁵ is 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. Since we've already been using this routine throughout this chapter, you've already seen lots of 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

15. *Stdout.put* is actually a macro, not a procedure. The distinction between the two is beyond the scope of this chapter. However, this text will describe their differences a little later.

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* routines¹⁶. The following sample program produces the same output as the previous program; however, it 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

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

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

    endwhile;
    stdout.put( nl );

end NumsInColumns2;

```

Program 2.5 Demonstration of 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.

2.10.7 The stdin.getc Routine.

The *stdin.getc* routine reads the next available character from the standard input device’s input buffer¹⁷. It returns this character in the CPU’s AL register. The following example program demonstrates a simple use of this routine:

16. 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.

17. “Buffer” is just a fancy term for an array.

```

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'.

        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;

```

Program 2.6 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 just a little later in this chapter.

2.10.8 The *stdin.getiX* Routines

The *stdin.geti8*, *stdin.geti16*, and *stdin.geti32* routines read eight, 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 white space characters (spaces, tabs, etc.) and then convert the following stream of decimal digits (with an optional, leading, minus sign) into the corresponding integer. These rou-

tines 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 following sample program 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;

```

Program 2.7 stdin.getiX Example Code

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

2.10.9 The `stdin.readLine` 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.geti8` call.

In general, the `stdin` routines only read text from the user 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 may 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 white space characters) thus preserving space on the screen. So the input buffer behavior is desirable every now and then.

Unfortunately, the buffered behavior of the input routines is definitely counter-intuitive 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.

2.10.10 The `stdin.get` Macro

The `stdin.get` macro combines many of the standard input routines into a single call, in much the same way that `stdout.put` combines all of the output routines into a single call. Actually, `stdin.get` is much easier to use than `stdout.put` since 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* macro, 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 registers¹⁸.

2.11 Putting It All Together

This chapter has covered a lot of ground! While you've still got 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.

In this chapter, you've seen the basic format for an HLA program. You've seen how to declare integer, character, and boolean variables. You have taken a look at the internal organization of the Intel 80x86 CPU family and learned about the MOV, ADD, and SUB instructions. You've looked at the basic HLA high level language control structures (IF, WHILE, REPEAT, FOR, BREAK, BREAKIF, FOREVER, and TRY) as well as what constitutes a legal boolean expression in these statements. Finally, this chapter has introduced several commonly-used routines in the HLA Standard Library.

You might think that knowing only three machine instructions is hardly sufficient to write meaningful programs. However, those three instructions (*mov*, *add*, and *sub*), combined with the HLA high level control structures and the HLA Standard Library routines are actually equivalent to knowing several dozen machine instructions. Certainly enough to write simple programs. Indeed, with only a few more arithmetic instructions plus the ability to write your own procedures, you'll be able to write almost any program. Of course, your journey into the world of assembly language has only just begun; you'll learn some more instructions, and how to use them, starting in the next chapter.

2.12 Sample Programs

This section contains several little HLA programs that demonstrate some of HLA's features appearing in this chapter. These short examples also demonstrate that it is possible to write meaningful (if simple) programs in HLA using nothing more than the information appearing in this chapter. You may find all of the sample programs appearing in this section in the "ch02" subdirectory of the "volume1" directory in the software that accompanies this text.

2.12.1 Powers of Two Table Generation

The following sample program generates a table listing all the powers of two between 2**0 and 2**30.

18. Note that register input is always in hexadecimal or base 16. The next chapter will discuss hexadecimal numbers.

```

// PowersOfTwo-
//
// This program generates a nicely-formatted
// "Powers of Two" table. It computes the
// various powers of two by successively
// doubling the value in the pwrOf2 variable.

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

static

    pwrOf2:    int32;
    LoopCntr:  int32;

begin PowersOfTwo;

    // Print a start up banner.

    stdout.put( "Powers of two: ", nl, nl );

    // Initialize "pwrOf2" with 2**0 (two raised to the zero power).

    mov( 1, pwrOf2 );

    // Because of the limitations of 32-bit signed integers,
    // we can only display 2**0..2**30.

    mov( 0, LoopCntr );
    while( LoopCntr < 31 ) do

        stdout.put( "2**(", LoopCntr:2, ") = ", pwrOf2:10, nl );

        // Double the value in pwrOf2 to compute the
        // next power of two.

        mov( pwrOf2, eax );
        add( eax, eax );
        mov( eax, pwrOf2 );

        // Move on to the next loop iteration.

        inc( LoopCntr );

    endwhile;
    stdout.newln();

end PowersOfTwo;

```

Program 2.8 Powers of Two Table Generator Program

2.12.2 Checkerboard Program

This short little program demonstrates how to generate a checkerboard pattern with HLA.

```

// CheckerBoard-
//
// This program demonstrates how to draw a
// checkerboard using a set of nested while
// loops.

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

static

    xCoord:      int8;    // Counts off eight squares in each row.
    yCoord:      int8;    // Counts off four pairs of squares in each column.
    ColCntr:     int8;    // Counts off four rows in each square.

begin CheckerBoard;

    mov( 0, yCoord );
    while( yCoord < 4 ) do

        // Display a row that begins with black.

        mov( 4, ColCntr );
        repeat

            // Each square is a 4x4 group of
            // spaces (white) or asterisks (black).
            // Print out one row of asterisks/spaces
            // for the current row of squares:

            mov( 0, xCoord );
            while( xCoord < 4 ) do

                stdout.put( "****  " );
                add( 1, xCoord );

            endwhile;
            stdout.newln();
            sub( 1, ColCntr );

        until( ColCntr = 0 );

        // Display a row that begins with white.

        mov( 4, ColCntr );
        repeat

            // Print out a single row of
            // spaces/asterisks for this
            // row of squares:

            mov( 0, xCoord );
            while( xCoord < 4 ) do

                stdout.put( "    ****" );
                add( 1, xCoord );

            endwhile;
            stdout.newln();
            sub( 1, ColCntr );

```

```

        until( ColCntr = 0 );

        add( 1, yCoord );

    endwhile;

end CheckerBoard;

```

Program 2.9 Checkerboard Generation Program

2.12.3 Fibonacci Number Generation

The Fibonacci sequence is very important to certain algorithms in Computer Science and other fields. The following sample program generates a sequence of Fibonacci numbers for $n=1..40$.

```

// This program generates the fibonacci
// sequence for n=1..40.
//
// The fibonacci sequence is defined recursively
// for positive integers as follows:
//
//  fib(1) = 1;
//  fib(2) = 1;
//  fib( n ) = fib( n-1 ) + fib( n-2 ).
//
// This program provides an iterative solution.

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

static

    FibCntr:    int32;
    CurFib:     int32;
    LastFib:    int32;
    TwoFibsAgo: int32;

begin fib;

    // Some simple initialization:

    mov( 1, LastFib );
    mov( 1, TwoFibsAgo );

    // Print fib(1) and fib(2) as a special case:

    stdout.put
    (
        "fib( 1) =      1", nl
        "fib( 2) =      1", nl
    )

```

```

);

// Use a loop to compute the remaining fib values:

mov( 3, FibCntr );
while( FibCntr <= 40 ) do

    // Get the last two computed fibonacci values
    // and add them together:

    mov( LastFib, ebx );
    mov( TwoFibsAgo, eax );
    add( ebx, eax );

    // Save the result and print it:

    mov( eax, CurFib );
    stdout.put( "fib(",FibCntr:2, ") =", CurFib:10, nl );

    // Recycle current LastFib (in ebx) as TwoFibsAgo,
    // and recycle CurFib as LastFib.

    mov( eax, LastFib );
    mov( ebx, TwoFibsAgo );

    // Bump up our loop counter:

    add( 1, FibCntr );

endwhile;

end fib;

```

Program 2.10 Fibonacci Sequence Generator
