Installing HLA Under Windows

New Easy Installation:

You can find a program titled "hlasetup.exe" on Webster. Running this application automatically installs HLA on your system. That's all there is to it. Those who wish to exercise complete control over the placement of the HLA executables can still choose to manually install HLA, but this is not recommended for first-time users.

Manual Installation under Windows:

HLA can operate in one of several modes. In the standard mode, it converts an HLA source file directly into an object file like most assemblers. In other modes, it has the ability to translate HLA source code into another source form that is compatible with several other assemblers, such as MASM, TASM, FASM, NASM, and Gas. A separate assembler, such as MASM, can compile that low-level intermediate source code to produce an object code file. Strictly speaking, this step (converting to a low-level assembler format and assembling via MASM/FASM/GASM/Gas is not necessary, but there are some times when it's advantageous to work in this manner. 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 or hlalib_safe.lib) and, possibly, several operating system specific library files (e.g., kernel32.lib under Win32). 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 your particular Operating System. These instructions describe installation under Windows; see the next section if you're using Linux, FreeBSD, or Mac OSX. 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.

The HLA package contains the HLA compiler, the HLA Standard Library, and a set of include files for the HLA Standard Library. It also includes copies of the Pelles C librarian and linker, and some other tools. These tools will let you produce executable files under Windows. However, it's still a good idea for Windows' users to grab a copy of the Microsoft linker. The easiest way to get all the Microsoft files you need is to download the *Visual C++ Express Edition* package from Microsoft's web site. As I was writing this, this package could be found at http://www.microsoft.com/Express/vc/.

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

- 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 HLA v2.2 "hla.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 hla.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 HLAPARSE.EXE. The HLA program is a "shell" program that runs the HLA compiler (HLAPARSE.EXE), MASM (ML.EXE), the linker (LINK.EXE), and other programs. You can think of HLA.EXE as the "HLA Compiler".
- Next, I set some environment variables:

```
path=c:\hla;%path%
set hlalib=c:\hla\hlalib
set hlainc=c:\hla\include
```

- HLA is a Win32 Console Window program. To run HLA you must open up a console Window. If you've downloaded the Visual C++ Expression Edition package, you should run the command prompt program from Visual C++ (generally found at Start→Programs→Microsoft Visual C++ Express Edition→Visual Studio Tools. If you've not installed the Visual C++ Expression Edition tools, you can usually find the command prompt program in places like Start→Programs→Accessories→Command Prompt (under Windows 2000) You might find it in another location. You can also start the command prompt processor by selecting Start->Run and entering "cmd".
- 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).
- Next, let's verify the correct operation of the linker. Type "link /?" (if you've installed the Microsoft linker as part of the Visual C++ Express Edition) or "polink /?" 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.

What You've Just Done:

Before describing how to write, compile, and run your very first HLA program, it's probably worthwhile to take a quick step back and carefully consider what we've accomplished in the previous section so it's easier to troubleshoot problems that may come up. This section describes how the steps you went through in the previous section affect the execution of HLA.

Note: This section is of interest to all HLA users; whether you've installed HLA via the HLASETUP.EXE program, or you've manually installed HLA.

In order to execute HLA.EXE, HLAPARSE.EXE, and various other programs needed to compile and run an HLA program, the operating system needs to be able to find all of the executable files that HLA needs. In theory, the executable files that HLA needs could be spread out all over your system as long as you tell the OS where to find every file. In practice, however, this makes troubleshooting the setup a lot more difficult if something goes wrong. Therefore, it's best to put all the necessary executables in the same directory.

Note that Windows doesn't know anything at all about the "C:\HLA" subdirectory; it's not going to automatically look for the HLA executables in this subdirectory, you have to tell Windows about this directory. This is done using the "PATH" environment variable. Whenever you tell Windows to run a program from a command prompt window, the OS first looks for a program with the given name in the current directory. If it finds the program (with a ".EXE", ".COM", or ".BAT" suffix), it will run that program from the current directory. If it does not find a program with an allowable suffix in the current directory, then the OS will use the PATH environment variable to determine which subdirectories to search through in order to find the executable program. The PATH environment variable is a (possibly empty) string that takes the following form:

pathToDirectory; *pathToDirectory*; *pathToDirectory*;...

Each *pathToDirectory* item in the list above generally represents a full path to some directory in the system. Examples include "c:\hla", "c:\windows", and "c:\bin"; these are always paths to directories, not to individual files. A PATH environment string may contain zero or more such paths; if there are two or more paths, each subdirectory path is separated from the others by a semicolon (the ellipses ["..."] above signify that you may have additional paths in the string, you don't actually place three periods at the end of the list).

When Windows fails to find an executable program you specify on the command line in the current directory, it tries searching for the executable file in each of the directory paths found in the PATH environment variable. Windows searches for the executable file in the subdirectories in the order that they appear in the environment string. That is, it searches for the executable in the first directory path first; if it doesn't find the executable in that directory, it tries the second path in the environment string; then the third, then the fourth, etc. If Windows exhausts the list of directory paths without finding the executable file, it displays an error message. For example, suppose your PATH environment variable contains the following:

```
c:\hla; c:\windows; c:\bin
```

If you type the command "xyz file1" at the command line prompt, Windows will first search for program "xyz.exe", "xyz.com", or "xyz.bat" in the current directory. Failing to find the program there, Windows will search for "xyz" in the "C:\HLA" subdirectory. If it's not there, then Windows tries the "C:\WINDOWS" subdirectory. If this still fails, Windows tries the "C:\BIN" subdirectory. If that fails, then Windows prints an error message and returns control to the command line prompt. If Windows finds the "xyz" program somewhere along the way, then Windows runs the program and the process stops at the first subdirectory containing the "xyz" program.

The search order through the PATH environment string is very important. Windows will execute the first program whose name matches the command you supply on the command line prompt. This is why the previous section had you put "c:\hla;" at the beginning of the environment string; this causes Windows to run programs like HLA.EXE, HLAPARSE.EXE, and LINK.EXE from the "C:\HLA" subdirectory rather than some other directory. This is very important! For example, there are many versions of the "LINK.EXE" program and not all of them work with HLA (and chances are pretty good that you might find an incompatible version of LINK.EXE on your system). Were you to place the "C:\HLA" directory path at the end of the PATH environment string, the system might execute an incompatible version of the linker when attempting to compile and link and HLA program. This generally causes the HLA compilation process to abort with an error. Placing the "C:\HLA" directory path first in the PATH environment variable helps avoid this problem[1].

By specifying the PATH environment variable, you tell Windows where it can find the executable files that HLA needs in order to compile your HLA programs. However, HLA and LINK also need to be able to find certain files. You may specify the location of these files explicitly when you compile a program, but this is a lot of work. Fortunately, both HLA.EXE and LINK.EXE also look at some environment variable strings in order to find their files, so you can specify these paths just once and not have to reenter them every time you run HLA.

Most HLA programs are not stand-alone projects. Generally, an HLA program will make use of routines found in the HLA Standard Library. The HLA Standard Library contains many conversion routines, input/output routines, string functions, and so on. Calling HLA Standard Library routines saves a considerable amount of effort when writing assembly language code. However, the HLA compiler isn't automatically aware of all the routines you can call in the HLA Standard Library. You have to explicitly tell the compiler to make these routines available to your programs by including one or more header files during the compilation process. This is accomplished using the HLA *#include* and *#includeonce* directives. For example, the following statement tells the HLA compiler to include all the definitions found in the "stdlib.hhf" header file (and all the header files that "stdlib.hhf" includes):

```
#include( "stdlib.hhf" )
```

When HLA encounters a #include or #includeonce directive in the source file, it substitutes the content of the specified file in place of the *#include* or *#includeonce* . The question is "where does this file come from?" If the string you specify is a full pathname, HLA will attempt to include the file from the location you specify; if it cannot find the file in the specified directory, the HLA will report an error. E.g.,

```
#include( "c:\myproject\myheader.hhf" )
```

In this example, HLA will look for the file "myheader.hhf" in the "c:\myproject" directory. If HLA fails to find the file, it will generate an error during compilation.

If you specify a plain filename as the *#include* or *#includeonce* argument, then HLA will first attempt to find the file in the current directory (the one you're in when you issued the HLA command at the command line prompt). If HLA finds the file, it substitutes the file's contents for the include directive and compilation continues. If HLA does not find the file, then it checks out the "HLAINC" environment variable, whose definition takes the following form:

```
hlainc=c:\hla\include
```

Unlike the PATH environment variable, the HLAINC environment variable allows only a single directory path as an operand. This is the path to the HLA include files directory which is "c:\hla\include" (assuming you've followed the

directions in the previous section). Since HLA header files usually come in two varieties: headers associated with library routines and header files associated with the current project, the fact that HLA only provides one include path is not much of a limitation. You should keep all project-related header files in the same directory as your other source files for the project; the HLA library header files (and header files for any generic library modules you write) belong in the "C:\HLA\INCLUDE" directory. Note that if you want to place header files in some directory other than the current directory or the directory that the HLAINC environment variable specifies, you will have to specify the path to the include file in the *#include* or *#includeonce* statement.

If HLA cannot find the specified header file in the current directory or in the directory specified by the HLAINC environment variable, then you'll get an error to the effect that HLA cannot find the specified include file. Needless to say, most assemblies will fail if HLA cannot find the appropriate header files.

Since most HLA programs will use one or more of the HLA Standard Library header files, chances are good that the assembly won't be successful if the HLAINC environment string is not set up properly. Conversely, if HLAINC does contain the appropriate string, then HLA will be able to successfully compile your code to assembly and produce an object file. The last step, converting the object file to an executable, introduces another possible source of problems. The LINK program combines the object file associated with your code with HLA Standard Library and Windows library modules. Even the most trivial HLA program will need to link with (at least) one or more Windows module. (The most trivial HLA program is probably the one that immediately returns to the OS; such a program needs to call the Windows ExitProcess API in order to return to the OS, so at the very least you'll need to link with the Windows' kernel32.lib module to be able to call ExitProcess.) Once again, however, the library modules that your program needs could be anywhere on the disk. You'll have to use some environment variables to tell HLA and the linker where it can find the library modules. You accomplish this using two environment variables: one for HLA and one for the linker. We'll discuss the HLALIB environment variable first, since it's the easiest to understand.

The HLA Standard Library contains thousands of small little routines that have been combined into a single file: either "hlalib.lib" or “hlalib_safe.lib” (“hlalib_safe.lib” is a *thread-safe* version of the library). Whenever you call a particular standard library routine, the linker extracts the specific routine you call from the "hlalib.lib" library module and links combines this code with your program. Once again, the linker program and HLA don't know where to find these files, you have to tell them where to find it. This is done using the HLALIB environment variable; this environment variable contains a single pathname to the directory containing the HLALIB.LIB and HLALIB_SAFE.LIB files. I.e.,

```
set hlalib=c:\hla\hlalib
```

The important thing to note in this example is that the path is the directory containing the HLALIB.LIB and HLALIB_SAFE.LIB files. This is in direct contrast to the PATH and HLAINC environment objects where you specify only the subdirectory containing the desired files.

In addition to the "hlalib.lib" or “hlalib_safe.lib” library file, you'll also need to link your HLA programs against various Windows library modules. Basic HLA programs will need to link against the Windows' kernel32.lib, user32.lib, and gdi32.lib library modules. These library modules (plus several other Windows related library modules) are found in the Visual C++ Express Edition package; though in the previous section you should have copied these three library modules to the "c:\hla\hlalib" subdirectory. You'll need to tell the LINK.EXE program where it can find these files; this is done with the LIB environment variable. The LIB environment variable's syntax is very similar to that for the PATH environment variable. You get to specify a list of directories in the LIB environment string and LINK.EXE will automatically search for missing library files in each of the paths you specify, in the order you specify, until it finds a matching filename. By prepending "c:\hla\hlalib;" to this environment string, you tell LINK.EXE to search for library modules like KERNEL32.LIB, USER32.LIB, and GDI32.LIB in the "C:\hla\hlalib" subdirectory if it doesn't find them in the current subdirectory.

Note: in the future, if you make other Win32 API calls you may need to copy additional .LIB files from the Visual C++ Express Edition package to the "c:\hla\hlalib" directory. However, for most basic HLA programs (and certainly, all console mode programs) you won't need to do this. Another alternative would be to add the path to the Visual C++ Express Edition's libraries directory to the LIB environment string (generally, installing the Visual C++ Express Edition package does this for you).

Okay, with this explanation out of the way, it's time to write, compile, and run, our first HLA program!

Running HLA:

Now it's time to try your hand at writing an honest to goodness HLA program and verify that the whole system is working. A long running convention is to write a "Hello World" programs as the very first program in any language. This document will continue that cherished convention.

When writing HLA programs, the best approach is to create a single directory for each project you write. I'd suggest creating a subdirectory "c:\hla\projects" and then create a new subdirectory inside "c:\hla\projects" for each HLA project you write. For the example in this section, you might create the directory "c:\hla\projects\hw" (for "Hello World").

There are many different ways to create project directories. The most common way to do this under Windows is in the Windows Explorer (right click on a window and select "New→Folder"). However, since HLA is a command-line based tool, it's probably best to describe how to do this from the command line just to introduce some (possibly) new command line commands.

The first step is to bring up a command prompt window. Generally, this is done by selecting "Start→Programs→Accessories→Command Prompt" from the Windows Start menu. You may also select "Start→Run" and type "cmd" to bring up a command prompt shell. If you wind up writing a lot of HLA code, you'll want a faster and easier way to run the command prompt, so I recommend creating a shortcut to the command prompt program on your desktop. To do this, click on Start (and release the mouse button) and then move the mouse cursor to select "Programs→Accessories→Command Prompt". Now right-click on the "Command Prompt" menu item and drag it onto your desktop. Select "Create Shortcut(s) Here" from the resulting pop-up menu.

If you're using Visual C++ Express Edition, then select Start→Programs→Microsoft Visual C++ Express Edition→Visual Studio Tools and right-click on that menu item. You should now have a shortcut to the command prompt shell program on your desktop and you can easily run the shell by double-clicking on its icon.

After creating a shortcut, there are some things you can do to make HLA development a little easier. Right click on the shortcut you've just created and select properties. In the window that pops up, you'll probably want to change the "Start In:" string to "c:\hla\projects". This tells the command prompt program to make the specified path the current directory whenever you run the command prompt program by double clicking on this icon. By starting off inside the c:\hla\projects" subdirectory, you'll find that you save some typing (assuming, of course, you wind up putting most of your projects in the "c:\hla\projects" directory; use a different path here if this is not the case). Next, select the "layout" tab in the Command Prompt Properties window. Under "Screen Buffer Size" you'll probably want to make the value larger (the default is 300 on my system). I've found that 3000 is a good number here. This number specifies the number of lines of text that the system will save when data scrolls off the top of the screen. This lets you view up to 3,000 lines of text from program execution (including your program's output, HLA error messages, etc.). If you have a large monitor, you might also want to change the Window Size values as well. The default of 80 columns is probably fine, though you may want to expand the height to 50 lines (or whatever your monitor allows). Once you've set up the command window properties, double-click on the command prompt icon to start it running.

The first step, before doing anything else, is to verify that you've properly set up the environment variables. To check out their values, simply type "set" followed by ENTER. The "set" command without any parameters tells the command shell to dump the current values of all the environment variables active in the command prompt window. Scan through this list (scrolling back using the scroll bar if there are too many definitions to all fit in the window at one time) and search for the PATH, LIB, HLAINC, and HLALIB environment variables. Verify that they are present and have the correct values (that you've entered in the previous sections). If they're not present or the values are incorrect, HLA will not execute properly and you need to fix this problem first (Win95/98 users, did you remember to reboot after changing the autoexec.bat file?). If the environment variables are present and correct, then you're ready to try writing and running your first HLA program.

Switch to the "C:\HLA" subdirectory by using the following DOS (command line prompt) command[5]:

```
cd c:\hla
```

"cd" stands for "Change Directory". This command expects a single command line parameter that is the path of the directory that you want to make the "current directory." If you ever want to know what the current directory is, simply type "cd" without any parameters and the Command Shell will display the current directory[6]

If you haven't done so already, it's time to create the "projects" directory where you will place the HLA projects you create. Do this with the following command:

```
mkdir projects
```

"mkdir" stands for "make directory" and this command requires a single argument - the name of the directory you want to create. If you do no specify a full pathname (as is the case in this example), the command shell creates the directory within the current directory. Verify that you've properly created the "projects" subdirectory by issuing the following DOS command:

```
dir
```

"dir" stands for "directory" and tells the command shell to display all the files in the current directory. This should list the projects directory you just created (plus all the other files and directories in the "c:\hla" directory). Note that you can also use the Windows Explorer to create directories and view the contents of directories. However, since HLA is a command prompt based application, it's useful to learn a few commands that will prove useful.

Now, switch to the projects subdirectory by using the following command:

```
cd projects
```

At this point, it's a good idea to create a new subdirectory for the "Hello World" project. Do this by using the following command:

```
mkdir hw
```

CD into this directory once you've created it. Now you're ready to begin work on the "Hello World" program.

Leaving the command prompt Window open for the time being, run the editor of your choice (e.g., NOTEPAD.EXE, if you don't have any other preferences). Enter the following HLA program into the editor:

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

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

end HelloWorld;
```

Scan over the program you've entered and verify that you've entered it exactly as written above. Save the file from your editor as hw.hla in the c:\hla\projects\hw subdirectory (generally using the File>Save or File>Save As menu item in the editor). Switch over to the command prompt window and verify that the file is present by issuing a "DIR" command; this should list the hw.hla file. If it's not present, try saving the file again and be sure to browse to the "c:\hla\projects\hw" before actually saving the file.

WARNING: NOTEPAD.EXE has a habit of tacking a ".txt" to the end of filenames you save to disk. Default installations of Windows do not display file suffixes of known file types (and ".txt" is a known type). Therefore, a directory window may show a program name like "hw.hla" when, in fact, the filename is "hw.hla.txt". Probably the number one problem people have when testing out their HLA installation is that the compiler claims it cannot find the file "hw.hla" when the user first attempts to compile the file. This is because it's really named "hla.hw.txt". You must change the filename to "hw.hla" before HLA will accept it. This is the number one HLA installation problem. Watch out for this!

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 output similar to the following (this changes all the time, so don't expect it to be exact):

```
HLA (High Level Assembler)
Use '-license' to see licensing information.
Version 2.0 build 407 (prototype)
Win32 COFF output
OBJ output using HLA Back Engine
-test active

HLA Lib Path:     g:\hla\hlalib\hlalib.lib
HLA include path: g:\hla\hlalibsrc\trunk\hlainc
HLA temp path:
Linker Lib Path:  C:\Program Files\Microsoft Visual Studio 9.0\VC\LIB;C:\Program
Files\Microsoft SDK
s\Windows\v6.0A\lib;g:\hla\hlalib;g:\hla\hlalib

Files:
1: hw.hla

Compiling 'hw.hla' to 'hw.obj'
using command line:
[hlaparse -WIN32 -level=high  -v -ccoff -test "hw.hla"]

----------------------
HLA (High Level Assembler) Parser
use '-license' to view license information
Version 2.0 build 406 (prototype)
-t active
File: hw.hla
Output Path: ""
hlainc Path: "g:\hla\hlalibsrc\trunk\hlainc"
hlaauxinc Path: "(null)"
Compiler running under Windows OS
Back-end assembler: HLABE
Language Level: high

Compiling "hw.hla" to "hw.obj"
Compilation complete, 18758 lines,   0.454 seconds,   41317 lines/second
------------
HLA Back Engine Object code formatter

HLABE compiling 'hw.hla' to 'hw.obj'
Optimization passes: 3+2
----------------------
Linking via [link @"hw.link._.link"]
Microsoft (R) Incremental Linker Version 9.00.30729.01
Copyright (C) Microsoft Corporation.  All rights reserved.

-heap:0x1000000,0x1000000
-stack:0x1000000,0x1000000
-base:0x4000000
-entry:HLAMain
-section:.text,ER
-section:.data,RW
-section:.bss,RW
kernel32.lib
user32.lib
gdi32.lib
-subsystem:console
```

```
-out:hw.exe
g:\hla\hlalib\hlalib.lib
hw.obj
ctl32.dll
```

If you get output that is similar to the above, you're in business.

Manually installing HLA is a complex and slightly involved process. Fortunately, the hlasetup.exe program automates almost everything so that you don't have to worry about changing registry settings and things like that. If you're a first-time HLA user, you definitely want to use this method to install HLA. Manual installation is really intended for upgrades as new versions of HLA appear. You do not have to change the environment variables to install a new version of HLA, simply extract the executable files over the top of your existing installation and everything will work fine.

The most common problems people have running HLA involve the location of the Win32 library files, the choice of linker, and appropriately setting the hlalib, halib_safe, and hlainc environment variables. 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 Microsoft Visual C++ user, installation of VC++ should have set up the LIB path for you. If not, then locate these files and copy them to the HLA\HLALIB directory. If these files are not present on your system, you should download the Microsoft Visual C++ Express edition to obtain them.

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. HLA ships with a copy of the Pelles linker (polink.exe) that you can use if you've not downloaded Microsoft's Visual C++ Express edition.

Standard Configurations Under Windows

The "standard" HLA configuration under Windows consists of HLA.EXE, HLAPARSE.EXE, PORC.EXE, and POLINK.EXE. This standard configuration generates object files directly, compiles any resource files using the Pelles C PORC.EXE resource compiler, and links the object modules together using the Pelles C linker (POLINK). The Pelles C tools were chosen for the standard configuration under Windows because they are freely distributable (unlike the Microsoft tools). For those who care about such things, HLAPARSE.EXE produces object modules directly. HLA's object code generator produces slightly more optimal output code than FASM, MASM, or TASM.

HLA also provides the ability to produce assembly language source files, in a MASM, TASM, FASM, NASM, or Gas format that can be assembled to object code using one of these assemblers. So why would anyone want to have HLA produce assembly language output to be run through a different assembler (much like GCC does)? For common applications, there is no need to do this. However, in some specialized situations having this facility is quite useful. For example, rather than using the HLA back-engine native code generator, you may elect to have HLA generate FASM source code to be processed by the FASM assembler. There are three reasons for doing this:

- You want to see how HLA would translate the HLA program (in HLA syntax) to a lower-level assembly language (in FASM syntax); this is great, for example, for seeing how macros expand or how HLA processes high-level control constructs.
- If there is a defect in the internal HLA back engine that prevents HLA from directly generating an object code file, you can probably produce a source file and successfully compile your program using the external version of FASM.

Another configuration is to have HLA produce a MASM compatible output file and use Microsoft's MASM to translate that output source file into an object file. There are several reasons why you might want to use MASM:

- MASM can inject additional symbolic debugging information (usable by Visual Studio's debugger) into the object file, making it easier to debug HLA applications using Visual Studio.
- FASM and HLABE may have some code generation defect that you can't work around.

- The HLA back engine's output might not be completely compatible with some other object module tool you're using.
- You want to take HLA output and merge it with some MASM projects you have.

Although MASM is not a freely distributable program (and, therefore, is not included in the HLA download), you may download a free copy from the Microsoft Web site or obtain a copy as part of the Visual C++ Express Edition package.

Another configuration is to have HLA produce a NASM compatible output file and use the Netwide Assembler (NASM) to translate that output source file into an object file. There are a couple of reasons why you might want to use NASM:

- HLA's back engine may have some code generation defect that you can't work around.
- HLA's back engine output might not be completely compatible with some other object module tool you're using and NASM's output is compatible.
- You want to take HLA output and merge it with some NASM projects you have.
- You want to compile the code on an operating system that supports NASM but doesn't directly support HLA.

One last assembler choice under Windows is Borland's Turbo Assembler (TASM). There is one main reason why you would want to use TASM to process HLA output – you want to link HLA output with a Borland Delphi project. Delphi is very particular about the object files it will link against. Effectively, you can only use TASM-generated output files when linking with Delphi code. Therefore, if you want to link your HLA modules into a Delphi application, you'll need to use the TASM output mode. Like MASM, TASM is not a freely distributable product and cannot be included as part of the HLA download. However, Borland will provide a free copy as part of their free C++ download on their website (registration required). Note that TASM support in HLA has been deprecated and stop functioning as time passes.

Under Windows, you may use either the freely distributable Pelle's C linker (Polink) or the Microsoft linker to process the object code output from the HLA system. Polink is provided with the HLA download (subject, of course, to the Pelles C license agreement). Microsoft's linker is a commercial product (and as such, it is not included as part of the HLA download), but it is available as a free download from Microsoft's web site and as part of the Visual C++ express edition package. HLA will use either linker as the final stage in producing an executable. The Microsoft linker has been around longer and has, arguably, fewer bugs than Polink, but the choice is yours. Another possible linker option is the Borland Turbo linker (TLINK). Just note that HLA.EXE will not automatically run TLINK; you will have to run it manually after producing an OMF object file with HLA. Note that only MASM and TASM are capable of producing OMF files. FASM and HLA's internal code generator do not generate OMF object code files, so you cannot use TLINK with their output.

To produce libraries, you may optionally employ a librarian such as Microsoft's LIB.EXE, the Pelle's C POLIB.EXE, or Borland's Turbo Librarian (TLIB.EXE). The HLA.EXE program does not automatically run these programs; you will have to run them manually to create a .LIB file from your object files. Please see the documentation for these products for details on their use. The HLA download includes the POLIB.EXE program and the HLA standard library source code includes a make file option that will use any of these three librarians to produce the HLA hlalib.lib library file.

Note that it is possible to mix and match modules in the HLA system, within certain reasonable limitations. For example, you could use the FASM assembler and the Microsoft linker, the TASM assembler and the POLINK linker, or even the MASM assembler the TLINK linker. In general, FASM output works fine with the Microsoft linker and librarian or the Pelle's C linker and librarian, MASM output works best with Microsoft's linker and librarian, and Turbo assembler works best with the Borland tools or the Microsoft tools.

Under Windows, the default configuration is to generate an MSCOFF object file directly and use the POLINK linker to process the resulting object file(s). See the section on "Customizing HLA" for details on changing the default configuration.