30

DYNAMIC LINK LIBRARIES

Dynamic link libraries (DLLs) provide an important facility within the Windows operating system. DLLs provide the ability to plug prewritten modules into the operating system, allowing applications to share common code, modularize commonly used functions, and provide extendability. The Windows operating system itself is comprised mostly of DLLs, and any application that makes use of the Win32 API is making use of the DLL facility. An overview of Dynamic Link Libraries (DLLs) application can gain the use of procedures located within a DLL by two methods: load-time dynamic linking and run-time dynamic linking.

Load-time Dynamic Linking

An application employs load-time dynamic linking by specifying the names of the DLL procedures directly in the source code. The linker inserts references to these procedures when it locates them in an import library linked with the application or by using the IMPORTS section of the module definition file for the application. When the application is executed, the Windows loader loads the DLLs into memory and resolves these references.

This is the easiest form of dynamic linking, but it can cause problems under special circumstances. For example, if an application references a DLL procedure in this manner, the DLL must exist when the application is executed, even if the application never actually calls that procedure. Also, the application must know at compile-time the names of all procedures that will be load-time linked.

Run-time Dynamic Linking

Although run-time dynamic linking overcomes the limitations of load-time dynamic linking, it does requires more work on the part of the application. Rather than specify at compile-time the DLL procedures that an application wishes to use, the application uses the LoadLibrary(), LoadLibraryEx(), GetProcAddress(), and FreeLibrary() functions to specify at run-time the names of the DLLs and procedures that it wishes to reference.

Run-time dynamic linking also allows an application to provide support for functionality that is not available when the application is created. For example, a word processing application can provide conversion routines inside DLLs that convert between different file formats. If runtime dynamic linking is used, new conversion DLLs can be added that contain conversion routines for new file formats that did not exist when the application was written. Since the application determines what conversion DLLs exist at run time, installing the new DLLs allows the application to make use of the new conversion procedures. The application would use the FindFirstFile() and FindNextFile() functions to retrieve the names of the DLLs that are present. It could then load each DLL, obtain the address of the

conversion procedure, and place these addresses in a structure that it would later use during the file conversion process. For an example of this, see the example under the LoadLibrary() function later in this chapter.

When using GetProcAddress(), you must specify the routine whose address you wish to obtain. You can specify the name of the routine as an ASCII string or you may specify the ordinal number of the routine. See the description of GetProcAddress() later in this chapter. Although using ordinal numbers is more efficient than using procedure names when calling GetProcAddress(), you should be aware of potential problems.

Routines in the DLL are kept in a name table. Windows uses the ordinal number as an index to this table when resolving references to the routines. The only check that is made by Windows is to ensure that the ordinal number is within the range of the name table. If DLL routines are assigned ordinal numbers that are not contiguous (i.e. there are gaps in the numbering), it is possible to specify a number that is within the range of the name table but references an unused entry. GetProcAddress() will blindly return the invalid entry in the name table for this ordinal. The result most likely will be a general protection fault when the application attempts to make use of the returned address. In addition, if the developer of the DLL allows the linker to assign default ordinal numbers to the routines in the DLL, it is possible that a new version of the DLL will have a different ordinal number assignment scheme. This would obviously cause GetProcAddress() to return the address of the wrong routine.

Creating Dynamic Link Libraries

The steps involved in creating a DLL are very similar to those for creating standard Windows executable programs. The source code for a DLL is identical to the source code for any executable module. The following example shows the source code for a very simple DLL:

```
#include <windows.h>

//  AddNumbers - This is a routine in a DLL that an application may call.

int AddNumbers(int n1, int n2)
{
    return(n1 + n2);
}
```

This is the same as any subroutine you might code for an ordinary application. The differences occur in the module definition file (.DEF). The linker uses the module definition file to provide information specific to the creation of DLLs. The following example illustrates a minimal module definition file for the example given above:

```
LIBRARY MyDll
EXPORTS
    AddNumbers @1
```

The function AddNumbers() is defined as being exported (from the perspective of the DLL) and is assigned the ordinal value of one. The ordinal value is optional and, if omitted, is assigned by the linker automatically. When the object file created from the source code example above is linked with the example module definition file, the result is the dynamic link library.

If your application will be using runtime dynamic linking to access this DLL, then you've completed the creation process. If you plan to use load-time dynamic linking to access this DLL, then you must complete a few more tasks first.

Since load-time dynamic linking requires resolution of all references at compile/link time, you must first provide a header file (.H) that defines the routines in your DLL. The header file typically contains prototype definitions for all public entry points in the DLL, as well as any structures or definitions. For the above example, the header file would look like this:

```
//  Header file for MYDLL.DLL - Defines entry points

int AddNumbers(int n1, int n2);
```

Applications employing load-time dynamic link with your DLL will include this file to gain the definition of the AddNumbers() routine. This will satisfy the compiler. The linker, however, needs to know where to find the routine

AddNumbers(). This is typically done by linking the application with an Import Library that is created from your module definition file. This import library will tell the linker that the routine AddNumbers() exists within the file MYDLL.DLL. Many development platforms create the import library for you. However, if you must create the import library yourself, you can use the LIB.EXE utility with a command line similar to the following:

```
lib /machine:IX86 /OUT:mydll.lib /DEF:mydll.def
```

This example creates an import library suitable for use on x86 based platforms. The application would include the library MYDLL.LIB as input to the linker, allowing the references to the DLL to be properly resolved.

If an import library is not provided with the DLL, then an application using load-time dynamic linking must employ the module definition file to inform the linker of the locations of the DLL routines. This is done using the IMPORTS section of the module definition file. An application using the DLL shown in these examples would include the following lines in its module definition file:

```
IMPORTS
    MyDll.MyDllFunc
```

Accessing Data Within a DLL

When a DLL dynamically allocates memory, that memory is allocated on behalf of the process that is calling the DLL. This memory is accessible only to threads belonging to this process. This would include memory allocated using GlobalAlloc(), LocalAlloc(), HeapAlloc(), VirtualAlloc(), or any of the standard C library routines, such as mallc() and new().

If a DLL dynamically allocates memory, so that it can be shared among all of the processes currently using the DLL, it should use the file-mapping functions to create named shared memory. For more details on file mapping, see Chapter 17, I/O with Files.

Global and static variables declared in the DLL 's source code exist within the data segment of the DLL. Each process attached to the DLL is provided with its own copy of this data segment.

It is possible for the module definition file to change the default attributes of global and static data, allowing the data to be shared among all processes using the DLL. For example, including the following line in the module definition file will cause the DLL 's initialized data to be shared among all processes, while uninitialized data will remain private to each process:

```
SECTIONS
    .GLOBALS READ WRITE SHARED
```

Finally, a DLL can make use of the thread local storage functions to provide a table of values that are unique to each thread accessing the DLL. For more information on thread local storage, refer to Chapter 25, Processes and Threads. In addition, use of thread local storage typically requires implementation of a notification entry point within the DLL.

Using the DLL Notification Entry Point

The DLL notification entry point allows a DLL to be informed every time a process attaches to, or detaches from, the DLL, or a process attached to the DLL creates or destroys a thread. To implement a DLL notification entry point, you must code a function in your DLL, based on the DLLEntryPoint() prototype function:

```
BOOL DLLEntryPoint(HINSTANCE hInstDLL, DWORD dwNotification, LPVOID lpReserved);
```

The function name DLLEntryPoint() is simply a placeholder. By default the linker looks for a function with the name of DLLMain(). You may name the function anything you wish if you specify the dll entry point.. This is typically done by using an /ENTRY switch on the linker command line, as shown in the following example:

```
link /entry:MyEntry /machine:IX86 mydll.obj
```

Refer to the detailed description of this function for more information.

Function Summary

Table 30-1 summarizes the functions used to work with Dynamic Link Libraries. A detailed description of each of these functions follows the table.

Table 30-1. Dynamic Link Library Function Summary

Function	Purpose
DisableThreadLibraryCalls	Prevents Windows from sending notifications to the specified library during thread creation and termination.
DLLEntryPoint	Optional notification entry point for dynamic link libraries.
FreeLibrary	Decrements the usage count of a loaded DLL and releases the DLL from memory when the usage count reaches zero.
FreeLibraryAndExitThread	Decrements the usage count of a loaded DLL and releases the DLL from memory when the usage count reaches zero. Terminates the calling thread.
GetProcAddress	Returns the address of a routine located in a DLL.
LoadLibrary	If necessary, loads a DLL into memory. Increments the DLLs usage count and returns a handle to the DLL.
LoadLibraryEx	If necessary, loads a DLL into memory. Increments the DLLs usage count and returns a handle to the DLL. Allows the calling application to disable notifications to the DLL during process and thread creation and termination.

DISABLETHREADLIBRARYCALLS

WIN32S WINDOWS95 WINDOWSNT

Description	An application uses the DisableThreadLibraryCalls() function to inform Windows that it should no longer notify the specified DLL whenever the application creates or terminates threads. You must be careful when using this function because DLLs may need to provide per-thread structures or resources. If you are certain that no DLLs require the thread notification messages, and your application creates and destroys many threads, using DisableThreadLibraryCalls() can improve performance. This function will fail if the DLL has active static thread local storage.
Syntax	BOOL DisableThreadLibraryCalls(HMODULE hDLLibrary)
Parameters	
hDLLibrary	HMODULE: A handle to the DLL. This handle is returned by the LoadLibrary() function.
Returns	BOOL: If the function is successful, it returns TRUE. Otherwise, it returns FALSE. The application can obtain further information by calling GetLastError().
Include File	winbase.h
See Also	LoadLibrary(), LoadLibraryEx()

DLLENTRYPOINT

WIN32S WINDOWS95 WINDOWSNT

Description	Windows will execute a DLL 's DLLEntryPoint() function to notify the DLL of certain important events. Use of this routine is optional; however, for a DLL to implement it, the name of the function must be provided to the linker as the library's execution entry point. The name DLLEntryPoint() is simply a placeholder. DLLMain() is the default entry point name. If the DLL includes a DLLMain() function, there is no need for specifying the entry point at link time. However, windows will call whatever function is specified as the library's entry point at link time if the name is different than DLLMain().
Syntax	BOOL DLLEntryPoint(HINSTANCE hInstDLL, DWORD dwNotification, LPVOID lpReserved)
Parameters	
hInstDLL	HINSTANCE: The instance handle of the DLL itself.
dwNotification	DWORD: Notification code. See Table 30-2 for a list of valid notification codes that may be sent by Windows.

lpReserved LPVOID: Reserved, set to NULL.

Table 30-2. Dynamic Link Library Notification Codes

Notification Code	Description
DLL_PROCESS_ATTACH	Informs the DLL that a new process is attaching. A DLL_THREAD_ATTACH notifcication is not generated for any threads that currently exist within the process, including the initial thread.
DLL_THREAD_ATTACH	Informs the DLL that a new thread has been created by a process that is attached to the DLL.
DLL_THREAD_DETACH	Informs the DLL that the executing thread is terminating.
DLL_PROCESS_DETACH	Informs the DLL that the executing process is finished with the DLL, either because the process has terminated or it has freed the library.

Returns BOOL: The return value is ignored for all notification codes except DLL_PROCESS_ATTACH. When DLL_PROCESS_ATTACH is used, the DLL should return TRUE if initialization was successful, and the application may continue. If the DLL returns FALSE, then the application will be terminated if it is using load-time dynamic linking or will receive an error code from LoadLibrary() or LoadLibraryEx() if it is using runtime dynamic linking.

Example The following example illustrates the the DLLMain()notification entry point. This example simply keeps track of the number of processes and threads currently attached.

```
// Make this data shared among all
// all applications that use this DLL.
//................................
#pragma data_seg( ".GLOBALS" )
int nProcessCount = 0;
int nThreadCount  = 0;
#pragma data_seg()

BOOL WINAPI DLLMain( HINSTANCE hInstDLL, DWORD dwNotification, LPVOID lpReserved )
{
    switch(dwNotification)
    {
        case DLL_PROCESS_ATTACH :
               // DLL initialization code goes here. Formerly this
               // would be in the LibMain function of a 16-bit DLL.
               //.................................................
               nProcessCount++;
               return( TRUE );

        case DLL_PROCESS_DETACH :
               // DLL cleanup code goes here. Formerly this would
               // be in the WEP function of a 16-bit DLL.
               //.............................................
               nProcessCount--;
               break;

        case DLL_THREAD_ATTACH :
               // Special initialization code for new threads goes here.
               // This is so the DLL can "Thread Protect" itself.
               //.....................................................
               nThreadCount++;
               break;

        case DLL_THREAD_DETACH :
               // Special cleanup code for threads goes here.
               //.........................................
               nThreadCount--;
               break;
    }

    return( FALSE );
}
```

FREELIBRARY

WIN32S WINDOWS95 WINDOWSNT

Description	An application uses the FreeLibrary() function to inform Windows that it no longer needs the indicated DLL. Windows will decrement the usage count for the DLL; if the usage count reaches zero, Windows will release the DLL from memory.
Syntax	BOOL FreeLibrary(HMODULE hDLLibrary)
Parameters	
hDLLibrary	HMODULE: A handle to the DLL. This handle is returned by the LoadLibrary( ) function.
Returns	BOOL: If the function is successful, it returns TRUE; otherwise, it returns FALSE. The application can obtain further information by calling GetLastError( ).
Include File	winbase.h
See Also	GetModuleHandle( ), LoadLibrary(), LoadLibraryEx()
Example	See LoadLibrary() for an example of this function.

FREELIBRARYANDEXITTHREAD

WIN32S WINDOWS95 WINDOWSNT

Description	An application uses the FreeLibraryAndExitThread() function to inform Windows that it no longer needs the indicated DLL and that the calling thread should terminate. Windows will decrement the usage count for the DLL, and if the usage count reaches zero, Windows will release the DLL from memory. Finally, Windows will terminate the calling thread. This function is an atomic combination of FreeLibrary() and ExitThread(). It is useful in situations where the calling thread is created and executes within a DLL. The thread cannot simply call FreeLibrary() because Windows may unload the DLL, causing the thread's code to become invalid before the thread has had a chance to regain control and call ExitThread().
Syntax	BOOL FreeLibraryAndExitThread(HMODULE hDLLibrary)
Parameters	
hDLLibrary	HMODULE: A handle to the DLL. This handle is returned by the LoadLibrary() function.
Returns	BOOL: If this function is successful, it does not return. After freeing the indicated library, the calling thread is terminated and does not regain control. Otherwise, FALSE is returned.
Include File	winbase.h
See Also	ExitThread(), FreeLibrary(), LoadLibrary(), LoadLibraryEx()
Example	The following example illustrates the use of this function. It is assumed that the following code exists within a loaded DLL and is making use of another DLL. When this function is called, the indicated library will be freed, and the calling thread will exit. If the function returns, the application issues a fatal error.

```
HMODULE hModule;
hModule = LoadLibrary("MYDLLLIB.DLL");
.
.
.
FreeLibraryAndExit(hModule);
FatalError("Current thread is unable to terminate.");
```

GETPROCADDRESS

WIN32S WINDOWS95 WINDOWSNT

Description	Use GetProcAddress() to obtain the address of a function that resides within a DLL. The application can then use this address to invoke the function.
Syntax	FARPROC GetProcAddress(HMODULE hDLLibrary, LPCSTR lpszProcName)
Parameters	
hDLLibrary	HMODULE: A handle to the DLL. This handle is returned by the LoadLibrary() function.

lpszProcName	LPCSTR: This parameter can be either a pointer to a null-terminated string identifying the procedure or the ordinal number of a procedure. If an ordinal number is used, the ordinal must be in the low word, and the high word must be zero.
Returns	FARPROC: If the function is successful, it returns the address of the requested procedure's entry point. Otherwise, the return value is NULL. The application can obtain further information by calling GetLastError().
Include File	winbase.h
See Also	FreeLibrary(), GetModuleHandle(), LoadLibrary(), LoadLibraryEx()
Example	See LoadLibrary() for an example of this function.

LOADLIBRARY

WIN32S WINDOWS95 WINDOWSNT

Description	Use LoadLibrary() to obtain the module handle of a DLL. If the DLL is not resident in memory, Windows will load it. Windows then increments the usage count for the library and returns a module handle that identifies the library.
Syntax	HANDLE LoadLibrary(LPCSTR lpszLibraryName)
Parameters	
lpszLibraryName	LPCSTR: Points to a null-terminated string that specifies the file name of the module to load. This module may be a DLL or an executable module. If no file extension is specified, .DLL is assumed. If no path name is specified, Windows searches for the DLL first in the directory from which the calling application was loaded, then in the current directory, the Windows system directory, the Windows directory, and finally, through all directories referenced in the PATH environment string.
Returns	HANDLE: If this function is successful, it returns the module handle of the loaded library. Otherwise, it returns NULL. The application can obtain further information by calling GetLastError().
Include File	winbase.h
See Also	FreeLibrary(), FindResource(), GetProcAddress( ), LoadLibraryEx(), LoadResource()
Example	The following example loads the library GDI32.DLL and uses GetProcAddress() to obtain the address of the routines BeginPath(), EndPath(), and StrokePath(). These routines do not exist in older versions of Windows NT. If the routines were simply referenced using load-time dynamic linking, the application would not load under versions of Window NT where these functions are not available. Using runtime dynamic linking, the application is able to continue with conventional drawing, even though the path functions do not exist.

```
LRESULT CALLBACK WndProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam )
{
   switch( uMsg )
   {
      case WM_COMMAND :
               switch( LOWORD( wParam ) )
               {
                  case IDM_TEST :
                         {
                            // Get the instance handle of the GDI DLL.
                            //.....................................
                            HINSTANCE hLib = LoadLibrary( "GDI32.DLL" );

                            if ( hLib )
                            {
                               // Get the function pointers to the path functions needed.
                               //.....................................................
                               FARPROC fpBeginPath  = GetProcAddress( hLib, "BeginPath" );
                               FARPROC fpEndPath    = GetProcAddress( hLib, "EndPath" );
                               FARPROC fpStrokePath = GetProcAddress( hLib, "StrokePath" );
                               HDC     hDC          = GetDC( hWnd );
```

```
                        // If paths are supported, draw with them.
                        // Otherwise, use standard drawing.
                        //.....................................
                        if ( fpBeginPath && fpEndPath && fpStrokePath )
                        {
                           (*fpBeginPath)( hDC );
                           LineTo( hDC, 200, 100 );
                           (*fpEndPath)( hDC );
                           (*fpStrokePath)( hDC );
                        }
                        else
                           LineTo( hDC, 200, 100 );

                        ReleaseDC( hWnd, hDC );

                        FreeLibrary( hLib );
                     }
                     else
                        MessageBox( hWnd, "DLL could not be loaded", NULL, MB_OK | MB_ICONASTERISK );
                  }
                  break;
      .
      .
      .
```

LOADLIBRARYEX

WIN32S WINDOWS95 WINDOWSNT

Description	Use LoadLibraryEx() to obtain the module handle of a DLL. If the DLL is not resident in memory, Windows will load it. Windows then increments the usage count for the library and returns a module handle that identifies the library.
Syntax	HANDLE LoadLibraryEx(LPCSTR lpszLibraryName, HANDLE hReserved, DWORD dwFlags)
Parameters	
lpszLibraryName	LPCSTR: Points to a null-terminated string that specifies the file name of the module to load. This module may be a DLL or an executable module. If no file extension is specified, .DLL is assumed. If no path name is specified, Windows searches for the DLL first in the directory from which the calling application was loaded, then in the current directory, the Windows system directory, the Windows directory, and finally, through all directories referenced in the PATH environment string.
hReserved	HANDLE: Reserved, must be NULL.
dwFlags	DWORD: If this value is DONT_RESOLVE_DLL_REFERENCES, calls to the DLLEntryPoint() function will not be generated for process and thread creation and termination. If this value is zero, then this function is identical to LoadLibrary().
Returns	HANDLE: If this function is successful, it returns the module handle of the loaded library. Otherwise, it returns NULL. The application can obtain further information by calling GetLastError().
Include File	winbase.h
See Also	FreeLibrary(), FindResource(), GetProcAddress(), LoadResource()
Example	The following example loads a DLL into memory, obtains the address of a procedure within the library, calls the library function, and finally frees the library. Note that Windows will not generate calls to the DLL 's DLLMain() routine during this code.

```
typedef int (*ADDNUMBERS)(int, int);

LRESULT CALLBACK WndProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam )
{
   switch( uMsg )
   {
      case WM_COMMAND :
              switch( LOWORD( wParam ) )
```

```
{
   case IDM_TEST :
          {
             // Get the instance handle of the MyDLL.DLL.
             //.....................................
             HINSTANCE hLib = LoadLibraryEx( "MyDLL.DLL", 0, DONT_RESOLVE_DLL_REFERENCES );

             if ( hLib )
             {
                // Get the function pointers to the function needed.
                //.................................................
                ADDNUMBERS fpAddNumbers = (ADDNUMBERS)GetProcAddress( hLib, "AddNumbers" );

                if ( fpAddNumbers )
                {
                   TCHAR szMsg[32];
                   int   nAnswer = (*fpAddNumbers)( 10, 20 );

                   wsprintf( szMsg, "10 + 20 = %d", nAnswer );
                   MessageBox( hWnd, szMsg, lpszTitle, MB_OK | MB_ICONINFORMATION );
                }
                else
                   MessageBox( hWnd, "Cannot find function!", NULL, MB_OK | MB_ICONASTERISK );

                FreeLibrary( hLib );
             }
             else
                MessageBox( hWnd, "DLL could not be loaded", NULL, MB_OK | MB_ICONASTERISK );
          }
          break;
.
.
.
```