DLL - HOW TO WRITE

First, we will discuss the issues and the requirements that you should consider while developing your own DLLs.

Types of DLLs

When you load a DLL in an application, two methods of linking let you call the exported DLL functions. The two methods of linking are −

- load-time dynamic linking, and
- run-time dynamic linking.

Load-time dynamic linking

In load-time dynamic linking, an application makes explicit calls to the exported DLL functions like local functions. To use load-time dynamic linking, provide a header *.h* file and an import library *.lib* file, when you compile and link the application. When you do this, the linker will provide the system with the information that is required to load the DLL and resolve the exported DLL function locations at load time.

Runtime dynamic linking

In runtime dynamic linking, an application calls either the LoadLibrary function or the LoadLibraryEx function to load the DLL at runtime. After the DLL is successfully loaded, you use the GetProcAddress function, to obtain the address of the exported DLL function that you want to call. When you use runtime dynamic linking, you do not need an import library file.

The following list describes the application criteria for choosing between load-time dynamic linking and runtime dynamic linking −

- **Startup performance** − If the initial startup performance of the application is important, you should use run-time dynamic linking.
- **Ease of use** − In load-time dynamic linking, the exported DLL functions are like local functions. It helps you call these functions easily.
- **Application logic** − In runtime dynamic linking, an application can branch to load different modules as required. This is important when you develop multiple-language versions.

The DLL Entry Point

When you create a DLL, you can optionally specify an entry point function. The entry point function is called when processes or threads attach themselves to the DLL or detach themselves from the DLL. You can use the entry point function to initialize or destroy data structures as required by the DLL.

Additionally, if the application is multithreaded, you can use thread local storage *TLS* to allocate memory that is private to each thread in the entry point function. The following code is an example of the DLL entry point function.

```
BOOL APIENTRY DllMain(
   HANDLE hModule,     // Handle to DLL module
   DWORD ul_reason_for_call,
   LPVOID lpReserved )     // Reserved
{
   switch ( ul_reason_for_call )
   {
      case DLL_PROCESS_ATTACHED:
      // A process is loading the DLL.
      break;
      case DLL_THREAD_ATTACHED:
      // A process is creating a new thread.
      break;

      case DLL_THREAD_DETACH:
      // A thread exits normally.
      break;

      case DLL_PROCESS_DETACH:
      // A process unloads the DLL.
      break;
   }
   return TRUE;
}
```

When the entry point function returns a FALSE value, the application will not start if you are using load-time dynamic linking. If you are using runtime dynamic linking, only the individual DLL will not load.

The entry point function should only perform simple initialization tasks and should not call any other DLL loading or termination functions. For example, in the entry point function, you should not directly or indirectly call the **LoadLibrary** function or the **LoadLibraryEx** function. Additionally, you should not call the **FreeLibrary** function when the process is terminating.

WARNING − In multithreaded applications, make sure that access to the DLL global data is synchronized *threadsafe* to avoid possible data corruption. To do this, use TLS to provide unique data for each thread.

Exporting DLL Functions

To export DLL functions, you can either add a function keyword to the exported DLL functions or create a module definition . *def* file that lists the exported DLL functions.

To use a function keyword, you must declare each function that you want to export with the following keyword −

```
__declspec(dllexport)
```

To use exported DLL functions in the application, you must declare each function that you want to import with the following keyword −

```
__declspec(dllimport)
```

Typically, you would use one header file having **define** statement and an **ifdef** statement to separate the export statement and the import statement.

You can also use a module definition file to declare exported DLL functions. When you use a module definition file, you do not have to add the function keyword to the exported DLL functions. In the module definition file, you declare the **LIBRARY** statement and the **EXPORTS** statement for the DLL. The following code is an example of a definition file.

```
// SampleDLL.def
//
LIBRARY "sampleDLL"

EXPORTS
   HelloWorld
```

Write a Sample DLL

In Microsoft Visual C++ 6.0, you can create a DLL by selecting either the **Win32 Dynamic-Link Library** project type or the **MFC AppWizard** ***dll*** project type.

The following code is an example of a DLL that was created in Visual C++ by using the Win32 Dynamic-Link Library project type.

```
// SampleDLL.cpp

#include "stdafx.h"
#define EXPORTING_DLL
#include "sampleDLL.h"

BOOL APIENTRY DllMain( HANDLE hModule, DWORD  ul_reason_for_call, LPVOID lpReserved )
{
   return TRUE;
}

void HelloWorld()
{
   MessageBox( NULL, TEXT("Hello World"),
   TEXT("In a DLL"), MB_OK);
}
```

```
// File: SampleDLL.h
//
#ifndef INDLL_H
#define INDLL_H

   #ifdef EXPORTING_DLL
      extern __declspec(dllexport) void HelloWorld() ;
   #else
      extern __declspec(dllimport) void HelloWorld() ;
   #endif

#endif
```

Calling a Sample DLL

The following code is an example of a Win32 Application project that calls the exported DLL function in the SampleDLL DLL.

```
// SampleApp.cpp

#include "stdafx.h"
#include "sampleDLL.h"

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nCmdShow)
{
   HelloWorld();
   return 0;
}
```

NOTE − In load-time dynamic linking, you must link the SampleDLL.lib import library that is created when you build the SampleDLL project.

In runtime dynamic linking, you use code that is similar to the following code to call the SampleDLL.dll exported DLL function.

```
...
typedef VOID (*DLLPROC) (LPTSTR);
...
HINSTANCE hinstDLL;
DLLPROC HelloWorld;
BOOL fFreeDLL;

hinstDLL = LoadLibrary("sampleDLL.dll");

if (hinstDLL != NULL)
{
   HelloWorld = (DLLPROC) GetProcAddress(hinstDLL, "HelloWorld");
   if (HelloWorld != NULL)
      (HelloWorld);

   fFreeDLL = FreeLibrary(hinstDLL);
}
...
```

When you compile and link the SampleDLL application, the Windows operating system searches for the SampleDLL DLL in the following locations in this order −

- The application folder
- The current folder
- The Windows system folder (The **GetSystemDirectory** function returns the path of the Windows system folder).
- The Windows folder (The **GetWindowsDirectory** function returns the path of the Windows folder).