Writing a DLL in HLA

Dynamic link libraries provide an efficient mechanism for sharing code and cross-language linkage. The HLA language does not require any specific syntax to create a DLL; most of the work is done by the linker. However, to successfully write and call DLLs with HLA, you must follow some standard conventions.

Acknowledgement: I learned much of the material needed to write DLLs in HLA by visiting the following web page and looking at the CRCDemo file (which demonstrates how to write DLLs in assembly language). For more information on DLLs in assembly, you might want to take a look at this page yourself:

http://www.geocities.com/SiliconValley/Heights/7394/index.html

I certainly acknowledge stealing lots of information and ideas from this CRC code and documentation.

Creating a Dynamic Link Library

Win32 Dynamic Link Libraries provide a mechanism whereby two or more programs can share the same set of library object modules on the disk. At the very least, DLLs save space on the disk; if properly written and loaded into memory, DLLs can also share run-time memory and reduce swap space usage on the hard disk.

Perhaps even more important that saving space, DLLs provide a mechanism whereby two different programming languages may communicate with one another. Although there is usually no problems calling an assembly language (i.e., HLA) module from any given high level language, DLLs do provide one higher level of generality. In order to achieve this generality, Microsoft had to carefully describe the calling mechanism between DLLs and other modules. In order to communicate data, all languages that support DLLs need to agree on the calling and parameter passing mechanisms.

Microsoft has laid down the following rules for DLLs (among others):

- Procedures/functions with a fixed parameter list use the stdcall calling mechanism.
- Procedures/functions with a variable number of parameters use the C calling mechanism.
- Parameters can be bytes, words, doublewords, pointers, or strings. Pointers are machine addresses; strings are pointers to a zero-terminated sequence of characters, and it is up to the two modules to agree on how to interpret byte, word, or dword data (e.g., char, int16, uns32, etc.)

Stdcall procedures push their parameters from left to right as they are encountered in the parameter list. In stdcall procedures, it is the procedure's responsibility to clean up the parameters pushed on the stack.

HLA uses the stdcall calling mechanism for the HLL-style procedure calls, so this simplifies the interface to DLL code when using fixed parameter lists (variable parameter lists are rare in DLLs, but should they be necessary, one can always drop down into “pure” assembly in HLA and accomodate the DLL).

The only other issue, with respect to stdcall conventions, is the naming convention. The stdcall mechanism *mangles* procedure names. In particular, a procedure name like “XXXX” is translated to “_XXX@nn” where “nn” is the number of bytes of parameters passed to the procedure. HLA does not automatically mangle procedure names, but using the “external” directive you can easily specify the mangled name.

DLLs must provide a special procedure that Windows calls to initialize the procedure. This DLL entry point must use an HLA definition like the following:

procedure dll(instance:dword; reason:dword; reserved:dword);   external("_dll@12");

This function must return true in AL if the DLL can be successfully initialized; it returns false if it cannot properly initialize the DLL. Note that “dll” and “_dll@12” are example names; you may use any reasonable identifiers you choose here.

The DLL initialization function always has three parameters. The second parameter is the only one of real interest to the DLL initialization code. This parameter contains the reason for calling this code, which is one of the following constants defined in the w.hhf header file:

- w.DLL_PROCESS_ATTACH
- w.DLL_PROCESS_DETACH
- w.DLL_THREAD_ATTACH
- w.DLL_THREAD_DETACH

The w.DLL_XXXXX_ATTACH values indicate that some program is linking in the DLL. During these calls, you should open any files, initialize any variables, and execute any other initialization code that may be necessary for the proper operation of the DLL. Note that, by default, all processes that attach to a DLL get their own copy of any data defined in the DLL. Therefore, you do not have to worry about disturbing previous links to the DLL during the current initialization process.

The w.DLL_XXXXX_DETACH values indicate that a process or thread is shutting down. During these calls, you should close any files and perform any other necessary cleanup (e.g., freeing memory) that you would normally do before a program ends.

The following code demonstrates a short DLL:

```
unit dllExample;
#include( "w.hhf" );

static
    ThisInstance: dword;
```

```
procedure dll( instance:dword; reason:dword; reserved:dword );
        @stdcall; @external( "_dll@12" );

procedure dllFunc1( dw:dword ); @stdcall; @external( "_dllFunc1@4" );
procedure dllFunc2( dw2:dword ); @stdcall; @external( "_dllFunc2@4" );



procedure dll( instance:dword; reason:dword; reserved:dword ); @nodisplay;
begin dll;

    // Save the instance value.

    mov( instance, eax );
    mov( eax, ThisInstance );

    if( reason = w.DLL_PROCESS_ATTACH ) then

        // Do this code if we're attaching this DLL to a process...

    endif;

    // Return true if successful, false if unsuccessful.

    mov( true, eax );

end dll;


procedure dllFunc1( dw:dword ); @nodisplay;
begin dllFunc1;

    mov( dw, eax );

end dllFunc1;


procedure dllFunc2( dw2:dword ); @nodisplay;
begin dllFunc2;

    push( edx );
    mov( dw2, eax );
    mul( dw2, eax );
    pop( edx );

end dllFunc2;

end dllExample;
```

As you can see here, there is very little difference between a standard unit and an HLA unit intended to become a DLL. The name mangling is one difference, placing the external declarations directly in the file (rather than in an include file) is another difference. The only functional difference is the presence of the DLL initialization procedure ("dll" in this example).

The real work in creating a DLL occurs during the link phase. You cannot compile a DLL the same way you compile a standard HLA program - some additional steps are necessary. Creating a DLL requires lots of command line parameters, so it is best to create a makefile and a "linker" file to avoid excess typing at the command line. Consider the following make file for the module above:

```
dll.dll: dll.obj
      link dll.obj @dll.linkresp

dll.obj: dll.hla
      hla -@ -c dll.hla
```

This makefile generates the dll.dll file (it will also produce several other files, dll.lib being the most important one). The real work appears in the "dll.linkresp" linker file. This file contains the following text:

```
-DLL
-entry:dll
-base:0x40000000
-out:dll.dll
-export:dll
-export:dllFunc1
-export:dllFunc2
```

The "-DLL" option tells the linker to produce a "dll.dll" and a "dll.lib" file rather than just a "dll.exe" file (note: the linker will also produce some other files, but these two are the ones important to us).

The "-entry:dll" option tells the linker that the name of the DLL initialization code is the procedure "dll". If you change the name of your DLL initialization code, you should also change this option.

The "-base:0x40000000" option tells the linker that this DLL has a base address of 1GByte. For efficiency reasons, you should try to specify a unique value here. If two active DLLs specify the same base address, different processes cannot concurrently share the two DLLs. The programs will still operate, but they will not share the code, wasting some memory and requiring longer load times.

The "-out:dll.dll" command specifies the output name for the DLL. The suffix should be ".dll" and the base filename should be an appropriate name for your DLL ("dll" was appropriate in this case, it would not be appropriate in other cases).

The “-export” options specify the names of the external procedures you wish to make available to other modules. Alternately, you may create a “.DEF” file and use the “-DEF:deffilename.def” option to pass the exported file names on to the linker (see the Microsoft documentation for a description of DEF files).

If you run this make file, it will compile the dll.hla source file producing the dll.dll and dll.lib object modules.

Linking and Calling Procedures in a Dynamic Link Library

Creating a DLL in HLA is only half the battle. The other half is calling a procedure in a DLL from an HLA program. Here is a sample program that calls the DLL procedures in the previous section:

```
// Sample program that calls routines in dll.dll.
//
//  Compile this with the command line option:
//
//      hla dllmain dll.lib
//
//  Of course, you must build the DLL first.

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


procedure dllFunc1( dw:dword ); @stdcall; @external( "_dllFunc1@4" );
procedure dllFunc2( dw:dword ); @stdcall; @external( "_dllFunc2@4" );


begin callDLL;

    xor( eax, eax );
    dllFunc1( 12345 );
    stdout.put( "After dllFunc1, eax = ", (type uns32 eax ), nl );

    dllFunc2( 100 );
    stdout.put( "After dllFunc2, eax = ", (type uns32 eax ), nl );


end callDLL;
```

To compile this main program, you would use the following HLA command line:

```
hla dllmain dll.lib
```

The “dll.lib” file contains the linkages necessary to load and link in the dll module at run-time.

Going Farther

This document only explains “implicitly loaded” DLLs. Implicitly loaded DLLs are always loaded into memory when the main module loads into memory. If you want to control the loading of the DLL module into memory, you will want to take a look at “explicitly loaded” DLLs. Such DLLs, however, will have to be the subject of a different example.