15 HLA Units and External Compilation

15.1 HLA Units and External Compilation

This section discusses how to create separately compilable modules in HLA and how you can link HLA code with code written in other languages.

15.2 External Declarations

HLA provides two features to support separate compilation: units and external objects. HLA uses a very general scheme, similar to C++ to communicate linkage information between object modules. This scheme lets HLA programmers link to their HLA programs code written in HLA, non-HLA assembly code (e.g., MASM), and even code written in other high level languages (HLLs). Conversely, the HLA program can also write modules to be linked with programs written in this other languages (as well as HLA).

Writing separate modules is quite similar to writing a single HLA program. The first thing to note is that an executable can have only one main program. When writing HLA programs, the **program** reserved word tells HLA that you are writing a module that contains a main program. When writing other modules, you must use a **unit** rather than a **program** so as not to generate an extra main procedure. If you wish to write a library module that contains only procedures and no main program, you would use an HLA unit. Units have a syntax that is nearly identical to programs, there just isn't a **begin** associated with the **unit**, e.g.,

```
unit UnitName;

   << Declarations >>

end UnitName;
```

Since a **unit** does not contain a main program, it cannot compile into a stand-alone program; therefore, you should always compile units with the "-c" command line option to avoid running the linker on the unit code (which will always produce a link error)[1].

HLA uses the **external** keyword to communicate names between modules in a compilation group. If a symbol is defined to be external, HLA assumes that the symbol is declared in a separate module and leaves it up to the linker to resolve the symbol's address.

Only two types of symbols may be external: subroutines (procedures, methods, and iterators) and static variables[2]. Variables declared in the **var** section cannot be external because the linker cannot statically resolve their run-time address. Constants declared in the **const** or **val** sections cannot be external, however this is not a limitation because most programmers place public constants in header files and include them in the source files that require them.

Recall the syntax for an original-style procedure declaration presented in the chapter on procedure declarations:

```
procedure identifier ( optional_parameter_list ); procedure_options
   declarations
begin identifier;
   statements
```

1. Actually, the HLA.EXE program allows you to specify several ".HLA" files on the command line. The command line option "-c" is only necessary if none of the files on the command line contain a main program.

2. For the purposes of this discussion, variables appearing in the READONLY, and STORAGE sections are treated as static variables along with variables declared in the STATIC section.

```
end identifier;
```

There are two additional forms to consider:

```
procedure identifier ( optional_parameter_list );
    options
    external;

procedure identifier ( optional_parameter_list );
    options
    external("extname");
```

These two forms tell the HLA compiler that it is okay to call the specified procedure, but the procedure itself may not otherwise appear in the current source file. It is the responsibility of the linker to ensure that the specified external procedures actually appear within the object modules the linker is combining.

The first form above is generally used when the external procedure is an HLA procedure that appears in a different source module. HLA assumes that the external name is the same name as the procedure identifier.

The second form above is generally used when calling code written in a language other than HLA[1]. This form lets you explicitly state (via the string constant "extname") the name of the external procedure. This is especially important when calling procedures whose names contain characters that are not "HLA-Friendly." For example, many Windows API calls have at signs ("@") in their names; to call such routines you would use the second form of the external declaration above supplying the Windows API compatible name as the parameter to the **external** reserved word.

It is legal to declare an external procedure in the same source file that the procedure's actual code appears. However, the external declaration must appear *before* the actual declaration or HLA will generate an error. Whenever an external declaration appears in the same source file as the actual procedure code, HLA emits code to ensure that the procedure's name is *public*. Therefore, the external declaration *must* appear in the same file as the procedure's code if you wish the linker to be able to resolve the procedure's address at link time. This external declaration serves the same purpose as the "public" directive in other assemblers (e.g., MASM). Note that, unlike C/C++, procedure names are not automatically public. An external declaration must appear in the same file as the procedure code to make the symbol public.

Also, note above that the only options an external procedure declaration supports are the **@returns, @pascal, @cdecal,** and **@stdcall** options. You cannot use the **@align, @noalignstack, @noframe** or **@nodisplay** options in an external declaration. Conversely, if an **external** (or **forward**, for that matter) declaration appears in a source file, the corresponding procedure code may only contain the **@align, @noalignstadk, @noframe,** and/or **@nodisplay** options. The **@returns, @pascal, @cdecl,** and **@stdcall** options are not legal in a procedure declaration if a corresponding **external** (or **forward**) declaration is present in the source code.

Note: External procedures are only legal at lex level one. You cannot declare an external procedure that is embedded inside another procedure.

In addition to procedures, HLA also lets you declare **external** variables. You may reference such variables in different source modules. The declaration of an external variable is very similar to the declaration of an external procedure: you follow the variable's name with the external clause. If an optional string parameter is not present, HLA uses the variable's name as it's external name. If you need to specify a specific name, to avoid conflicts with other languages or to contain characters illegal in an HLA identifier, then provide a string with the identifier you need.

Note that HLA does not allow the **external** keyword after every static declaration. Instead, only the following variable declarations allow the **external** keyword:

```
name: procedure optional_parameters; @external;
name: pointer to typename; @external;
name: typename; @external;
```

1. Or when the HLA procedure name is a MASM reserved word.

```
name: typename [ dimensions ]; @external;
```

In particular, note that static variable declarations with initializers cannot be external. Also note that **enum, record,** and **union** variables (those variables you directly create as **enum, record,** or **union**) may not be external. This is not a serious limitation, however, since you can declare a named type in the **type** section and use the third form above to create an external object of the desired type (this is also how you would declare **external** class variables).

Like the C/C++ language, you normally put all your external declarations in a header file and include that header file using the **#include** directive in each of the source files that reference the external symbols. This eases program maintenance by having to change only a single definition in an include file rather than multiple definitions across different source files (if not using include files). See the HLA Standard Library code for some good examples of using HLA header files.

By convention, HLA header files that contain external declarations always have an ".HHF" suffix (HLA Header File). To help make your programs easy to read by others, you should always use this same suffix for your HLA header files.

15.3 HLA Naming Conventions and Other Languages

If you wish to link together code written in a different language with code written in HLA, you must be aware of the differences in naming conventions between the two languages.

With respect to names, keep in mind that HLA is a case-neutral language. To the outside world, this means that HLA is case sensitive. Therefore, all public names that HLA exports are case sensitive. If you are using a case insensitive language like Pascal or Delphi, you should check with your compiler vendor to determine how the language emits public names (usually, case insensitive languages convert all public symbols to all upper case or all lower case). Some languages, e.g., MASM, let you choose whether public symbols are case sensitive or case insensitive; for such languages, you should select case sensitivity as the default and spell your names the same (with respect to case) between the HLA code and the other language.

In some cases, it might not be possible to match an HLA identifier with a public or external identifier in another language. One possible reason for this problem is that HLA only allows alphanumeric characters and underscores in identifiers; some other languages (e.g., MASM) allow other characters in their names while other languages (e.g., C++) often "mangle" their names by adding additional characters that are normally illegal within identifiers (e.g., the at sign, "@").

The HLA **external** directive provides an option that lets you use a standard HLA identifier within your program, but utilize a completely different identifier as the public symbol. The standard HLA identifier restrictions do not apply to the external name[1]. This variant of the external directive takes the following forms:

External procedure declaration:

```
procedure ProcName; @external( "ExtProcName" );
```

External variable declaration:

```
varName: SomeType; @external( "ExtVarName" );
```

Within the confines of the HLA program, you would use the HLA identifiers *ProcName* and *varName*. To the outside world, however, you would use the names *ExtProcName* and *ExtVarName* to reference these objects.

Since the **external** parameter is a string constant rather than an HLA identifier, you can use characters that would otherwise be illegal in an HLA identifier. For example, Microsoft's Visual C++ language and Windows often insert the "@" symbol into identifiers. Normally, this character is illegal in (user-defined) HLA symbols. You may, however, give an identifier a legal HLA name and then specify the VC++ compatible name within the string constant. For example, here is a typical procedure declaration found in the HLA standard library "fileio.hla" source file:

```
procedure WriteFile
```

1. However, since HLA emits the identifier to the MASM assembly language output file, the external identifier must be MASM compatible.

```
(
        overlapped:      dword;
    var bytesWritten:    dword;
        len:             dword;
    var buffer:          byte;
        Handle:          dword
);
    @external( "_WriteFile@20" );
```

(The "@20" suffix is a Win32 convention that indicates that there are 20 bytes of parameter data in this external function.)

As noted above, many languages "mangle" their external names for one reason or another. In addition to the "@20" suffix in the previous example, you will also note that VC++ added a leading underscore to the name (this procedure calls the Win32 API *WriteFile* function). Once again, this name mangling is a function of the particular compiler being used. Since Windows itself is written in VC++, Win32 API calls follow the VC++ standards for name mangling.

In addition to giving you the ability to conform external names as needed by external languages, the string parameter of the **external** directive will let you change the name for more mundane reasons. For example, if you really don't like the external name, perhaps it is not descriptive of the operation, you can use the string parameter feature of the external directive to allow the use of a different, perhaps more descriptive, name in your HLA code.

Some languages, for example C++, provide *function* overloading. This means that a program can use the same name to reference two completely different procedures in the code. Within the object file, however, all names must be unique. Once again, the compiler's name mangling facilities come into play to generate unique names. How a particular name is mangled is extremely compiler sensitive (e.g., Borland's C++ mangles names differently than Microsoft's Visual C++, even when compiling the same exact C++ program). When deciding on the name with which to reference an external procedure, you may need to consult your compiler documentation or be willing to experiment around a bit.

15.4 HLA Calling Conventions and Other Languages

Of course, HLA is an assembly language, so it is possible via the **push** and **call** instructions to mimic any calling sequence used by any language that allows the call of external assembly language code (which covers almost all languages). However, when using the HLA high level language features, in particular, HLA procedure declarations and calls, there are some details you must be aware of in order to successfully call code written in other languages or have those other languages call your code.

By default, HLA assumes that all parameters are pushed on the stack in a left-to-right order as the parameters appear in the formal parameter list. Some languages, like Pascal and Delphi, use this same calling mechanism. A few languages, most notably C/C++, push their parameters in the right-to-left order. If the language expects the parameters to be in the reverse order (right-to-left), a simple solution is to use the **@cdecl** or **@stdcall** procedure options to specify the calling convention.

Many languages, like HLA, Pascal, and Delphi, make it the procedure's responsibility to clear parameters from the stack when the procedure returns to the caller. Some languages, like C/C++ make it the caller's responsibility to clear parameters from the stack after the procedure returns to the caller. Procedures you declare with the **@pascal** and **@stdcall** procedure options automatically remove their parameter data from the stack when they return. Procedures you declare with the **@cdecl** option leave it up to the caller to remove the parameter data from the stack. Note that when using the HLA high-level procedure calling syntax, HLA automatically pushes the parameters on the stack in the correct order ("correct" as defined by the procedure's calling convention).

HLA procedures do not support a variable number of parameters in a parameter list. If you need this facility (e.g., to call a C/C++ function) then you will need to manually push the parameters on the stack yourself prior to calling the function. Procedures that have a variable number of parameters almost always using the **@cdecl** calling convention; since only the caller knows how much parameter data to remove from the stack, the procedure generally cannot remove the parameter data (as the **@pascal** and **@stdcall** conventions do).

15.5 Calling Procedures Written in a Different Language

When calling a subroutine written in a different language, your code must pass the parameters as the other language expects and clean up the parameters if the target language requires your code to do so upon return. Generally, calling code written in other languages is relatively easy. You have to ensure that you're passing the parameters in the proper places (e.g., in registers or pushing them on the stack in an appropriate order). Generally, such a call only requires that you provide a suitable external procedure declaration (e.g., swapping the order of the parameters in the parameter list if the language passes parameters in a right-to-left order). Some languages may require additional data structures (e.g., static links) to be passed. It is your responsibility to determine if such data is necessary and pass it to the subroutine you are calling.

15.6 Calling HLA Procedures From Another Language

Calling HLA procedures from another language is somewhat more complex that the converse operation. You still have the problem of parameter ordering; though this is usually fixed by reversing the parameters in the parameter list (e.g., using the **@cdecl** or **@stdcall** procedure options).

A bigger problem is the responsibility of cleaning up the parameters on the stack. By default, an HLA procedure automatically removes parameter data from the stack upon return. If the calling code thinks that it has the responsibility to do this cleanup, the parameter data will be removed twice, with disastrous results. Such code must use the **@cdecl** calling convention or you must use the **@noframe** option (and probably **@nodisplay** as well) to disable the automatic generation of procedure entry and exit code. Then you must manually write the code that sets up the activation record and returns from the procedure. Upon return, you must use the **ret**() instruction without a numeric parameter.

HLA external procedures must always be declared at lex level one. Since the condition of the stack is unknown upon entry into HLA code from some externally written code, your external HLA procedures should not depend upon the display to access non-local variables. HLA procedures that other languages call should always have the **@nodisplay** option associated with them. While it is okay to access non-local STATIC objects, you should never attempt to access non-local **var** objects from a procedure that code written in a different language will call.

HLA's **@pascal, @stdcall,** and **@cdecl** procedure options cover the calling conventions of most modern high level languages. However, other calling conventions do exist (for example, the METAWARE compilers give you an option of passing parameters in the left-to-right order and it is the caller's responsibility to clean up the stack afterwards). Some languages don't even pass their parameters on the stack. Some languages pass some or all of the parameters in registers. If you are linking your HLA code with a language that uses one of these non-standard calling conventions, it is your responsibility to write the explicit HLA code that passes these parameters and cleans up the parameter data upon return from the procedure.

15.7 Linking in Code Written in Other Languages

When linking in code written in a different language to an HLA main program, keep in mind that the foreign code may make calls to the standard library associated with the other language. You may need to link in that code as well. Also keep in mind that some compilers emit code that assumes that certain initialization has occurred when the program is loaded into memory. Unfortunately, if the main program is not written in this other language (i.e., main is written in HLA), this initialization might not have been done. This may very well cause the routine you're linking into an HLA program to fail.

Conversely, be very careful about calling HLA standard library routines in code you expect to link into programs written in other languages. The HLA standard library routines (and the exception handling code, in particular), rely upon initialization that the HLA main program performs. This could create a problem, for example, if you attempt to execute some procedure that raises an exception and the exception handling code has not been initialized.

15.8 Calling HLA Code From Other Languages

As explained earlier, calling HLA procedures and functions from other languages is generally easy. Just create an "external" procedure declaration (to make your procedure's name public), compile the procedure as part of a unit, link it with your other code, and you're in business. There is one catch, and I quote from the chapter on Mixed Language Programming from the first edition of "The Art of Assembly Language":

> *A large percentage of the HLA Standard Library routines include exception handling statements or call other routines that use exception handling statements. Unless you've set up the HLA exception handling subsystem properly, you should not call any HLA Standard Library Routines from non-HLA programs.*

Similarly, you should not use any exception handling statements in code that you call from non-HLA code unless you've properly set up the exception handling subsystem.

Until now, that advice has simply meant "*Don't use exceptions and don't call any routines that use exceptions (e.g., HLA Standard Library routines) when calling HLA procedures from a non-HLA main program.*" What is the reason for this tough restriction? Simple, other than myself and perhaps a few hearty programmers who've probed the internals of HLA-generated code, very few people have known how to set up the HLA exception handling system properly.

Properly setting up the HLA exception handling system isn't that complex. In fact, once you know what you're doing, it's actually quite easy. However, until now that knowledge hasn't been publically available, so the best advice has always been "don't even try it." The purpose of this section is to rectify this situation by describing what you need to do to initialize HLA's exception handling system.

Before going too much farther, I should point out that the information in this document is specific to Windows. While the same concepts apply to Linux, Mac OS X, and FreeBSD there are a few differences. If there is demand for such a thing, I'll be more than happy to create a document such as this one for those users. The principle differences have to do with the way x86 CPU exceptions are handled. The general HLA exception handling mechanism is the same under all OSes, it's just a question of how the HLA exception handling subsystem taps into the OS' exception system. If you're interested in seein a portable version of the following description, take a look at the source code for the HLABE (HLA back engine) code in the HLA compiler source files. HLABE is HLA code that a C/C++ program calls and it properly sets up the HLA run-time system when called for the HLA compiler.

When an HLA program first starts running, it executes a (compiler-generated) call to an HLA Standard Library procedure called *BuildExcepts*. *BuildExcepts* creates a Windows-compatible SEH (Structured Exception Handling) record in the main program's stack frame. This SEH record becomes the "catch-all" for any exceptions that the program doesn't specifically handle. Should an exception wind its way down to this particular exception handling record, then the code executes the programs default exception handler, that displays an error message and aborts the program.

The problem with calling HLA code from another language is that this default SEH record has never been built, because there is no HLA main program executing that built this record upon initial execution. When an unhandled exception comes along, the system generally crashes or hangs as there exists no default exception handler to deal with the exception. To avoid this problem (so you can use exceptions and call code that uses exceptions), what you have to do is manually build that SEH record yourself. Actually, you don't have to build the SEH record yourself - that's exactly what the HLA Standard Library *BuildExcepts* procedure does. What you have to do is call this procedure so it can build the SEH record for you.

In a normal HLA main program, an application calls *BuildExcepts* exactly once - immediately upon entry into the main program. This creates a single SEH exception handling record that sits around on the stack until the program exits. Unfortunately, when you call HLA code from some other language, you don't get the opportunity to build this SEH record at the beginning of the main program's execution (and even if you did, there is no guarantee that the exception handling system in place in that other language is compatible with HLA's). Therefore, we won't be able to build the SEH record once and forget about it; instead, we'll have to build the SEH record on each call to some HLA procedure from external code, and we'll have to tear down that SEH record before leaving. Yep, this is all overhead that you're going to execute on each call to an HLA function you make from some other language. The good news is that setting up (and tearing down) the SEH record takes less than a dozen instructions, so it's not that big of a deal.

Setting up and tearing down the SEH isn't the only work involved in supporting exceptions in HLA code. There are a couple of routines and a couple of data structures that the HLA compiler automatically generates whenever you write a main program. You'll have to manually supply these routines and data structures yourself.

The data structures exist to support HLA coroutines. Though it's unlikely you'll use coroutines in HLA code you call from C or some other language, you still have to create a coroutine data structure for the "main program" because the HLA exception handling code references this data structure. This is easily achieved with the following HLA code:

```
static
    MainPgmVMT:dword:= &QuitMain;

    // The following comprise the Main Program's coroutine data structure.

    MainPgmCoroutine: dword[ 5 ]; @external( "MainPgmCoroutine__hla_" );
    MainPgmCoroutine: dword; @nostorage;
                        dword &MainPgmVMT, 0, 0;
    SaveSEHPointer:dword; @nostorage;
                        dword 0, 0;
```

The important field in this structure is the *SaveSEHPointer* field. The exception handling system expects a pointer to the previous SEH record in this field. The *BuildExcepts* stores the old SEH pointer in this field, when your code returns it should restore the SEH pointer from this field. You can ignore the remaining fields in these two data structures; they just exist to keep HLA happy.

The HLA Standard Library provides three routines we'll need to reference in the exception handler code we're setting up. However, the HLA Standard Library header files don't provide prototypes for all of these routines (because it would be unusual for user code to call them), therefore, you'll also have to manually supply prototypes for these routines. The prototypes are

```
procedure BuildExcepts; @external("BuildExcepts__hla_");
procedure HardwareException; @external( "HardwareException__hla_" );
procedure DefaultExceptionHandler; @external(
"DefaultExceptionHandler__hla_" );
```

BuildExcepts we've already discussed. The *HardwareException* procedure is where the system would normally transfer control on a hardware exception. The *DefaultExceptionHandler* is the code that HLA jumps to whenever an exception occurs. The purpose behind these last two procedures is to allow the HLA compiler to link in a separate set of exception handling routines depending on whether you want a "compact" exception handler or the full exception handler (the difference has to do with the size of the string data that HLA would link in). Throughout this paper, we'll assume you want to link in the full exception-handling package. See the details in the HLA reference manual concerning exceptions (and look at the code HLA emits for short exceptions) if you're interested in linking in the shorter version of the exception handler (with a single generic message rather than exception-specific messages).

In addition to the Standard Library routines given above, the HLA compiler also writes a couple of procedures (and provides program termination code). These procedures take the following form:

```
procedure QuitMain;
begin QuitMain;

    ExitProcess( 1 );

end QuitMain;

procedure HWexcept;
begin HWexcept;

    jmp HardwareException;

end HWexcept;
```

```
procedure DfltExHndlr;
begin DfltExHndlr;

    jmp DefaultExceptionHandler;

end DfltExHndlr;
```

QuitMain, in the HLA generated code, is really just a label, not a full procedure. HLA transfers control to this label whenever it wants to terminate the program. As some exceptions will transfer control to this label, you must supply this label in your code. All this procedure's body need do is return control to the operating system. You can actually sneak in anything else you want, but when the procedure completes, it must return control to Windows (e.g., via the ExitProcess call).

The *HWexcept* label is where HLA's initialization code points the "hardware exception vector." Specifically, hardware exceptions like divide errors, segmentation faults, bounds violations, etc., first jump to this procedure. This short procedure simply passes control to the routine in the HLA Standard Library that actually handles the hardware exception.

DfltExHndlr is another procedure written by the HLA compiler. The purpose of this routine is to allow HLA code to link with the full exception handler (*DefaultExceptionHandler*) or the short exception handler (see the HLA standard library exception handling code for details). As noted earlier, in this paper we're going to use the full exception handling system.

To explain how to use all these functions and data types, an example is in order. Consider the following C program that will call an HLA procedure named *hlaFunc*:

```
/*
** A demonstration of how you can call HLA code
** that calls the HLA Standard Library from code
** that is not an HLA main program (in this case, it's
** a "C" program).
**
** Note: this program was compiled with Microsoft VC++
** using the following command lines:
**
**  c:>vcvars32
**  c:>hla -c hlafunc.hla
**  c:>cl cdemo.c hlafunc.obj hlalib.lib kernel32.lib user32.lib
*/

#include <stdio.h>

extern void hlaFunc( int value );

int
main( void )
{
    printf( "Calling HLA code\n" );
    hlaFunc( 10 );
    printf( "Returned from HLA code\n" );

    return 0;

}
```

As usual, we'll place the code we want to call from our C function in an HLA unit and compile this to an .OBJ file. Here's the complete HLA procedure (discussion to follow):

```
unit hlaFuncUnit;

// We want to demonstrate how to call HLA Standard Library
// routines from code that is called from C, so let's include
// the standard library right here.

#include( "stdlib.hhf" )

// Here's the sample function we're going to call from external
// code ("C" in this example) that demonstrates HLA stdlib calls
// and exception handling.

procedure hlaFunc( i:int32 ); @cdecl; @external( "_hlaFunc" );

// These are declarations for procedures that exist in the HLA
// standard library, but are "shrouded" in the sense that there
// aren't corresponding declarations in the stdlib.hhf file (these
// routines generally get called by HLA generated code, and nothing
// else; however, as we have to simulate "HLA generated code" here,
// we have to manually provide these declarations):

procedure BuildExcepts; @external("BuildExcepts__hla_");
procedure HardwareException; @external( "HardwareException__hla_" );
procedure DefaultExceptionHandler; @external(
"DefaultExceptionHandler__hla_" );

// The following are forward/external declarations for procedures
// that are normally created by the HLA compiler when you write
// a "main program." As we are not using an HLA main program here,
// we have to manually create these procedures.

procedure HWexcept; @external( "HWexcept__hla_" );
procedure DfltExHndlr; @external( "DfltExHndlr__hla_" );
procedure QuitMain; @external( "QuitMain__hla_" );

// The following is a Win32 API function this code calls:

procedure ExitProcess( rtnCode:dword ); @external( "_ExitProcess@4" );


// The following are some global, public, variables that the
// HLA exception handling run-time system expect the compiler
// to create for the HLA main program. Once again, as we are not
// writing an HLA main program here, we have to manually supply
// these objects:

static
   MainPgmVMT:dword:= &QuitMain;

   MainPgmCoroutine: dword[ 5 ]; @external( "MainPgmCoroutine__hla_" );
   MainPgmCoroutine: dword; @nostorage;
             dword &MainPgmVMT, 0, 0;
   SaveSEHPointer:dword; @nostorage;
             dword 0, 0;
```

```
// HLA main programs provide a "QuitMain" external label that
// exception handling code can when the exception causes the
// program to abort. This label immediately terminates program
// execution. As we are not writing an HLA main program, the HLA
// compiler does not provide this code for us, we have to supply
// it manually. You can do anything you want here, as long as you
// cause the *whole* program to terminate execution. This particular
// example simply calls ExitProcess and returns a termination code
// of one (which you can change to anything you want; non-zero usually
// indicates successful completion of the application, but this label
// normally gets called when the application aborts because of some
// exception, so returning zero isn't typical in this particular case.

procedure QuitMain;
begin QuitMain;

    ExitProcess( 1 );

end QuitMain;


// HWexcept is where the OS would normally transfer control
// when an x86 exception occurs. This procedure is normally
// written by the HLA compiler and simply jumps to an
// appropriate handler in the HLA Standard Library.

procedure HWexcept;
begin HWexcept;

    jmp HardwareException;

end HWexcept;

// DfltExHndlr is where the exception handling code transfers
// control when an HLA exception occurs. This is normally
// written by the compiler (to allow the compiler to choose
// between the full and short forms of the default exception
// handler). NOTE: the following code invokes the *full*
// exception handler (lots of meaningful messages, at the
// expense of the space needed for all those messages).

procedure DfltExHndlr;
begin DfltExHndlr;

    jmp DefaultExceptionHandler;

end DfltExHndlr;


// Here's the HLA code we're going to call from C that
// demonstrates exception handling without an HLA main program.

procedure hlaFunc( i:int32 );
var
    s:string;
    saveSEH:dword;
```

```
begin hlaFunc;

    // Before doing anything else, save a copy of the SEH pointer:
    #asm
        mov eax, fs:[0]
    #endasm
    mov( eax, saveSEH );

    // Upon entry into any HLA code that needs exception support,
    // we have to set up the structured exception handling record
    // for HLA:

    call BuildExcepts;

    // Because exception handling code can mess up all the registers,
    // we need to preserve EBX, ESI, and EDI across this call:

    push( esi );
    push( edi );
    push( ebx );

    // Okay, here's the code we're going to execute that uses
    // exceptions, calls HLA stdlib routines, etc., even though
    // caller is not an HLA program:

    try

        stdout.put( "stdout.put called from HLA code, i = ", i, nl );
        raise( 5 );

      exception( 5 );
        stdout.put( "Exception handled by HLA code" nl );

    endtry;


    // One more demonstration, this time with an exception
    // occurring deep down inside an HLA Standard Library routine:

    try
        stralloc( 16 );
        mov( eax, s );
        str.cpy( "Hello World", s );
        stdout.put( "Successfully copied 'Hello World' to s: ", s, nl );
        str.cpy( "0123456789abcdefghijklmnop", s );
        stdout.put( "Shouldn't get here" nl );

      anyexception

        stdout.put( "Exception code: ", eax, nl );
        ex.printExceptionError();

    endtry;
    strfree( s );
    stdout.put( "Returning to C code" nl );

    // Restore the registers we saved earlier:
```

```
    pop( ebx );
    pop( edi );
    pop( esi );

    // Restore the saved SEH value:

    mov( saveSEH, eax );
    #asm
        mov fs:[0], eax
    #endasm

end hlaFunc;

end hlaFuncUnit;
```

The *hlaFunc* procedure appearing at the end of this source file is of primary interest to us here. The HLA function you call from C (or any other language) must begin by immediately saving the SEH pointer and then calling *BuildExcepts* upon entry into the procedure. This constructs the HLA SEH record and initializes the HLA exception handling system. Just as important, before the procedure returns it must clean up the SEH record; this is accomplished with the last two **mov** instructions in this code (including the one appearing in the **#asm..#endasm** sequence). Everything between those two points is the normal body of your procedure. This code can use the **try..endtry** statement, raise exceptions, and call external procedures that using **try..end** and/or raise exceptions. The code appearing in this sample both demonstrates directly raising an exception and calling an HLA Standard Library routine that raises an exception. Also note how this code is free to call HLA Standard Library routines without fear of crashing the system should an exception occur.

It is important to realize that you must call *BuildExcepts* and clean up the SEH record in each HLA procedure you call from some other language. Note, however, that you don't have to do this for HLA procedures that you only call from HLA code (that has already built the SEH record).

15.9 Exercising Complete Control with HLA

Note: This section was written with Windows and Unix (Linux/FreeBSD/Mac OS X) programmers in mind. Most of the examples are Windows examples; this document provides Unix-specific examples only when there is a major difference in the way the compiler operates under Windows versus Unix.

A common complaint I get about HLA is that it "hides the machine from the user and generates tons of code behind the programmer's back." This is usually followed by something along the lines of "true assemblers don't do that." The truth is that the HLA compiler generates very little extraneous code and there is actually only a little bit of overhead in an HLA program. Part of the confusion stems from the fact that many users think of calls to the HLA Standard Library as part of "the HLA language." For example, many programmers who see the ubiquitous "Hello World" program written in HLA automatically assume that the "stdout.put" library call is part of the language and demonstrates the "bloat" that exists in HLA. Obviously, such a belief is erroneous since anyone could write their own I/O routines and replace the call to stdout.put with their code and HLA would be none the wiser.

However, to say that there is no code overhead or to say that HLA doesn't emit code behind the programmers back isn't true either. HLA was designed to make learning assembly language programming easy for beginners. Therefore, HLA does automatically generate some code to help beginners. Fortunately, it's easy to turn this extra code generation off and have HLA only generate code that you've written. The purpose of this document is to describe how to turn off all extraneous code generation so you, the advanced assembly programmer, can exercise absolute control over the machine code that HLA generates.

By the way, it is understood that if you intend to exercise absolute control over your machine code, you won't achieve this if you're using HLA's high-level control statements and certain other high level features that HLA provides. Fortunately, none of those high level features (that generate code behind your back) are necessary in an HLA program. You can easily avoid the extraneous code generation by simply not using those high-level control statements in your assembly programs. Since HLA allows you to write "pure assembly code" without any high level features, and there is nothing forcing you to use those statements, using high level control statements as an example of HLA's bloat is illogical. If you don't want such bloat in your assembly programs, don't use those statements!

15.9.1 Overhead Present in an HLA Program

Many people naturally assume that the HLA compiler introduces a lot of extra code into the assembly file it produces. They base their beliefs on several things including the sophistication of the HLA Standard Library (the HLA compiler must call some code to do some initialization required by the Standard Library, just like C), the sophistication of the data structures, and because of HLA's support for high-level control structures. This, however, is a misconception. Although the HLA compiler does emit some initialization code when it compiles an HLA program, this code is actually quite small; it's probably under a hundred bytes, not thousands of bytes or even hundreds of bytes. So let's get that misconception out of the way real fast; to prove this issue, we'll compile an empty HLA program and take a look at the MASM and Gas code it produces.

15.9.1.1 The "empty" Program

Conceptually, the simplest program we can write (and execute) is the empty program. The empty program compiles and runs, but just immediately returns to Windows without doing much of anything else. One would hope that the empty program would produce the smallest possible executable file size. Here's the empty program in HLA:

```
program t;
begin t;
end t;
```

The Canonical Empty Program

Here's the MASM code that (an early version of) HLA emitted (when using MASM as a back-end assembler) under Windows for the above program:

```
; Assembly code emitted by HLA compiler
; Version 2.0 build 485 (prototype)
; HLA compiler written by Randall Hyde
; MASM compatible output

  if @Version lt 612
  .586p
  else
  .686p
  .mmx
  .xmm
  endif
  .model flat, syscall
  option noscoped
  option casemap:none


offset32 equ <offset flat:>
```

```
  assume fs:nothing
ExceptionPtr__hla_ equ <(dword ptr fs:[0])>


  .code


        public     QuitMain__hla_
        public     DfltExHndlr__hla_
        public     _HLAMain
        public     HWexcept__hla_
        public     start
        externdef  shorthwExcept__hla_:near32
        externdef  abstract__hla_:near32
        externdef  BuildExcepts__hla_:near32
        externdef  Raise__hla_:near32
        externdef  shortDfltExcept__hla_:near32



  .data

        externdef  MainPgmCoroutine__hla_:byte
        externdef  __imp__MessageBoxA@16:dword
        externdef  __imp__ExitProcess@4:dword


  .code




HWexcept__hla_ proc near32
        jmp        shorthwExcept__hla_
HWexcept__hla_ endp

DfltExHndlr__hla_ proc near32
        jmp        shortDfltExcept__hla_
DfltExHndlr__hla_ endp



_HLAMain proc near32

start   proc near32
start   endp

        call       BuildExcepts__hla_
        pushd      0
        mov        ebp, esp
        push        ebp


QuitMain__hla_::
        pushd      0
        call       dword ptr __imp__ExitProcess@4
```

```
_HLAMain endp



                end
```

MASM Output Code for the Empty Program

Consider for a moment, the code appearing just before the main program (_HLAMain) in the assembly (MASM syntax) file:

```
HWexcept__hla_ proc near32
        jmp        shorthwExcept__hla_
HWexcept__hla_ endp

DfltExHndlr__hla_ proc near32
        jmp        shortDfltExcept__hla_
DfltExHndlr__hla_ endp
```

Obviously, these two jump instructions don't add much code to the executable, but they do jump to some external code, so it's fair to ask about the code associated with *shorthwExcept__hla_* and *shortDefltExcept__hla_* (these are two HLA Standard Library modules). These two procedures are actually quite small; their source code appears in the HLA Standard Library and is duplicated here:

```
unit shortHWexceptionUnit;
?@nodisplay := true;
?@noframe := true;

#macro fallsThrough( procName, externalName );

   procedure procName; @external( @string:externalName );
   procedure procName; begin procName; end procName;

#endmacro


fallsThrough( shw1, SHORTHWEXCEPT__HLA_ )
fallsThrough( shw2, shorthwExcept__hla_ )

procedure shortHWexcept;
begin shortHWexcept;
   mov( 1, eax );
   ret();
end shortHWexcept;

end shortHWexceptionUnit;
```

The shorthwExcept___hla_ Procedure

```
static
   messageBox:procedure
   (
      code:uns32;
      var title:var;
      var msg:var;
      n:uns32
   ); external( "__imp__MessageBoxA@16" );

   ExitProcess:procedure( code:uns32 );
      external( "__imp__ExitProcess@4" );


readonly
   DefaultMessage:byte; @nostorage;
      byte "Unhandled exception error.", 0;

   HLAException: byte; @nostorage;
      byte "HLA Exception Handler", 0;

#macro fallsThrough( procName, externalName );

   procedure procName; @external( @string:externalName );
   procedure procName; begin procName; end procName;

#endmacro



procedure defaultException; @external( "SHORTDFLTEXCEPT__HLA_" );

fallsThrough( defaultException2, shortDfltExcept__hla_ );
fallsThrough( defaultException3, _shortDfltExcept__hla_ );

procedure defaultException;
begin defaultException;

   messageBox( $30, HLAException, DefaultMessage, 0 );
   ExitProcess( 0 );

end defaultException;
```

The **shortDfltExcept__hla_** Procedure (Win32 version)

If you know anything about machine code, you'll probably realize quick that these procedures are very small. In fact, there are probably more bytes required for the two exception strings as the actual object code requires. Although I haven't actually counted the bytes, I'd guess that these two procedures and their data are well under 100 bytes, total.

Returning to the empty program, the main program (*_HLAMain*) for this file contains the following MASM code:

```
_HLAMain proc near32

start   proc near32
start   endp

        call       BuildExcepts__hla_
        pushd      0
        mov        ebp, esp
        push        ebp


QuitMain__hla_::
        pushd      0
        call       dword ptr __imp__ExitProcess@4
_HLAMain endp
```

The call to *BuildExcepts__hla_* and the three instructions that follow are the "overhead" associated with a typical HLA program. The last two instruction return control to the operating system; It's hard to call these two instructions overhead as every Windows program is going to need something like these two instructions (these would only be overhead if the program returns to Windows somewhere else and these last two instructions never execute).

The three instructions following the call above set up the stack frame for the main program. This provides access to the **var** objects found in the main program (there are none, or there would actually be another **sub** instruction present above). In some respect, these instructions are pure overhead since there are no automatic (**var**) objects in this program (and HLA sets up the stack frame in order to access automatic variables from the main program). However, we are talking about *three instructions* here that normally execute only once. I'm claiming that this isn't an incredible amount of bloat.

That leaves us with the call to the *BuildExcepts__hla_* procedure. This is another HLA Standard Library module that initializes HLA's exception handling system. Here's what the code to the *BuildExcepts__hla_* procedure looks like (for the non-threaded version, threaded code has additional overhead and we won't consider that here):

```
pop( eax );

// Fill in the MainPgmCoroutine data structure:

lea( ebx, MainPgmCoroutineVMT );
mov( ebx, MainPgmCoroutine.theCoroutineVMT );
mov( 0, MainPgmCoroutine.currentESP );
mov( 0, MainPgmCoroutine.stackPointer );
mov( 0, MainPgmCoroutine.pointerToLastCaller );

// Build an structured exception handler frame on the stack:

pushd( &DfltExHandlr );
push( ebp );
pushd( &MainPgmCoroutine );
pushd( &HWexcept );
```

```
          // push( fs:[0] );

          xor( ebx, ebx );
          fseg:push( (type dword [ebx]) );

          // mov( esp, fs:[0] );

          fseg:mov( esp, [ebx] );

          // We need to initialize the main program's coroutine object
          // with the pointer to the exception record we just created.
          // Note that we must initialize the ExceptionContext field
          // with this address:

          mov( esp, MainPgmCoroutine.exceptionContext );

          jmp( eax );
```

HLA Standard Library BuildExcepts__hla_ Procedure

Again, as you can see, there's not a whole lot of code here. The vast majority of this code simply initializes HLA's exception handing subsystem. You've just seen all the "bloated" code that HLA emits for most programs. You'll see a little bit later than it's even possible to remove all this code from an HLA output file (assuming you can live without exception support or are willing to write the code to support exceptions yourself).

15.9.2 The empty Program, Part II

Although the empty program of the previous section is the smallest program we can write that will compile and run, it's not the smallest program we can create with HLA, assuming we don't care if it doesn't run. The smallest possible program you can write with HLA would consist of a **unit** with a single procedure that has no instructions associated with it. The following is such an empty program:

```
unit empty;

 procedure main; @external( "_HLAMain" );
 procedure main; @noframe;
 begin main;
 end main;

end empty;
```

The "empty2" Program

To properly link and produce an .EXE file without error, an HLA program must have a procedure named *_HLAMain*. the external declaration above and the corresponding procedure declaration for main achieves this. Note the presence of the **@noframe** procedure option. This tells HLA to skip any extra code emission for the procedure. Here's a typical MASM file that the above produces when you tell HLA to emit a MASM-compatible assembly language source file:

```
; Assembly code emitted by HLA compiler
; Version 2.0 build 483 (prototype)
; HLA compiler written by Randall Hyde
; MASM compatible output

  if @Version lt 612
  .586p
  else
  .686p
  .mmx
  .xmm
  endif
  .model flat, syscall
  option noscoped
  option casemap:none


offset32 equ <offset flat:>

  assume fs:nothing
ExceptionPtr__hla_ equ <(dword ptr fs:[0])>






  .code


        public     _HLAMain
        externdef  HWexcept__hla_:near32
        externdef  abstract__hla_:near32
        externdef  Raise__hla_:near32
        externdef  shortDfltExcept__hla_:near32






  .data

        externdef  __imp__MessageBoxA@16:dword
        externdef  __imp__ExitProcess@4:dword




  .code

_HLAMain proc near32
_HLAMain endp
```

```
                    end
```

MASM Output File for the "empty2" Program

For this program, all of the include files are empty, so there's no need to list them here. If you compile this program to an executable, the resulting file is only 400-500 bytes long. This is because there is no code, so we don't need a 4K block associated with the code; there is no data, so we don't need a 4K block associated with the data segment; there is no Win32 API pointer data because we don't make any Win32 API calls. The PE/COFF header information still requires some memory, which is why the file is 400-500 bytes long.

15.9.3 Overhead Associated With Exceptions

As you saw earlier in the "empty" example, there is a bit of overhead associated with HLA's exception support. The empty program requires somewhere around 100 bytes of data and code to support exceptions. In fact, if you're sloppy or unaware, HLA's exception handling facilities can require quite a bit more overhead. Consider the following program:

```
program empty3;
#include( "stdlib.hhf" )
begin empty3;
end empty3;
```

The "empty3" Program

Here's the MASM code that the HLA compiler produces when you compile empty3 (specifying MASM output):

```
; Assembly code emitted by HLA compiler
; Version 2.0 build 483 (prototype)
; HLA compiler written by Randall Hyde
; MASM compatible output

  if @Version lt 612
  .586p
  else
  .686p
  .mmx
  .xmm
  endif
  .model flat, syscall
  option noscoped
  option casemap:none


offset32 equ <offset flat:>

  assume fs:nothing
```

```
ExceptionPtr__hla_ equu <(dword ptr fs:[0])>





  .code


        public     QuitMain__hla_
        public     DfltExHndlr__hla_
        public     _HLAMain
        public     HWexcept__hla_
        public     start
        externdef  DefaultExceptionHandler__hla_:near32
        externdef  abstract__hla_:near32
        externdef  HardwareException__hla_:near32
        externdef  BuildExcepts__hla_:near32
        externdef  Raise__hla_:near32
        externdef  shortDfltExcept__hla_:near32






  .data

        externdef  MainPgmCoroutine__hla_:byte
        externdef  __imp__MessageBoxA@16:dword
        externdef  __imp__ExitProcess@4:dword
        align      (4)




  .code




;/* HWexcept__hla_ gets called when Windows raises the exception. */

HWexcept__hla_ proc near32
        jmp        HardwareException__hla_
HWexcept__hla_ endp

DfltExHndlr__hla_ proc near32
        jmp        DefaultExceptionHandler__hla_
DfltExHndlr__hla_ endp



_HLAMain proc near32
```

```
start   proc near32
start   endp

        call       BuildExcepts__hla_
        pushd      0
        mov        ebp, esp
        push        ebp


QuitMain__hla_::
        pushd      0
        call       dword ptr __imp__ExitProcess@4
_HLAMain endp




                end
```

The "empty3.asm" Output File

You'll have to look close to see a difference between this MASM file and the one for the original "empty" program. Here are the lines that changed:

```
HWexcept__hla_  proc    near32
        jmp HardwareException__hla_
HWexcept__hla_  endp

DfltExHndlr__hla_ proc  near32
        jmp DefaultExceptionHandler
DfltExHndlr__hla_ endp
```

Here's the original code:

```
HWexcept__hla_  proc    near32
                jmp     shorthwExcept__hla_
HWexcept__hla_  endp

DfltExHndlr__hla_ proc  near32
                jmp     shortDfltExcept__hla_
DfltExHndlr__hla_ endp
```

The difference between the two is the standard library routines that they call. By the way, if you compile this code to an .EXE file, you'll discover that the .EXE file is exactly the same size as the original code: 10,732 bytes (with HLA v2.0). However, it turns out that there is over 3K of additional data in the empty3 version of this program. What is it that the #include("stdlib.hhf") has done to this code?

Well, the stdlib.hhf header file includes the excepts.hhf header file and the excepts.hhf header file assigns the value "true" to an HLA compile-time variable (**@exceptions)** that tells HLA whether you want the full exception handling system or an abbreviated version. When the HLA compiler encounters the begin clause associated with the main program, it checks the value of this compile-time variable. If it contains true, then HLA emits the *HWexcept__hla_* and *DfltExHndlr__hla_* procedures that transfer control to the full exception handler code (*HardwareException__hla_* and *DefaultExceptionHandler__hla_*). If the **@exceptions** compile-time variable contains **false** (the default value), then HLA emits these procedures with jumps to the shortened versions of these routines. Now the code for the full routines isn't a whole lot larger than the code for the short routines, the big difference is the amount of data. The short exception handler routines print a very short generic message (the same message for all exceptions) if they wind up being invoked. The full routines print a descriptive message that varies by the actual exception the system raises. Therefore, the full version of the exception handling code has this really big string array and all the data associated with that array is what consumes the better than 3K of additional space that the *empty3* program requires.

Since the **@exceptions** variable is a compile-time variable you can set during compilation, you can force HLA to use the shortened default exception handlers, even if you've included stdlib.hhf or excepts.hhf, by simply setting **@exceptions** to **false** prior to the begin clause of the main program, e.g.,

```
program empty3;
#include( "stdlib.hhf" )
?@exceptions := false;
begin empty3;
end empty3;
```

Program 2.11^: Modified 'empty3' Program That Uses the Short Exception Code

If you don't really need, or care about, informative exception handling in your code, and you're including the *excepts.hhf* header file (or some other header file that indirectly includes *excepts.hhf*, and this includes many of the Standard Library header files), then you can trim the size of your program down a bit by setting **@exceptions** to **false** prior to the **begin** clause of your main program. Note, however, that having nice descriptive messages is great when an exception actually occurs; so it's probably a good idea to use the full exception-handling package when you're testing and debugging your code. Then set **@exceptions** to **false** before creating your production code to shave 3K off the executable's size.

Note that you cannot trap any hardware exceptions (e.g., divide by zero) when using the short exception handler. If you want to be able to trap hardware exceptions but you don't want the overhead of the exception string messages you've got a couple of choices: (1) implement Windows structured exception handling yourself (difficult) or (2) grab the sources to the exception handling library code and remove all the message strings. Generally, 3K is such a small amount that it isn't worth the effort to try and shave this data from your code.

Later, this document will discuss the overhead associated with HLA's high-level control statements. But as long as we're on the subject of exceptions, it's probably worthwhile to explore the cost of the HLA **raise** and **try..endtry** statements. Here's a sample HLA program that exercises these statements and the corresponding MASM code:

```
program ExceptsDemo;
begin ExceptsDemo;

    #asm ;raise stmt #endasm
    raise( 1 );

    #asm ;try stmt #endasm

    try

        mov( 0, al );
```

```
      #asm ;unprotected stmt #endasm

      unprotected

        mov( 1, al );

      #asm ;exception( 1 ) stmt #endasm

      exception( 1 )

        mov( 2, al );

      #asm ;exception( 2 ) stmt #endasm

      exception( 2 )

        mov( 3, al );

      #asm ;anyexception stmt #endasm

      anyexception

        mov( 4, al );

      #asm ;endtry stmt #endasm

    endtry;
    mov( 5, al );

end ExceptsDemo;
```

Sample HLA Program to Demonstrate Exceptions

```
; Assembly code emitted by HLA compiler
; Version 2.0 build 483 (prototype)
; HLA compiler written by Randall Hyde
; MASM compatible output

  if @Version lt 612
  .586p
  else
  .686p
  .mmx
  .xmm
  endif
  .model flat, syscall
  option noscoped
  option casemap:none


offset32 equ <offset flat:>

  assume fs:nothing
```

```
ExceptionPtr__hla_ equ <(dword ptr fs:[0])>


  .code


        public     QuitMain__hla_
        public     DfltExHndlr__hla_
        public     _HLAMain
        public     HWexcept__hla_
        public     start
        externdef  shorthwExcept__hla_:near32
        externdef  abstract__hla_:near32
        externdef  BuildExcepts__hla_:near32
        externdef  Raise__hla_:near32
        externdef  shortDfltExcept__hla_:near32


exception__hla_5 equ Raise__hla_




  .data

        externdef  MainPgmCoroutine__hla_:byte
        externdef  __imp__MessageBoxA@16:dword
        externdef  __imp__ExitProcess@4:dword




  .code




HWexcept__hla_ proc near32
        jmp        shorthwExcept__hla_
HWexcept__hla_ endp

DfltExHndlr__hla_ proc near32
        jmp        shortDfltExcept__hla_
DfltExHndlr__hla_ endp



_HLAMain proc near32

start   proc near32
start   endp

        call       BuildExcepts__hla_
        pushd      0
        mov        ebp, esp
        push        ebp
```

```
                ;raise stmt
        mov       eax, 1
        jmp       Raise__hla_

                ;try stmt
        pushd     offset32 exception__hla_2
        push       ebp
        db        064h
        mov       ebp, ds:[0]
        push       dword ptr [ebp+8]
        mov       ebp, [esp+4]
        pushd     offset32 HWexcept__hla_
        db        064h
        push       dword ptr ds:[0]
        db        064h
        mov       ds:[0], esp
        mov       al, 0

                ;unprotected stmt
        db        064h
        mov       esp, ds:[0]
        db        064h
        pop       dword ptr ds:[0]
        add       esp, 8
        pop       ebp
        add       esp, 4
        mov       al, 1

                ;exception( 1 ) stmt
        jmp       endtry__hla_1
exception__hla_2:
        cmp       eax, 1
        jne       exception__hla_3
        mov       al, 2

                ;exception( 2 ) stmt
        jmp       endtry__hla_1
exception__hla_3:
        cmp       eax, 2
        jne       exception__hla_4
        mov       al, 3

                ;anyexception stmt
        jmp       endtry__hla_1
exception__hla_4:
        mov       al, 4

                ;endtry stmt
endtry__hla_1:
        mov       al, 5
QuitMain__hla_::
        pushd     0
        call      dword ptr __imp__ExitProcess@4
_HLAMain endp
```

```
                end
```

MASM Output File From the Exceptions Source

The purpose of this document is not to explain how structured exception handling under Windows works (upon which HLA's exception handlers are based). Therefore, I'm not going to bother explaining what any of the statements mean in the code above. Instead, the important thing is to note the amount of code that each statement or clause produces.

The **raise** statement is simple. It loads the value of its argument into EAX and then transfers control to the *Raise__hla_* standard library procedure (see the standard library sources if you're interested, it is a fairly short routine, though). As you can see, the **raise** statement doesn't generate a whole lot of code.

The **try..endtry** statement is at the other extreme. This statement probably generates more code than any other single high-level control statement that HLA provides[1]. To get an idea of the amount of code generated for each clause, note that I've used the **#asm..#endasm** directive to inject comments into the MASM output file and I've used instructions of the form "mov(const, al);" to help delineate the code that HLA produces for each of the **try..endtry** clauses.

The **try..endtry** statement is very powerful and provides a sophisticated solution to the problem of exception handling. However, as you can see, the **try..endtry** sequence generates quite a bit of code (not a tremendous amount, but it add up if you place a lot of **try..endtry** statements in your program). If you're trying to write code that is as fast and as short as possible, you may produce better quality code by simply returning an error status from your procedures and functions rather than raising exceptions in those functions and relying on a **try..endtry** block to catch the exception. There is no guarantee that the explicit return value approach is faster or shorter, but it usually is.shorter and faster (though it's nowhere near as convenient as **raise/try..endtry** and far more error prone). Just something to keep in mind.

15.9.4 Overhead Associated with Procedures, Iterators, and Methods

HLA was designed as a tool to teach assembly language programming to absolute beginners. Therefore, it does a couple of things that, by default, make it easier on beginners but may produce some excess code that an advanced assembly programmer would never write. One place where this is especially true is in the declaration and invocation of HLA procedures. Fortunately, HLA provides many options that let you control the extra code it emits for beginners (including turning off the code generation). This section explores the options you can use to control code generation for procedures and calls to procedures[2].

By default, HLA automatically generates code at the beginning of a procedure to construct the activation record for that procedure, align the stack to a double-word boundary, allocate local variables, and build a display for that procedure[3]. HLA also automatically generates the code to clean up the activation record and return from the procedure (and for the **@stdcall** and **@pascal** calling sequences, this code also cleans up the parameters on the stack). Sometimes this code is unnecessary (e.g., the procedure doesn't have any stack-based parameters or local variables),

1. Technically speaking, this is not true. Using the conjunction(&&) and disjunction (||) operators, you can generate some really large if, while, etc., statements. However, anyone who creates a really huge boolean expression is going to expect a bit of code bloat).

2. This section will use the generic term "procedures" to mean any HLA procedure, iterator, or method, unless otherwise noted.

3. Displays are advanced data structures that provide access to non-local automatic variables.

slightly less than efficient (e.g., you can access all the parameters and locals off ESP and you don't need to set up a stack frame with EBP), or you want to do things a litttle differently for some specific reason. Obviously, in these situations, HLA's default behavior is not what you want. Fortunately, HLA makes it easy to modify it's behavior for a specific procedure or even change the overall default behavior.

To begin with, it's probably a good idea to look at the code HLA automatically generates for a procedure. We'll use the following example repeatedly with slight modifications in this section:

```
program ProcDemo;

    procedure demo( b:byte; w:word; d:dword; var refvar:dword );
    var
        localVar:   dword;

    begin demo;

        nop();

    end demo;

begin ProcDemo;
end ProcDemo;
```

The Generic HLA Procedure

Here's the MASM assembly output for the demo procedure above (for the sake of brevity, I'll not put the whole MASM output file here - it's roughly the same code you'll find in the empty examples):

```
demo__hla_1 proc near32
        push        ebp
        push        dword ptr [ebp-4]
;/*Get frame ptr*/
        lea        ebp, [esp+4]
        push        ebp
        sub        esp, 4
        and        esp, -4
        nop
xdemo__hla_1__hla_:
        mov        esp, ebp
        pop        ebp
        ret        16
demo__hla_1 endp
```

HLA Code Generation (MASM assembly output) for the 'demo' Procedure

Notice that the original procedure only had one instruction (a **nop**). HLA actually generates nine additional instructions inside this procedure. While some of them (e.g., the **ret** instruction) would have to be present, some fat here can be trimmed, depending on your circumstances.

The first thing that you can usually trim away is the generation of the code that builds the display. This is the second through fourth instructions above (**push, lea, push**). Displays are a special data structure that provides access to non-local automatic variables in nested procedures. 99% of the time (or better), most assembly procedures won't need a display. That's because 98% of all assembly language programmers will never nest their procedures and the 2% that do can often pull other tricks to access non-local variables without using a display. Therefore, the vast majority of the time you can eliminate these statements that set up the display from the procedure code. This is easily accomplished by supplying the **@nodisplay** procedure option, e.g.,

```
program ProcDemo;

    procedure demo( b:byte; w:word; d:dword; var refvar:dword );
@nodisplay;
    var
        localVar:   dword;

    begin demo;

        nop();

    end demo;

begin ProcDemo;
end ProcDemo;
```

HLA Demo Program with @nodisplay Option

Here's the corresponding code that HLA emits for the program above:

```
demo__hla_1 proc near32
        push        ebp
        mov        ebp, esp
        sub        esp, 4
        and        esp, -4
        nop
xdemo__hla_1__hla_:
        mov        esp, ebp
        pop        ebp
        ret        16
demo__hla_1 endp
```

HLA Code Generation for Demo With @nodisplay Option

Well, this code looks a whole lot closer to a procedure with a standard entry/exit sequence. About the only surprising piece of code here is the **and** instruction. HLA automatically emits this code to guarantee that the stack is aligned upon a four-byte boundary upon entering the procedure. If the caller has misaligned the value in ESP such that it is not an even multiple of four, certain system calls may fail. The **and** instruction above ensures that ESP is double-word aligned. Unless you mess with ESP's value (or push word values on the stack), ESP is always double-word aligned. Note that this is true even if you specify some number of local variables whose aggregate size is not an even multiple of four (the sub instruction above reduces ESP by the number of bytes of local variables present, but HLA always rounds this value up to the next even multiple of four to keep ESP double-word aligned). If you know that ESP is double-word aligned (because you've not messed with the stack pointer), then the and instruction in the code above is superfluous. You may eliminate this extra instruction by specifying the **@noalignstack** procedure option:

```
 procedure demo( b:byte; w:word; d:dword; var refvar:dword );
   @nodisplay;
   @noalignstack;

    var
        localVar:   dword;

    begin demo;

        nop();

    end demo;

begin ProcDemo;
end ProcDemo;
```

HLA Demo Code With @noalignstack Option

Here's the corresponding code that HLA emits for the program above:

```
demo__hla_1 proc near32
        push        ebp
        mov        ebp, esp
        sub        esp, 4
        nop
xdemo__hla_1__hla_:
        mov        esp, ebp
        pop        ebp
        ret        16
demo__hla_1 endp
```

HLA Code Generation (MASM syntax) for Demo With @nodisplay Option

Now we've gotten down to the point where the code looks just like the standard entry/exit sequence you'd expect for a procedure. Of course, we could make some changes still. For example, the 80x86 CPU family supports two instructions, **enter** and **leave**, that you may use to build and destroy activation records (including displays, if necessary). While these instructions are typically slower than the discrete instructions that do the same job, they are certainly shorter and, therefore, some programmers prefer to use them. By default, HLA generates discrete instructions to build and destroy activation records. However, by using the **@enter** and **@leave** procedure options, you can tell HLA to use these instructions rather than the discrete instruction sequences:

```
 procedure demo( b:byte; w:word; d:dword; var refvar:dword );
   @nodisplay;
   @noalignstack;
   @enter;
   @leave;

    var
        localVar:   dword;

    begin demo;

        nop();

    end demo;

begin ProcDemo;
end ProcDemo;
```

HLA Demo Code With @enter and @leave Options

Here’s the corresponding code that HLA emits for the program above:

```
demo__hla_1 proc near32
        enter      0, 4
        nop
xdemo__hla_1__hla_:
        leave
        ret        16
demo__hla_1 endp
```

HLA Code Generation for Demo With @enter and @leave Options

As you can see, this procedure is starting to become seriously shortened. HLA is emitting only three extra instructions (down from the original nine or so).

Of course, ’real’ assembly language programmers want to write all their own code. If HLA is automatically generating anything for them, no matter how convenient, they’re going to complain. Well, HLA provides the **@noframe** procedure option that eliminates all code generation other than the explicit machine instructions the programmer provides. Note that supplying **@noframe** implicitly supplies **@noalignstack** and, to a certain extent, **@nodisplay** since **@noframe** turns off all extra code generation in a procedure[1]. Here are the examples above specifying **@noframe**:

```
 procedure demo( b:byte; w:word; d:dword; var refvar:dword );
```

1. Note, however, that if @noframe is not present, HLA will still assume you want to allocate storage for a display and will consider this fact when assigning offsets to local variables found in the procedure. Therefore, it’s a good idea to go ahead and specify @nodisplay along with @noframe.

```
    @nodisplay; // Still should be here, see footnote
    @noframe;

     var
         localVar:   dword;

     begin demo;

         nop();

     end demo;

begin ProcDemo;
end ProcDemo;
```

HLA Demo Code With @noframe Option

Here’s the corresponding code that HLA emits for the program above:

```
demo__hla_1 proc near32
        nop
demo__hla_1 endp
```

HLA Code Generation for Demo With @nodisplay and @noframe Options

Now, however, we have a problem. There is no **ret** instruction to return from this procedure. But that’s okay, the "macho" assembly programmer who doesn’t want HLA generating any code for them surely wants the program to fall through this procedure to the next instruction in memory, or they wouldn’t have left out the **ret** instruction in the original code. Here’s what the procedure would normally look like when the **@noframe** option is present:

```
 procedure demo( b:byte; w:word; d:dword; var refvar:dword );
    @nodisplay; // Still should be here, see footnote
    @noframe;

     var
         localVar:   dword;

     begin demo;

         nop();
         ret( 16 );

     end demo;

begin ProcDemo;
end ProcDemo;
```

HLA Demo Code With @noframe and @nodisplay Options (Part II)

For those who want to write all their own code and not have HLA generate anything extra code in their procedures, constantly attaching **@noframe** and **@nodisplay** to every procedure declaration can get old, fast. Fortunately, HLA provides a mechanism that lets you set the default state for all of these procedure options.

As shipped, HLA defaults to the following options: **@frame, @display, @alignstack, @noenter,** and **@noleave**. You can change the defaults by using these options as compile-time variables and setting them to true or false. Here are the possible options:

: Procedure Options and Their Effect on Code Generation

Option	Effect if set to true	Effect if set to false
@enter	HLA generates ENTER instruction to build activation records upon procedure entry. Note that **@frame** must also be true for HLA to emit this code.	HLA generates discrete instructions to build activation records upon procedure entry. Note that **@frame** must also be true for HLA to emit this code.
@noenter	Same as setting **@enter** to false.	Same as setting **@enter** to true.
@leave	HLA emits the **leave** instruction to clean up the activation record upon exit. Note that **@frame** must also be true for HLA to emit this code.	HLA emits discrete instructions (**mov**, **pop**) to clean up the activation record upon exit. Note that **@frame** must also be true for HLA to emit this code.
@noleave	Same as setting **@leave** to false.	Same as setting **@leave** to true.
@display	HLA emits instructions that allocate storage for and initialize a display structure. If **@enter** is true, HLA emits an **enter** instruction to accomplish this, otherwise it emits discrete instructions. Note that **@frame** must also be true for HLA to emit this code.	HLA does not emit any instructions that allocate or initialize the display structure.
@nodisplay	Same as setting **@display** false.	Same as setting **@display** true
@alignstack	HLA emits an **and** instruction that guarantees ESP is double-word aligned after allocating local variables. Note that **@frame** must also be true for HLA to emit this code.	HLA does not emit the **and** instruction that double-word aligns ESP.
@noalignstack	Same as setting **@alignstack** to false.	Same as setting **@alignstack** to true.
@frame	HLA generates code to construct the stack frame and other duties (e.g., align the stack if **@alignstack** is true, build the display if **@display** is true).	HLA does not generate any extra code for the procedure. It is the programmer's responsibility to write any necessary code to build the stack frame, if required.
@noframe	Same as setting **@frame** to false	Same as setting **@frame** to true.

The "macho" assembly language programmer will probably include the following two statements at the beginning of every HLA program they write:

```
?@noframe := true;
?@nodisplay := true;
```

The inclusion of these two statements tells HLA that the programmer is responsible for writing all the code that appears within the source file. Note that you may re-enable display and frame generation on a procedure-by-procedure basis by using the **@frame** and **@display** procedure options. See the discussion of procedure options in the HLA reference manual for more details.

Note that HLA still makes parameter names and local variable names available to your procedures when you specify the **@noframe** option. However, the offsets associated with these variables assume that you've built a standard stack frame and that you're going to reference the objects off EBP. If this is not the case, then you should not use the parameter and local variable names in your code; you'll have to use numeric offsets (say, from ESP) or, better yet, create TEXT constants that provide the necessary offsets from ESP, e.g.,

```
program ProcDemo;

?@noframe := true;
?@nodisplay := true;


    procedure demo( _d:dword; var _refvar:dword );
    const
        d       :text := "(type dword [esp+12])";
        refvar  :text := "(type dword [esp+8])";
        localvar:text := "(type dword [esp])";
    begin demo;

        pushd( 0 );    // Allocate _localVar and initialize to zero.
        mov( d, eax );
        mov( eax, localvar );
        mov( refvar, ebx );
        mov( eax, [eax] );
        add( 4, esp ); // Remove localvar from stack.
        ret( 8 );      // Return and pop parameters

    end demo;

begin ProcDemo;
end ProcDemo;
```

Using TEXT Constants to Access Parameters and Local Variables

```
demo__hla_1 proc near32
        pushd      0
        mov        eax, dword ptr [esp+12]
        mov        dword ptr [esp], eax
        mov        ebx, dword ptr [esp+8]
        mov        [eax], eax
        add        esp, 4
        ret        8
demo__hla_1 endp
```

Code Generation for the Above HLA Procedure (MASM syntax output)

15.9.5 Overhead Associated with Procedure Calls

As long as you manually pass the parameters yourself and use the **call** instruction, HLA does not inject any extra instructions into your code. However, if you use HLA's high-level procedure call syntax, HLA may very well emit some extra instructions into the code stream. If this bothers you, well, don't use the high level calling syntax - stick with the manual ("pure assembly") calling syntax.

However, the high level calling syntax is very convenient, it is far more readable and maintainable, and most of the time it generates exactly the same code you're going to write by hand. Therefore, it makes sense to use it as often as you can and understand the degenerate cases (where HLA emits some bad code) so you can code those by hand when efficiency is a prime concern.

First, HLA does a great job with "pass by value" parameters when the size of the value is four, eight, or 16 bytes. Such parameters generally require only a single instruction per double word to push on the stack prior to the call[1]. As the objects get larger, passing them by value gets very expensive. At some point, HLA doesn't bother trying to push the data on the stack, instead, it uses a **movsd** instruction to copy the data onto the stack. The following code shows what happens when you try to pass a 256-byte variable by value:

```
program ProcDemo;

type
    b256:byte[256];

    procedure demo( b:b256 );
    begin demo;
    end demo;

static
    c:b256;

begin ProcDemo;

    demo( c );

end ProcDemo;
```

Code That Passes a 256-byte Array by Value

```
         lea        esp, [esp-256]
        push        esi
        push        edi
        push        ecx
        pushfd
```

1. Assuming of course, you're passing the parameters on the stack and not ina register.

```
        cld
        lea         esi, c__hla_2
        mov         ecx, 64
        lea         edi, [esp+16]
        rep movsd
        popfd
        pop         ecx
        pop         edi
        pop         esi
        call        demo__hla_1
```

MASM Code HLA Emits for the Call to 'demo' Above

This isn't an example of HLA generating bloated code. HLA is doing a reasonable job given the request of the source code. However, HLA makes it so easy to write code that blows up like this that you can often make a mistake and pass a large data structure by value, causing HLA to generate a fair amount of slowing executing code. Actually, once you get above 64 bytes, HLA usually generates a sequence like the one above (with possibly one or two additional instructions if the object's size is not an even multiple of four bytes. So the size won't change too much as the object gets larger, but the execution time required by the **rep movsd** instruction goes up linearly with the size of the object. Moral of the story: unless there are good semantic reasons for doing so, always pass large objects by reference rather than by value. Watch out for this, because HLA will gladly emit the code to pass it by value without complaining.

Note that for parameters up to 64 bytes in size, HLA will actually emit a series of discrete push instructions. For parameters that are 16 bytes or less, this is no big deal (it only takes four push instructions to pass a 16-bit parameter by value). However, it's going to take 16 push instructions to pass a single 64-bit parameter by value to a procedure. This can cause some serious code bloat if you're doing this a lot. Moral: same as before, pass large objects by reference rather than by value (large is probably anything greater than 16 bytes in size).

HLA can go through some real gymnastics attempting to pass small parameters by value, as well. Because most modern (32-bit) operating systems always expect the stack to be double-word aligned, HLA (like most languages and OS API functions) always passes a parameter using a multiple of four bytes to hold that value. So if you're passing an object that's one, two, or three bytes in size, HLA will pass four bytes as the actual parameter. The procedure (generally, this is actually up to the programmer) ignores the extra bytes. This creates a problem when attempting to pass certain parameters on the stack; HLA solves these problems at the expense of greater code. Consider the following HLA program that has a one byte parameter and calls the procedure several different ways:

```
program smallParmDemo;

procedure byteParm( b:byte );
begin byteParm;
end byteParm;

static
    b:byte;

begin smallParmDemo;

    byteParm( b );
    byteParm( al );
    byteParm( ah );
    byteParm( (type byte [eax]) );
```

```
end smallParmDemo;
```

Procedure with a One-Byte Parameter

Here's the MASM code HLA generates for each of the calls to *byteParm*:

```
        pushd       0
        push         eax
        mov         al, b__hla_2
        mov         [esp+4], al
        pop         eax
        call        byteParm__hla_1
        push         eax
        call        byteParm__hla_1
        sub         esp, 4
        mov         [esp], ah
        call        byteParm__hla_1
        pushd       0
        push         eax
        mov         al, byte ptr [eax]
        mov         [esp+4], al
        pop         eax
        call        byteParm__hla_1
```

MASM Code HLA Generates for the Calls to byteParm

Many of these calls have an incredible amount of bloat! Any mediocre assembly programmer can probably do a better job than this! Why is HLA so bad? The reason HLA generates some ugly code here is because HLA makes a promise that it won't change any register values when passing parameters to a procedure (just in case you're passing some additional parameters in some registers). This promise severely impacts HLA's options when it comes to copying parameter data to the stack[1]. Indeed, about the only option HLA has when it needs a register is to preserve that register's contents while copying the parameter data. Consider the first call to *byteParm* above (passing the byte variable b). HLA first makes room for *b* on the stack by pushing a double word zero value. The HLA emits code to push the value of EAX, copy *b's* value into AL, store AL into the stack location allocated earlier, and then restore EAX's original value.

Now the clever assembly programmer might claim that this could be done far more efficiently with a single instruction, as follows:

```
    push( (type dword b) );
```

99.999% of the time, that programmer would be right; this is a much better way to pass a single byte parameter in a dword slot on the stack (this instruction pushes the value of the three bytes the follow *b* in memory, but since the procedure will ignore those three bytes anyway, who cares?). Unfortunately, this trick fails spectacularly in one very special (and, admittedly, rare) case. Consider what happens when *b* is allocated as the 4096th byte in a page and the next page in memory is not read-enabled. This is cause the program to crash. Granted, it's incredibly unlikely

1. It is interesting to note that MASM does not make this same promise. It will happily wipe out the EAX register if it needs a scratch-pad register while passing parameter data to a procedure via the INVOKE statement. I like to believe that HLA is a bit more "civilized" in this regard.

that this will ever happen in an HLA program. However, HLA's design can't assume that it won't ever happen. So HLA has to generate safe, but ugly, code.

Of course, there's nothing preventing you from recognizing this problem and manually pushing *b's* value as a dword yourself. E.g., either of the following will work:

```
    push( (type dword b) );
    call byteParm;

-or-

    byteParm( #{ push( (type dword b) ); }# );
```

As long as you can ensure that there are three reasonable bytes following b, this scheme is quite a bit more efficient than the default code HLA generates.

The second and third calls to *byteParm* in the example above are the ones where HLA actually generates halfway decent code. If the byte parameter falls in the L.O. byte of a 32-bit register, HLA will simply push the contents of that 32-bit register onto the stack. You aren't going to do any better than this (short of passing the parameter in a register, rather than on the stack). The second call, passing the byte parameter in AH (or any other byte register that is not the L.O. byte of a 32-bit register) needs two instructions: one to allocate storage on the stack (**push**) and another to copy the register's value onto the stack. An expert assembly language programmer, if they know they've got a register to play around with, can, perhaps, generate slightly better code by copying the 8-bit value to the L.O. byte of that register and then pushing the full register, e.g.,

```
        mov( al, bl );
        push( ebx );
```

This sequence is slightly shorter, though probably not any faster, than the code that HLA generates.

The fourth example above is really just a special case of the first example. If you look at the two code sequences, you'll notice that they are equivalent.

HLA generates less than stellar code for some of these sequences because it assumes that all registers are in use and it shouldn't modify any register values. Obviously, this is not always the case when you're calling a procedure. However, it's a rather difficult problem for HLA to automatically determine if there is a free register available that it can use while passing parameters. Fortunately, HLA provides a way for you to tell it that it can freely use one register (which is all it needs) for processing parameters: the **@use** ***reg*** procedure option. Consider the following modification of the previous program:

```
program smallParmDemo;

procedure byteParm( b:byte ); @use ebx;
begin byteParm;
end byteParm;

static
    b:byte;

begin smallParmDemo;

    byteParm( b );
    byteParm( al );
    byteParm( ah );
    byteParm( (type byte [eax]) );

end smallParmDemo;
```

byteParm and @use ebx

The **@use ebx** option tells HLA that it can freely use the EBX, BX, BL, and BH registers when generating the code to pass parameters to this procedure. Here's the (MASM-syntax) code HLA generates when you allow it to use the EBX register in this capacity:

```
mov       bl, b__hla_2
push      ebx
call      byteParm__hla_1
push      eax
call      byteParm__hla_1
sub       esp, 4
mov       [esp], ah
call      byteParm__hla_1
mov       bl, byte ptr [eax]
push      ebx
call      byteParm__hla_1
```

HLA Generated Code for the Above Calls to byteParm

As you can see, the code is much better than before (not quite as good since it still doesn't assume it can push b directly onto the stack, but much better nonetheless). Of course, if you want to take absolute control, you can always push the parameter manually.

Of course, the stack isn't the most efficient place to pass parameters. The x86 registers are the best place to pass parameters (subject to the constraint that they fit in the registers). Note that HLA will allow you to pass parameters in register using a high level calling syntax as follows:

```
procedure parmsInRegs( a:dword in eax; var b:byte in ebx );
    .
    .
    .
```

There's nothing stopping you in HLA from simply loading a register with some value prior to a call and referencing that register inside the procedure without declaring any formal parameters. The nice thing about using the high level declaration and calling syntax is that HLA will automatically move the value into the register for you if you specify an actual parameter other than the register for that parameter. However, since there's not much in the way of bloat here, there's really no sense in discussing it farther in this document. See the HLA reference manual for more details.

Reference parameters have their own special problems. As long as you're passing a non-indexed static address (that is, the address of a **static**, **readonly**, or **storage**) object by reference, HLA generates good code (a single push instruction). However, once you throw in an index register or specify an automatic variable (whose offsets are indexed off EBP), HLA has to emit an LEA instruction to compute the effective address of the operand. Since the **lea** instruction requires a register, we're back to the same problem we had with the byte-sized operand earlier. Well, the solution is the same: if you want decent code, either pass the address manually or specify an @use procedure option to tell HLA that it can use a register for computing effective addresses.

HLA supports several other parameter passing mechanisms. This document won't cover them for two reasons: (1) 99% of the assembly language programmers out there have probably never heard of these parameter passing mechanisms, and (2) the 1% of them who have, know that they're usually inefficient anyway (and fast/short code avoids them like the plague).

15.9.6 Bloat in the HLA Standard Library

If you want to understand the purpose of every byte in your HLA programs you don't call HLA Standard Library routines. It's not that they're incredibly poorly written, but they're "black boxes" and unless you sit down and study their source code, you have no idea what (private) data they declare, what routines they call, or anything else about their efficiency.

The HLA Standard Library routines were not written to be the fastest nor the shortest examples of HLA code. They were written to be easy to read, understand, and maintain. Furthermore, many of the routines build upon other routines. A classic example is the *stdout.puti8* routine. This procedure takes a single byte parameter. It calls the *conv.i8ToStr* procedure to convert the value to a string, and then calls the *fileio.puts* function to actually print the string (specifying the standard output file handle as the "file"). The *conv.i8ToStr* function zero extends the eight-bit value to 16 bits and calls the *conv.i16ToStr* function. The *conv.i16ToStr* function zero extends its 16-bit value to 32 bits and calls the *conv.i32ToStr* function. The *conv.i32ToStr* function converts its 32-bit value (include 24 bits of zeros at this point) to a string and the chain of calls pass the string back to the original call from *stdout.puti8*. Each of these routines (except *conv.i32ToStr*, which does all the real work) is very short and trivial. If you program winds up calling all of these routines, this is probably the most compact representation you could devise. However, this obviously requires a lot more code than had the standard library simply provided a *conv.i8ToStr* function that did the conversion directly. Furthermore, all those extra calls, plus the fact that converting a 32-bit value to a string is more expensive than converting an eight-bit value to a string, means the code is going to run a bit slower. Therefore, if speed and/or space are prime considerations in your program, avoid the HLA Standard Library (or, always start with the source code to the routine you want to call and clean it up so avoid long call chains like the one in the above example).

There is another source of bloat that is indirectly related to the HLA Standard Library. The HLA Standard Library was modeled after the standard libraries found in C, C++, and other high level languages. As a result, calling these library routines causes you to "think" like a C programmer. As any expert assembly programmer can tell you, "thinking in assembly" is the only way to write efficient assembly programs. Even if all the routines in the HLA Standard Library were written as efficiently as possible, the mindset they leave you in is not conducive to writing efficient code. Therefore, take care when using the HLA Standard Library because it can cause you to write sloppy code if you're not carefully considering what you're doing at each step in your code.

15.9.7 Taking Control with HLA Units

Reading the HLA reference manual, you might get the impression that HLA applications are written as **programs** and separately compiled modules that you link with HLA or applications in other languages are written using **units**. HLA units are actually a bit more flexible than this, if you're willing to play some games. In particular, HLA units can completely free you from the yoke of HLA compiler-generated code and give you an environment where the only instructions that appear in your executable file are those instructions you write. This section will describe how to use HLA units to achieve this.

Fundamentally, there are only a couple of differences between HLA units and HLA programs. HLA programs allow you to declare automatic variables in a global **var** section, units do not[1]. The major difference, of course, is that HLA units don't have a "main program" associated with them as HLA programs do. If you look at the code that HLA generates for units and programs, you see only a couple of differences between the output files. Specifically, HLA collects all the code from the main program and creates a procedure named *_HLAMain* (_start). In addition, HLA emits some support code to initialize the exception handling system for programs, none of this code appears in the assembly output file for a unit. Other than these two issues, HLA units and programs are semantically equivalent.

To prove this point, the following is an HLA unit that compiles to the same exact code as the standard Hello World program.

```
unit unitAsPgm;
#include( "stdout.hhf" )
```

1. Which makes sense because VAR objects are always associated with a procedure or the HLA main program. In a unit, there is no main program with which you can associate automatic variables.

```
?@nodisplay := true;
?@noframe := true;

// Make these names public so the library routines
// and linker can find them.

procedure _HLAMain; @external;
procedure HWexcept__hla_; @external;
procedure DfltExHndlr__hla_; @external;



// The following are HLA Standard Library procedures.
// Just make 'em labels rather than procs because we
// just JMP to these labels.

label
    shorthwExcept__hla_; @external;
    shortDfltExcept__hla_; @external;
    BuildExcepts__hla_; @external;
    QuitMain; @external( "QuitMain__hla_" );

static

    // The following is the link to the Win32 API ExitProcess procedure
    // address.

    __imp__ExitProcess :dword; @external( "__imp__ExitProcess@4" );

    // The main program needs a coroutine object for
    // use by the exception handling subsystem:

    MainPgmCoroutine__hla_: dword; @external;
    MainPgmCoroutine__hla_: dword; @nostorage;
        dword &MainPgmVMT__hla_;
        dword 0,0,0,0;

    MainPgmVMT__hla_: dword := &QuitMain;







// The following are needed to provide linkage to
// the HLA exception handling routines.


procedure HWexcept__hla_;
begin HWexcept__hla_;
        jmp shorthwExcept__hla_;
end HWexcept__hla_;


procedure DfltExHndlr__hla_;
```

```
begin DfltExHndlr__hla_;
        jmp shortDfltExcept__hla_;
end DfltExHndlr__hla_;




procedure _HLAMain;
begin _HLAMain;

        call    BuildExcepts__hla_;
        pushd( 0 );        // no dynamic link (previous proc's EBP).
        mov( esp, ebp );   // Set up our stack frame.
        push( ebp );       // Main's display.

        // << put main program code here >>

        stdout.put( "Hello World" nl );



end _HLAMain;

// Fall through from the above and return to Windows.
// (this needs to be outside _HLAMain because QuitMain__hla_
// needs to be a public name).

procedure QuitMain;
begin QuitMain;

    pushd(0);
    call( __imp__ExitProcess );

end QuitMain;


end unitAsPgm;
```

Hello World **Program** Written as a **Unit**

The HLA compiler instructs the linker to start program execution at the label *_HLAMain*. By writing a procedure named *_HLAMain* and making this name public (via the **external** directive), this unit provides an HLA "main program" that the OS will invoke immediately after loading the program into memory. This main program explicitly contains the instructions that the HLA compiler would normally emit for a program (the call to *BuildExcepts__HLA_* and setting up the activation record). Following the initialization code is the invocation of the stdout.put macro that prints "Hello World" to the standard output. One unusual feature of this code is that the *QuitMain* label has to be global and public (i.e., we can't simply put the code that returns to Windows inside the *_HLAMain* procedure because external code references this label and you can't reference local labels from outside a procedure). The alternative would be to duplicate the code, but then we wouldn't have the semantic equivalent of the original Hello World program. If you compare the assembly output of this code with the assembly output of the standard Hello World program, you'll find that the code is nearly identical (about the only real difference is the extra procedure surrounding the code that returns to the OS; of course, this does not change the executable file one byte).

Of course, it doesn't make any sense to simply duplicate the effects of an HLA program within a unit (other than to prove it can be done). The real reason for using units in this fashion is to gain complete control over the code appearing in the executable file. Specifically, I'm assuming you want to dump some of the initialization code, data structures, and support code that exist primarily for the benefit of the HLA run-time system and exception handling subsystem. Here's the bottom line, if you want to take full responsibility for all the code appearing in your HLA program, write it as a unit and create an *_HLAMain* procedure to serve as your main program (note: Linux users need to name their main program *_start*). Here's the template you should use:

```
unit barebones;

?@nodisplay := true;
?@noframe := true;

procedure _HLAMain; @external;

procedure _HLAMain;
begin _HLAMain;

   // Put the code for your main program here.

end _HLAMain;

end barebones;
```

Bare Bones HLA Program Implemented via a Unit

If you write your code using this "barebones" unit as a template, you're going to be in complete control of the code in your program. Do keep in mind that unless you initialize the exception handling system using the code given earlier (*BuildExcepts__HLA_*, etc.), you'll not be able to use HLA exceptions and that pretty much means you can't call any HLA Standard Library routines (since a large percentage of those can raise an exception). However, you will have completely escaped HLA's interference with your code and the only machine instructions that will find their way into your programs are the ones you write (or the code associated with any external routines you call).

I've made a big deal about using HLA units to give you complete control over the code HLA emits. Throughout this document, I've given the impression that only hard-core, die-hard, macho, assembly language programmers would want to do this. Actually, there are many real-world applications where the code that HLA emits for programs would be inappropriate. A classic example is the need to write dynamic link libraries. Such code has to be implemented as a **unit**, you cannot use an HLA procedure for such code.

15.9.8 Hello World, Revisited

This document lamented about the size of a typical Hello World program and mentioned that it's possible to write a shorter version of the program using HLA. In this section, we'll explore how to write a short version of this program. Actually, let's forget exploring and jump right into things.

Based on what I've said about the HLA Standard Library, it should come as no surprise that the smallest Hello World program is not going to call any Standard Library routines. The most compact Hello World program is going to make direct OS API calls. Well, without further ado, here are the compact versions of the Hello World program for Windows and Linux (different versions are necessary since the OS APIs are different).

```
unit HelloWorld;
```

```
?@noframe := true;

procedure main; @external( "_HLAMain" );

static

        WriteFile:procedure
        (
                Handle:          dword;
            var buffer:          var;
                len:             dword;
            var bytesWritten:    dword;
                overlapped:      dword
        );
            @use edx;
            @stdcall;
            @external( "__imp__WriteFile@20" );


        GetStdHandle:procedure
        (
            WhichHandle:int32
        );
            @stdcall;
            @external( "__imp__GetStdHandle@4" );

        ExitProcess:procedure( exitcode:dword );
            @stdcall;
            @external( "__imp__ExitProcess@4" );


procedure main;
var
    BytesWritten    :dword;
begin main;

    GetStdHandle( -11 );
    WriteFile( eax, &hwString, 13, BytesWritten, 0 );
    ExitProcess( 0 );

    hwString:   byte    "Hello World", $d, $a;


end main;


end HelloWorld;
```

Windows Version of the Short Hello World Program

Here’s the Linux version of this program:

```
unit hw;
```

```
procedure main; @external( "_start" );

procedure main; @noframe;
begin main;

    // Print Hello World:

    mov( 4, eax );
    mov( 1, ebx );
    lea( ecx, helloWorld );
    mov( 12, edx );
    int( $80 );

    // return to Linux:

    mov( 1, eax );
    mov( 0, ebx );
    int( $80 );

  helloWorld: byte "Hello World", $a;

end main;

end hw;
```

Linux Version of the Short Hello World Program