About Dynamic-Link Libraries

<table>
<thead>
<tr>
<th>Authors</th>
<th><a href="mailto:tschou@iis.sinica.edu.tw">tschou@iis.sinica.edu.tw</a>, <a href="mailto:janeliu@iis.sinica.edu.tw">janeliu@iis.sinica.edu.tw</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Revision History</td>
<td>January 15, 2009</td>
</tr>
<tr>
<td>Note</td>
<td></td>
</tr>
<tr>
<td>Send corrections and suggestions to</td>
<td><a href="mailto:janeliu@iis.sinica.edu.tw">janeliu@iis.sinica.edu.tw</a></td>
</tr>
</tbody>
</table>

Introduction

The note provides a short list of how-to’s and don’t-do’s for writing and building dynamic-link libraries (DLL) in C. You can find more detailed and specific information than what is here from the document “About Dynamic-Link Libraries” at [http://msdn.microsoft.com/en-us/library/ms681914(VS.85).aspx](http://msdn.microsoft.com/en-us/library/ms681914(VS.85).aspx) and following the links provided there to other documents. If you need information that is not in this note, please look for and write up the information and share it with us.

This note does not contain some of the information on how to call and use DLL functions in C from C#. For that information, please consult the following:

The note also misses some of the development tool dependent information. That will be provided in a separate note.

Load-Time vs Run-Time Dynamic Linking

A module can call the function `LoadLibrary()` or `LoadLibraryEx()` to load a DLL at run time. We are not likely to use this option. Rather, we use load-time dynamic linking. In other words, all DLL’s are loaded when the application using them is loaded. (In our case, DLL’s are loaded when iNuC is loaded.)

Once loaded, a module linking with the import library for a DLL can call functions exported from the DLL as if they were local functions. This is because the import library
provides the system with information needed to load the DLL and locate the functions exported by it when the application is loaded.

**DLL Entry Point DLLMain( ) vs Module Init Function**

The API function `BOOL WINAPI DLLMain( )` was discussed in our meeting: Its prototype and template from MSDN listed below are for your information:

```c
BOOL WINAPI DllMain(
    __in  HINSTANCE hinstDLL, // handle of the DLL
    __in  DWORD fdwReason,     // reason for calling the function
    __in  LPVOID lpvReserved   // reserved
)
{
    // Perform actions based on the reason for calling.
    switch( fdwReason )
    {
        case DLL_PROCESS_ATTACH:
            // Initialize once for each new process.
            // Return FALSE to fail DLL load.
            break;

        case DLL_THREAD_ATTACH:
            // Do thread-specific initialization.
            break;

        case DLL_THREAD_DETACH:
            // Do thread-specific cleanup.
            break;

        case DLL_PROCESS_DETACH:
            // Perform any necessary cleanup.
            break;
    }
    return TRUE;  // Successful DLL_PROCESS_ATTACH.
}
```

In MSDN, this function is said to be an “optional entry point” into a DLL. The system calls the entry point of every DLL whenever it starts or terminates a process or a thread, i.e., for reasons of DLL attach (or detach) to process (or thread). For a run-time linked DLL, the system calls the function when `LoadLibrary` is called to load the DLL or `FreeLibrary` to unload the DLL at run-time.

The part “Main” of the function name is somewhat misleading. As indicated by the word `optional` in its definition, your DLL is not required to have an entry point. So, include the function in your module only if you want to do some simple initialization and/or cleanup whenever your DLL is attached and/or detached during startups and terminations of processes and threads: You can find examples of entry point functions at [http://www.codase.com/search/call?name=DisableThreadLibraryCalls](http://www.codase.com/search/call?name=DisableThreadLibraryCalls): These functions do work only for the reason `DLL_PROCESS_DETACH`. (They call the function `DisableThreadLibraryCalls()` to disables the `DLL_THREAD_ATTACH` and
DLL_THREAD_DETACH notifications for the specified dynamic-link library (DLL) in order to reduce the size of the working set.) Examples of entry point functions listed at http://www.winasm.net/forum/index.php?showtopic=2284 illustrates what may need to be done during process attachment.

MSDN documentation on it says: “There are serious limits on what you can do in a DLL entry point. To provide more complex initialization, create an initialization routine for the DLL.” In short, DLLMain ( ) is not the start procedure of the first thread of your module. For now, let us say that DLL’s provided by iNuC modules do not have entry point functions. As we planned earlier, we use the init API function (i.e., WTMInit( ), LIMInit( ), etc.) of your module as initialization routine of your DLL and start procedure of your main thread. However, if you want to do some initialization work (i.e., some bookkeeping) every time a thread starts or exits, you can conveniently do by having a DLLMain ( ).

Creation of DLL

http://msdn.microsoft.com/en-us/library/ms682592(VS.85).aspx provides information on DLL creation. Below is a summary of important points:

- Development tool: Many tool support the creation of Windows-based DLLs, including Microsoft Visual Studio 2005.
- Exporting Functions (and Data): What we want is to export data and functions of DLLs by names (i.e., not by ordinal values). You do so by using the keyword __declspec(dllexport): In other words, you add the keyword to declarations of function prototypes and data structure definitions you want to be exported from your DLL as in

```c
#define DLLEXPORT __declspec(dllexport)
#define DLLIMPORT __declspec(dllimport)

DLLEXPORT HANDLE MyAsynchronousFoo (BOOL Succeeded);
```

[Note: Please note the macros above. We will put them in iNuC.h. Use them! At least they save some typing for you.]

- Thread-Safe C Run-Time Library: iNuC is multi-threaded. Your DLL must be thread-safe and must use one of the thread-safe C run-time libraries: thread safe C run-time library (_MT) and DLL C run-time library (_MD, _DLL). They are slower than

---

1 The documentation at http://msdn.microsoft.com/en-us/library/ms682583.aspx also says “You can require applications to call the initialization routine before calling any other routines in the DLL. Alternatively, the initialization routine can create a file with an ACL that restricts access, and each routine in the DLL would call the initialization routine if the file does not exist.” We do not have to do it this way.

2 http://msdn.microsoft.com/en-us/library/7t44f2x1.aspx describes exporting data from DLL so that Win32-based applications can directly address data by name.
the non-thread-safe C run-time library, but the speed penalty for concurrency is
unavoidable for us. Larry Osterman’s weblog recommends the DLL C run-time
library because of “the code sharing opportunities.” At
http://msdn.microsoft.com/en-us/library/abx4dbyh.aspx, you can find .lib files that
make up C run-time libraries and associated compiler options and preprocessor
directives.

Import Library: An import library is a .lib file containing information that linker needs
to resolve external references to exported DLL functions. You can find from
library for your DLL. The syntax to use for building import library is

\text{LIB /DEF[:deffile] [options] [objfiles] [libraries]}

If you use MS Visual Studio 2005 to build your DLL project, the step of building
import library can be omitted because MS Visual Studio 2005 will do it for you.

Import and Mutual Import

Your module may need to use function and data names exported by other DLL’s (i.e., “use
\text{public symbols defined by a DLL”}). Your program is said to import them. You do so by using
the keyword \text{__declspec(dllimport)}, which is defined by macro as DLLIMPORT, on the
declarations as in

\text{DLLIMPORT VOID SomeDLLFunction();}

During one of our meetings, we mentioned the need for a module, a DLL (Say DLL1) to
import from another DLL (say DLL2). DLL2 may in turn import from DLL1. This is called
\text{mutual import}. The problem with mutual import of DLL’s is obvious: Neither one can be
built with building the other one first. There are two ways to overcome this circular
dependency problem:

It is illustrated by Figure 1, which shows two mutually importing DLLs: DLL1 and DLL2.
You build them in three steps:

\begin{itemize}
  \item In Step 1, you run LIB, with the /DEF option set, on DLL1. This step produces
        DLL1.lib, an import library, and DLL1.exp.
  \item In step 2, you use the import library DLL1.lib to build DLL2. This step produces in
        turn an import library for DLL2’s symbols.
\end{itemize}
In Step 3, you build executables of DLL1, using DLL1.exp and DLL2.lib as input. Note that an .exp file for DLL2 is not necessary because LIB was not used to build DLL2's import library.

The second method to break circular dependency of DLL’s is illustrated by Figure 2. It works with MS Visual Studio 2005. The tool will automatically generate YourDLL.lib and YourDLL.exp when you successfully build YourDLL.dll.

The method also has two steps. The purpose of the first step is to get .exp and .lib files for your DLL and have them contain export information on your DLL. For this purpose, you only need provide prototypes of your export functions. In other words, you can write empty functions first and build YourDLL project to get YourDLL.lib or YourDLL.exp, as illustrated by the following empty function.
The second step can proceed after you have .lib and .exp files of all the DLL's your DLL import from. In this step, you complete the implementation of each export function and compile your DLL project again. This method is easier when the relationship of mutual (circular) dependency of your project is complicated.

**Testing Your DLL**

To test your DLL, you need to write and build an .exe file that links to your DLL. As stated earlier, we use load-time dynamic linking. To do so, you include YourDLL.h when writing your TestYourDLL.c and link TestYourDLL.c with YourDLL.lib at compile time. (If you use MS Visual Studio 2005, you can specify .lib file(s) to link by selecting View->Property Pages->Linker->Input and then setting the .lib file(s) you want to link in Additional Dependencies field.) Below is an illustrative example.

```c
// File Name: YourDLL.h

// Use the macro DLLEXPORT defined in the common header file iNuC.h
DLLEXPORT int YourDLLVariable;      // Consider the variable global
DLLEXPORT VOID YourDLLFunction();

// File Name: TestYourDLL.c
#include "YourDLL.h"

void main()
{
    //Import the function and variable from your DLL
    printf("YourDLLVariable is %d\n", YourDLLVariable);
    YourDLLFunction();
}```