Windows Dynamic-Link Libraries (DLLs)

What do we have in this session?

Intro
Types of Dynamic Linking
DLLs and Memory Management
Advantages of Dynamic Linking
Dynamic-Link Library Creation
Creating Source Files
Exporting Functions
Creating an Import Library
Using an Import Library
Dynamic-Link Library Entry-Point Function
Calling the Entry-Point Function
Entry-Point Function Definition
Entry-Point Function Return Value
Load-Time Dynamic Linking
Run-Time Dynamic Linking
Dynamic-Link Library Search Order
Standard Search Order
Alternate Search Order
Dynamic-Link Library Data
Variable Scope
Dynamic Memory Allocation
Thread Local Storage
Dynamic-Link Library Redirection
Dynamic-Link Library Updates
Using Dynamic-Link Libraries: Program Examples
Creating a Simple Dynamic-Link Library Example
Using Load-Time Dynamic Linking Example
Using Run-Time Dynamic Linking Program Example
Using Shared Memory in a Dynamic-Link Library Example
DLL that Implements the Shared Memory
Processes that Use the Shared Memory Program Example
Using Thread Local Storage in a Dynamic-Link Library Program Example
Dynamic-Link Library Functions
Obsolete Functions
Intro

A dynamic-link library (DLL) is a module that contains functions and data that can be used by another module (application or DLL). A DLL can define two kinds of functions:

1. Exported
2. Internal

The exported functions are intended to be called by other modules, as well as from within the DLL where they are defined. Internal functions are typically intended to be called only from within the DLL where they are defined. Although a DLL can export data, its data is generally used only by its functions. However, there is nothing to prevent another module from reading or writing that address.

DLLs provide a way to modularize applications so that their functionality can be updated and reused more easily. DLLs also help reduce memory overhead when several applications use the same functionality at the same time, because although each application receives its own copy of the DLL data, the applications share the DLL code.

The Windows application programming interface (API) is implemented as a set of DLLs, so any process that uses the Windows API uses dynamic linking. Dynamic linking allows a module to include only the information needed to locate an exported DLL function at load time or run time. Dynamic linking differs from the more familiar static linking, in which the linker copies a library function's code into each module that calls it.

Types of Dynamic Linking

There are two methods for calling a function in a DLL (import):

1. In **load-time dynamic linking**, a module makes explicit calls to exported DLL functions as if they were local functions. This requires you to link the module with the import library for the DLL that contains the functions. An import library supplies the system with the information needed to load the DLL and locate the exported DLL functions when the application is loaded.
2. In **run-time dynamic linking**, a module uses the LoadLibrary() or LoadLibraryEx() function to load the DLL at run time. After the DLL is loaded, the module calls the GetProcAddress() function to get the addresses of the exported DLL functions. The
module calls the exported DLL functions using the function pointers returned by GetProcAddress(). This eliminates the need for an import library.

**DLLs and Memory Management**

Every process that loads the DLL maps it into its virtual address space. After the process loads the DLL into its virtual address, it can call the exported DLL functions. The system maintains a per-process reference count for each DLL. When a thread loads the DLL, the reference count is incremented by one. When the process terminates, or when the reference count becomes zero (run-time dynamic linking only), the DLL is unloaded from the virtual address space of the process. Like any other function, an exported DLL function runs in the context of the thread that calls it. Therefore, the following conditions apply:

1. The threads of the process that called the DLL can use handles opened by a DLL function. Similarly, handles opened by any thread of the calling process can be used in the DLL function.
2. The DLL uses the stack of the calling thread and the virtual address space of the calling process.
3. The DLL allocates memory from the virtual address space of the calling process.

**Advantages of Dynamic Linking**

Dynamic linking has the following advantages over static linking:

1. Multiple processes that load the same DLL at the same base address share a single copy of the DLL in physical memory. Doing this saves system memory and reduces swapping.
2. When the functions in a DLL change, the applications that use them do not need to be recompiled or relinked as long as the function arguments, calling conventions, and return values do not change. In contrast, statically linked object code requires that the application be relinked when the functions change.
3. A DLL can provide after-market support. For example, a display driver DLL can be modified to support a display that was not available when the application was initially shipped.
4. Programs written in different programming languages can call the same DLL function as long as the programs follow the same calling convention that the function uses. The calling convention (such as C, Pascal, or standard call) controls the order in which the calling function must push the arguments onto the stack, whether the function or the calling function is responsible for cleaning up the stack, and whether any arguments are
passed in registers. For more information, see the documentation included with your compiler.

A potential disadvantage to using DLLs is that the application is not self-contained; it depends on the existence of a separate DLL module. The system terminates processes using load-time dynamic linking if they require a DLL that is not found at process startup and gives an error message to the user. The system does not terminate a process using run-time dynamic linking in this situation, but functions exported by the missing DLL are not available to the program.

**Dynamic-Link Library Creation**

To create a Dynamic-Link Library (DLL), you must create one or more source code files, and possibly a linker file for exporting the functions. If you plan to allow applications that use your DLL to use load-time dynamic linking, you must also create an import library.

**Creating Source Files**

The source files for a DLL contain exported functions and data, internal functions and data, and an optional entry-point function for the DLL. You may use any development tools that support the creation of Windows-based DLLs. If your DLL may be used by a multithreaded application, you should make your DLL "thread-safe". You must synchronize access to all of the DLL's global data to avoid data corruption. You must also ensure that you link only with libraries that are thread-safe as well. For example, Microsoft Visual C++ contains multiple versions of the C Run-time Library, one that is not thread-safe and two that are.

**Exporting Functions**

How you specify which functions in a DLL should be exported depends on the tools that you are using for development. Some compilers allow you to export a function directly in the source code by using a modifier in the function declaration. Other times, you must specify exports in a file that you pass to the linker.

For example, using Visual C++, there are two possible ways to export DLL functions: with the `declspec(dllexport)` modifier or with a module-definition (.def) file. If you use the `declspec(dllexport)` modifier, it is not necessary to use a .def file.

**Creating an Import Library**
An import library (.lib) file contains information the linker needs to resolve external references to exported DLL functions, so the system can locate the specified DLL and exported DLL functions at run time. You can create an import library for your DLL when you build your DLL.

**Using an Import Library**

For example, to call the `CreateWindow()` function, you must link your code with the import library `User32.lib`. The reason is that `CreateWindow()` resides in a system DLL named `User32.dll`, and `User32.lib` is the import library used to resolve the calls to exported functions in `User32.dll` in your code. The linker creates a table that contains the address of each function call. Calls to functions in a DLL will be fixed up **when the DLL is loaded**. While the system is initializing the process, it loads `User32.dll` because the process depends on exported functions in that DLL, and it updates the entries in the function address table. All calls to `CreateWindow()` invoke the function exported from `User32.dll`.

**Dynamic-Link Library Entry-Point Function**

A DLL can optionally specify an entry-point function. If present, the system calls the entry-point function whenever a process or thread loads or unloads the DLL. It can be used to perform simple initialization and cleanup tasks. For example, it can set up thread local storage when a new thread is created, and clean it up when the thread is terminated.

If you are linking your DLL with the C run-time library, it may provide an entry-point function for you, and allow you to provide a separate initialization function. Check the documentation for your run-time library for more information.

If you are providing your own entry-point, see the `DllMain` function. The name `DllMain` is a placeholder for a user-defined function. You must specify the actual name you use when you build your DLL.

**Calling the Entry-Point Function**

The system calls the entry-point function whenever any one of the following events occurs:

1. A process loads the DLL. For processes using load-time dynamic linking, the DLL is loaded during process initialization. For processes using run-time linking, the DLL is loaded before `LoadLibrary()` or `LoadLibraryEx()` returns.
2. A process unloads the DLL. The DLL is unloaded when the process terminates or calls the `FreeLibrary()` function and the reference count becomes zero. If the process terminates as a result of the `TerminateProcess()` or `TerminateThread()` function, the system does not call the DLL entry-point function.
3. A new thread is created in a process that has loaded the DLL. You can use the DisableThreadLibraryCalls() function to disable notification when threads are created.

4. A thread of a process that has loaded the DLL terminates normally, not using TerminateThread() or TerminateProcess(). When a process unloads the DLL, the entry-point function is called only once for the entire process, rather than once for each existing thread of the process. You can use DisableThreadLibraryCalls() to disable notification when threads are terminated.

Only one thread at a time can call the entry-point function. The system calls the entry-point function in the context of the process or thread that caused the function to be called. This allows a DLL to use its entry-point function for allocating memory in the virtual address space of the calling process or to open handles accessible to the process. The entry-point function can also allocate memory that is private to a new thread by using thread local storage (TLS).

**Entry-Point Function Definition**

The DLL entry-point function must be declared with the **standard-call calling convention**. If the DLL entry point is not declared correctly, the DLL is not loaded, and the system displays a message indicating that the DLL entry point must be declared with WINAPI.

In the body of the function, you may handle any combination of the following scenarios in which the DLL entry point has been called:

1. A process loads the DLL (DLL_PROCESS_ATTACH).
2. The current process creates a new thread (DLL_THREAD_ATTACH).
3. A thread exits normally (DLL_THREAD_DETACH).
4. A process unloads the DLL (DLL_PROCESS_DETACH).

The entry-point function should perform only simple initialization tasks. It must not call the LoadLibrary() or LoadLibraryEx() function (or a function that calls these functions), because this may create dependency loops in the DLL load order. This can result in a DLL being used before the system has executed its initialization code. Similarly, the entry-point function must not call the FreeLibrary() function (or a function that calls FreeLibrary()) during process termination, because this can result in a DLL being used after the system has executed its termination code. Because Kernel32.dll is guaranteed to be loaded in the process address space when the entry-point function is called, calling functions in Kernel32.dll does not result in the DLL being used before its initialization code has been executed. Therefore, the entry-point function can create synchronization objects such as critical sections and mutexes, and use TLS, because these functions are located in Kernel32.dll. It is not safe to call the registry functions, for example, because they are located in Advapi32.dll.
For Windows 2000, do not create a named synchronization object in DllMain() because the system will then load an additional DLL. This restriction does not apply to subsequent versions of Windows.

Calling other functions may result in problems that are difficult to diagnose. For example, calling User, Shell, and COM functions can cause access violation errors, because some functions in their DLLs call LoadLibrary() to load other system components. Conversely, calling those functions during termination can cause access violation errors because the corresponding component may already have been unloaded or uninitialized. The following code snippet example demonstrates how to structure the DLL entry-point function.

```c
BOOL WINAPI DllMain(
    HINSTANCE hinstDLL,     // handle to DLL module
    DWORD fdwReason,        // reason for calling function
    LPVOID lpReserved )     // 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.
}
```

**EntryPoint Function Return Value**

When a DLL entry-point function is called because a process is loading, the function returns TRUE to indicate success. For processes using load-time linking, a return value of FALSE causes the process initialization to fail and the process terminates. For processes using run-time linking, a return value of FALSE causes the LoadLibrary() or LoadLibraryEx() function to return NULL, indicating failure. (The system immediately calls your entry-point function with DLL_PROCESS_DETACH and unloads the DLL.) The return value of the entry-point function is disregarded when the function is called for any other reason.
**Load-Time Dynamic Linking**

When the system starts a program that uses load-time dynamic linking, it uses the information the linker placed in the file to locate the names of the DLLs that are used by the process. The system then searches for the DLLs. If the system cannot locate a required DLL, it terminates the process and displays a dialog box that reports the error to the user. Otherwise, the system maps the DLL into the virtual address space of the process and increments the DLL reference count. The system calls the entry-point function. The function receives a code indicating that the process is loading the DLL. If the entry-point function does not return TRUE, the system terminates the process and reports the error. Finally, the system modifies the function address table with the starting addresses for the imported DLL functions. The DLL is mapped into the virtual address space of the process during its initialization and is loaded into physical memory only when needed.

**Run-Time Dynamic Linking**

When the application calls the LoadLibrary() or LoadLibraryEx() functions, the system attempts to locate the DLL. If the search succeeds, the system maps the DLL module into the virtual address space of the process and increments the reference count. If the call to LoadLibrary() or LoadLibraryEx() specifies a DLL whose code is already mapped into the virtual address space of the calling process, the function simply returns a handle to the DLL and increments the DLL reference count. Note that two DLLs that have the same base file name and extension but are found in different directories are not considered to be the same DLL. The system calls the entry-point function in the context of the thread that called LoadLibrary() or LoadLibraryEx(). The entry-point function is not called if the DLL was already loaded by the process through a call to LoadLibrary() or LoadLibraryEx() with no corresponding call to the FreeLibrary() function. If the system cannot find the DLL or if the entry-point function returns FALSE, LoadLibrary() or LoadLibraryEx() returns NULL. If LoadLibrary() or LoadLibraryEx() succeeds, it returns a handle to the DLL module. The process can use this handle to identify the DLL in a call to the GetProcAddress(), FreeLibrary(), or FreeLibraryAndExitThread() function. The GetModuleHandle() function returns a handle used in GetProcAddress(), FreeLibrary(), or FreeLibraryAndExitThread(). The GetModuleHandle() function succeeds only if the DLL module is already mapped into the address space of the process by load-time linking or by a previous call to LoadLibrary() or LoadLibraryEx(). Unlike LoadLibrary() or LoadLibraryEx(), GetModuleHandle() does not increment the module reference count. The GetModuleFileName()
function retrieves the full path of the module associated with a handle returned by
GetModuleHandle(), LoadLibrary(), or LoadLibraryEx().

The process can use GetProcAddress() to get the address of an exported function in the DLL
using a DLL module handle returned by LoadLibrary() or LoadLibraryEx, GetModuleHandle.

When the DLL module is no longer needed, the process can call FreeLibrary() or
FreeLibraryAndExitThread(). These functions decrement the module reference count and unmap
the DLL code from the virtual address space of the process if the reference count is zero.

Run-time dynamic linking enables the process to continue running even if a DLL is not
available. The process can then use an alternate method to accomplish its objective. For example,
if a process is unable to locate one DLL, it can try to use another, or it can notify the user of an
error. If the user can provide the full path of the missing DLL, the process can use this
information to load the DLL even though it is not in the normal search path. This situation
contrasts with load-time linking, in which the system simply terminates the process if it cannot
find the DLL.

Run-time dynamic linking can cause problems if the DLL uses the DllMain() function to perform
initialization for each thread of a process, because the entry-point is not called for threads that
existed before LoadLibrary() or LoadLibraryEx() is called.

**Dynamic-Link Library Search Order**

A system can contain multiple versions of the same dynamic-link library (DLL). Applications
can control the location from which a DLL is loaded by specifying a full path, using DLL
redirection, or by using a manifest. If none of these methods are used, the system searches for
the DLL at load time as described in this topic.

**Standard Search Order**

The DLL search order used by the system depends on whether safe DLL search mode is
enabled or disabled.

Safe DLL search mode is enabled by default. To disable this feature, create the
HKLM\System\CurrentControlSet\Control\Session
Manager\SafeDllSearchMode registry value and set it to 0.
Calling the `SetDllDirectory()` function effectively disables SafeDllSearchMode while the specified directory is in the search path and changes the search order as described in this topic.

Windows XP and Windows 2000 with SP4: Safe DLL search mode is disabled by default. To enable this feature, create the SafeDllSearchMode registry value and set it to 1. Safe DLL search mode is enabled by default starting with Windows XP with Service Pack 2 (SP2).

For Windows 2000, the SafeDllSearchMode value is not supported. The DLL search order is identical to the search order that occurs when safe DLL search mode is disabled. The SafeDllSearchMode value is supported starting with Windows 2000 with SP4.

If SafeDllSearchMode is enabled, the search order is as follows:

1. The directory from which the application loaded.
2. The system directory. Use the `GetSystemDirectory()` function to get the path of this directory.
3. The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched.
4. The Windows directory. Use the `GetWindowsDirectory()` function to get the path of this directory.
5. The current directory.
6. The directories that are listed in the PATH environment variable. Note that this does not include the per-application path specified by the App Paths registry key. The App Paths key is not used when computing the DLL search path.
If SafeDllSearchMode is disabled, the search order is as follows:

1. The directory from which the application loaded.
2. The current directory.
3. The system directory. Use the GetSystemDirectory() function to get the path of this directory.
4. The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched.
5. The Windows directory. Use the GetWindowsDirectory() function to get the path of this directory.
6. The directories that are listed in the PATH environment variable. Note that this does not include the per-application path specified by the App Paths registry key. The App Paths key is not used when computing the DLL search path.

**Alternate Search Order**

The standard search order used by the system can be changed by calling the LoadLibraryEx() function with LOAD_WITH_ALTERED_SEARCH_PATH. The standard search order can also be changed by calling the SetDllDirectory() function.

Windows XP: Changing the standard search order by calling SetDllDirectory() is not supported until Windows XP with Service Pack 1 (SP1).

Windows 2000: Changing the standard search order by calling SetDllDirectory() is not supported.

If you specify an alternate search strategy, its behavior continues until all associated executable modules have been located. After the system starts processing DLL initialization routines, the system reverts to the standard search strategy.

The LoadLibraryEx() function supports an alternate search order if the call specifies LOAD_WITH_ALTERED_SEARCH_PATH and the lpFileName parameter specifies an absolute path.

Note that the standard search strategy and the alternate search strategy specified by LoadLibraryEx() with LOAD_WITH_ALTERED_SEARCH_PATH differ in just one way: The standard search begins in the calling application's directory, and the alternate search begins in the directory of the executable module that LoadLibraryEx() is loading. If SafeDllSearchMode is enabled, the alternate search order is as follows:

1. The directory specified by lpFileName.
2. The system directory. Use the GetSystemDirectory() function to get the path of this directory.
3. The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched.
4. The Windows directory. Use the GetWindowsDirectory() function to get the path of this directory.
5. The current directory.
6. The directories that are listed in the PATH environment variable. Note that this does not include the per-application path specified by the App Paths registry key. The App Paths key is not used when computing the DLL search path.

If SafeDllSearchMode is disabled, the alternate search order is as follows:

1. The directory specified by lpFileName.
2. The current directory.
3. The system directory. Use the GetSystemDirectory() function to get the path of this directory.
4. The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched.
5. The Windows directory. Use the GetWindowsDirectory() function to get the path of this directory.
6. The directories that are listed in the PATH environment variable. Note that this does not include the per-application path specified by the App Paths registry key. The App Paths key is not used when computing the DLL search path.

The SetDllDirectory() function supports an alternate search order if the lpPathName parameter specifies a path. The alternate search order is as follows:

1. The directory from which the application loaded.
2. The directory specified by lpPathName.
3. The system directory. Use the GetSystemDirectory() function to get the path of this directory. The name of this directory is System32.
4. The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched. The name of this directory is System.
5. The Windows directory. Use the GetWindowsDirectory() function to get the path of this directory.
6. The directories that are listed in the PATH environment variable. Note that this does not include the per-application path specified by the App Paths registry key. The App Paths key is not used when computing the DLL search path.
If the lpPathName parameter is an empty string, the call removes the current directory from the search order. SetDllDirectory() effectively disables safe DLL search mode while the specified directory is in the search path. To restore safe DLL search mode based on the SafeDllSearchMode registry value and restore the current directory to the search order, call SetDllDirectory() with lpPathName as NULL.

Dynamic-Link Library Data

A Dynamic-Link Library (DLL) can contain global data or local data.

Variable Scope

The default scope of DLL variables is the same as that of variables declared in the application. Global variables in a DLL source code file are global to each process using the DLL. Static variables have scope limited to the block in which they are declared. As a result, each process has its own instance of the DLL global and static variables by default. Note that your development tools may allow you to override the default behavior. For example, the Visual C++ compiler supports #pragma section and the linker supports the /SECTION option. For more information, see the documentation included with your development tools.

Dynamic Memory Allocation

When a DLL allocates memory using any of the memory allocation functions (GlobalAlloc(), LocalAlloc(), HeapAlloc(), and VirtualAlloc()), the memory is allocated in the virtual address space of the calling process and is accessible only to the threads of that process. A DLL can use file mapping to allocate memory that can be shared among processes.

Thread Local Storage

The thread local storage (TLS) functions enable a DLL to allocate an index for storing and retrieving a different value for each thread of a multithreaded process. For example, a spreadsheet application can create a new instance of the same thread each time the user opens a new spreadsheet. A DLL providing the functions for various spreadsheet operations can use TLS to save information about the current state of each spreadsheet (row, column, and so on). Windows Server 2003 and Windows XP: The Visual C++ compiler supports a syntax that enables you to declare thread-local variables: _declspec(thread). If you use this syntax in a DLL, you will not be able to load the DLL explicitly using LoadLibrary() or LoadLibraryEx().
on versions of Windows prior to Windows Vista. If your DLL will be loaded explicitly, you must use the thread local storage functions instead of _declspec(thread).

Dynamic-Link Library Redirection

Applications can depend on a specific version of a shared DLL and start to fail if another application is installed with a newer or older version of the same DLL. There are two ways to ensure that your application uses the correct DLL: DLL redirection and side-by-side components. Developers and administrators should use DLL redirection for existing applications, because it does not require any changes to the application. If you are creating a new application or updating an application and want to isolate your application from potential problems, create a side-by-side component.

To use DLL redirection, create a redirection file for your application. The redirection file must be named as follows: App_name.local. For example, if the application name is Editor.exe, the redirection file should be named Editor.exe.local. You must install the .local file in the application directory. You must also install the DLLs in the application directory. The contents of a redirection file are ignored, but its presence causes Windows to check the application directory first whenever it loads a DLL, regardless of the path specified to LoadLibrary() or LoadLibraryEx(). If the DLL is not found in the application directory, then these functions use their usual search order. For example, if the application c:\myapp\myapp.exe calls LoadLibrary() using the following path:

c:\\program files\\common files\\system\\mydll.dll

And, if both c:\myapp\myapp.exe.local and c:\myapp\mydll.dll exist, LoadLibrary() loads c:\myapp\mydll.dll. Otherwise, LoadLibrary() loads c:\program files\common files\system\mydll.dll.

Alternatively, if a directory named c:\myapp\myapp.exe.local exists and contains mydll.dll, LoadLibrary() loads c:\myapp\myapp.exe.local\mydll.dll. Known DLLs cannot be redirected. For a list of known DLLs, see the following registry key:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs.
The system uses Windows File Protection to ensure that system DLLs such as these are not updated or deleted except by operating system updates such as service packs.

For Windows 2000, it is known DLLs can be redirected. If the application has a manifest, then any .local files are ignored. Manifests do not affect DLL redirection in Windows 2000. If you are using DLL redirection and the application does not have access to all drives and directories in the search order, LoadLibrary stops searching as soon as access is denied. (If you are not using DLL redirection, LoadLibrary skips directories that it cannot access and then continues searching.)

**It is good practice to install application DLLs in the same directory that contains the application**, even if you are not using DLL redirection. This ensures that installing the application does not overwrite other copies of the DLL and cause other applications to fail. Also, if you follow this good practice, other applications do not overwrite your copy of the DLL and cause your application to fail.

**Dynamic-Link Library Updates**
It is sometimes necessary to replace a DLL with a newer version. Before replacing a DLL, perform a version check to ensure that you are replacing an older version with a newer version. It is possible to replace a DLL that is in use. The method you use to replace DLLs that are in use depends on the operating system you are using. On Windows XP and later, applications should use **Isolated Applications** and **Side-by-side Assemblies**. It is not necessary to restart the computer if you perform the following steps:

1. Use the `MoveFileEx()` function to rename the DLL being replaced. Do not specify `MOVEFILE_COPY_ALLOWED`, and make sure the renamed file is on the same volume that contains the original file. You could also simply rename the file in the same directory by giving it a different extension.
2. Copy the new DLL to the directory that contains the renamed DLL. All applications will now use the new DLL.
3. Use `MoveFileEx()` with `MOVEFILE_DELAY_UNTIL_REBOOT` to delete the renamed DLL.

Before you make this replacement, applications will use the original DLL until it is unloaded. After you make the replacement, applications will use the new DLL. When you write a DLL, you must be careful to ensure that it is prepared for this situation, especially if the DLL maintains global state information or communicates with other services. If the DLL is not prepared for a change in global state information or communication protocols, updating the DLL will require you to restart the computer to ensure that all applications are using the same version of the DLL.

**Student Worksheet**

**Using Dynamic-Link Libraries: Program Examples**

The following examples demonstrate how to create and use a DLL:

1. Creating a simple dynamic-link library
2. Using load-time dynamic linking
3. Using run-time dynamic linking
4. Using shared memory in a dynamic-link library
5. Using thread local storage in a dynamic-link library

**Creating a Simple Dynamic-Link Library Example**

The following example is the source code needed to create a simple DLL, **Myputs.dll**. It defines a simple string-printing function called **myPuts()**. The Myputs DLL does not define an entry-
point function, because it is linked with the **C run-time library** and has no initialization or cleanup functions of its own to perform. To build the DLL, follow the directions in the documentation included with your development tools.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed. Click the OK button.

Click the Next button.
Select the DLL radio button for the Application type: and Empty project for the Additional options:. Click the Finish button.
Then, add the source file and give it a suitable name.
Next, add the following source code.

```c
// The myPuts function writes a null-terminated string to
// the standard output device.

// The export mechanism used here is the __declspec(export)
// method supported by Microsoft Visual Studio, but any
// other export method supported by your development
// environment may be substituted.
#include <windows.h>
#include <stdio.h>
declare EOF (-1)

#ifdef __cplusplus
extern "C"
{
    // we need to export the C interface
#endif

__declspec(dllexport) int myPuts(LPWSTR lpzMsg)
{
    DWORD cchWritten;
    HANDLE hConout;
    BOOL fRet;

    // Get a handle to the console output device.
    hConout = CreateFileW(L"CONOUT$",
        GENERIC_WRITE,
        ...)
FILE_SHARE_WRITE,
NULL,
OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL,
NULL);

if (INVALID_HANDLE_VALUE == hConout)
    return EOF;

wprintf(L"I'm in DLL lor, with myPuts() that can be exported!\n");

wprintf(L"I'm displaying a text sent to me by executable...\n");

wprintf(L"\n");
// Write a null-terminated string to the console output device

while (*lpszMsg != L'\0')
    { 
        fRet = WriteConsole(hConout, lpszMsg, 1, &cchWritten, NULL);
        if ( (fRet == FALSE) || (cchWritten != 1) )
            return EOF;
        lpszMsg++;
    }

wprintf(L"\n");

return 1;
}
#endif

Build and run the project. The following screenshot is a sample window, asking the executable (application) to be run (at this stage, we do not have the executable yet, that can import or use the function in the DLL). Just dismiss the window.

Under the project's Debug folder, the DLL and lib files should be generated and ready to be used.
Using Load-Time Dynamic Linking Example

After you have created a DLL, you can use the functions it defines in an application. The following is a simple console application that uses the myPuts() function exported from Myputs.dll (or imported by the application).

Because this example calls the DLL function explicitly, the module for the application must be linked with the import library Myputs.lib. For other compilers, to build DLLs, see the documentation included with your development tools.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.

Select an empty console mode application project. Click the Finish button.
Then, add the source file and give it a suitable name.
Next, add the following source code.

```c
#include <windows.h>
#include <stdio.h>

// a function from a DLL
extern "C" int myPuts(LPWSTR);

int main(void)
{
    int Ret = 1;

    wprintf(L"In the executable...\n");

    Ret = myPuts(L"Message sent to the DLL function..." );
    wprintf(L"Back to the executable, Ret = %d\n", Ret);
    return Ret;
}
```

Using Visual Studio IDE, we have two ways to tell the compiler where to find the library file (.lib). The first one is the Visual C++ project wide. In this case invoke the Tools > Options menu.
Select the Visual C++ Directories subfolder for the Projects and Solutions folder. For Show directories for: drop down box, select Library files.
Then add the library folder.
Another way telling the compiler where to find the library file is through the project’s setting. While the project was opened, invoke the project properties page.

Under the Linker folder, select the General link. For the Additional Library Directories field add the path to the library/dll.
Then, we need to tell the compiler the library file name that needs to be linked. Under the Linker folder, select the Input link. For the Additional Dependencies field, add the library file name. Click OK.
Next, we need to copy the DLL file to the application’s Debug folder (for this case).
Well, now we are ready to run the project. Run the application project.

The following screenshot shows a sample output which indicates the function in the DLL (can be exported) was successfully 'imported' by the application program.

Using Run-Time Dynamic Linking Program Example

You can use the same DLL in both load-time and run-time dynamic linking. The following example uses the LoadLibrary() function to get a handle to the Myputs DLL. If LoadLibrary() succeeds, the program uses the returned handle in the GetProcAddress() function to get the address of the DLL's myPuts function. After calling the DLL function, the program calls the FreeLibrary() function to unload the DLL.
Because the program uses run-time dynamic linking, it is not necessary to link the module with an import library for the DLL.
This example illustrates an important difference between run-time and load-time dynamic linking. If the DLL is not available, the application using load-time dynamic linking must simply terminate. The run-time dynamic linking example, however, can respond to the error.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.

![New Project dialog box](image)

Then, add the source file and give it a suitable name.
Next, add the following source code.

```c
// A simple program that uses LoadLibrary() and 
// GetProcAddress() to access myPuts() from Myputs.dll.
#include <windows.h>
#include <stdio.h>

typedef int (*MYPROC)(LPWSTR);

int wmain(int argc, WCHAR **argv)
{
    HINSTANCE hinstLib;
    MYPROC ProcAdd;
    BOOL fFreeResult, fRunTimeLinkSuccess = FALSE;

    // Get a handle to the DLL module.
    hinstLib = LoadLibrary(L"MyTestDll.dll");

    // If the handle is valid, try to get the function address.
    if (hinstLib != NULL)
    {
        wprintf(L"LoadLibrary() is OK!\n");
        ProcAdd = (MYPROC) GetProcAddress(hinstLib, "myPuts");

        // If the function address is valid, call the function.
        if (ProcAdd != NULL)
        {
            wprintf(L"GetProcAddress() is fine!\n");
        }
    }
}
```
fRunTimeLinkSuccess = TRUE;
(ProcAdd) (L"Another message from executable lol!");
}
else
    wprintf(L"GetProcAddress() failed miserably, error %d\n", GetLastError());
    // Free the DLL module.
    fFreeResult = FreeLibrary(hinstLib);
} 
else
    wprintf(L"LoadLibrary() failed miserably, error %d\n", GetLastError());

    // If unable to call the DLL function, use an alternative.
    if (!fRunTimeLinkSuccess)
        printf("Failed to call the DLL's function! Error %d\n", GetLastError());

        return 0;
}

Build the project. There should be no error because the LoadLibrary() happens during the run-time. Copy the previously created DLL, MyTestDll.dll into the project’s Debug folder.

Run the application project. The following sample output shows the application successfully 'imported' the DLL's function.

Using Shared Memory in a Dynamic-Link Library Example
The following example demonstrates how the DLL entry-point function can use a file-mapping object to set up memory that can be shared by processes that load the DLL. The shared DLL memory persists only as long as the DLL is loaded. Applications can use the SetSharedMem() and GetSharedMem() functions to access the shared memory.

**DLL that Implements the Shared Memory**

The example uses file mapping to map a block of named shared memory into the virtual address space of each process that loads the DLL. To do this, the entry-point function must:

1. Call the CreateFileMapping() function to get a handle to a file-mapping object. The first process that loads the DLL creates the file-mapping object. Subsequent processes open a handle to the existing object.
2. Call the MapViewOfFile() function to map a view into the virtual address space. This enables the process to access the shared memory.

Note that while you can specify default security attributes by passing in a NULL value for the lpAttributes parameter of CreateFileMapping(), you may choose to use a SECURITY_ATTRIBUTES structure to provide additional security.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
Then, add the source file and give it a suitable name.
Next, add the following source code.

```c
// The DLL code
#include <windows.h>
#include <memory.h>

#define SHMEMSIZE 4096

static LPVOID lpvMem = NULL; // pointer to shared memory
static HANDLE hMapObject = NULL; // handle to file mapping

// The DLL entry-point function sets up shared memory using a named file-mapping object.

BOOL WINAPI DllMain(HINSTANCE hinstDLL, // DLL module handle
                     DWORD fdwReason,    // reason called
                     LPVOID lpvReserved) // reserved
{  
    BOOL fInit, fIgnore;

    switch (fdwReason)
    {  
        // DLL load due to process initialization or LoadLibrary
        case DLL_PROCESS_ATTACH:
            // Create a named file mapping object
```
hMapObject = CreateFileMapping(
    INVALID_HANDLE_VALUE,   // use paging file
    NULL,                   // default security attributes
    PAGE_READWRITE,         // read/write access
    0,                      // size: high 32-bits
    SHMEMSIZE,              // size: low 32-bits
    L"dllmemfilemap");     // name of map object
if (hMapObject == NULL)
    return FALSE;
// The first process to attach initializes memory
fInit = (GetLastError() != ERROR_ALREADY_EXISTS);

// Get a pointer to the file-mapped shared memory
lpvMem = MapViewOfFile(
    hMapObject,          // object to map view of
    FILE_MAP_WRITE,      // read/write access
    0,                   // high offset: map from
    0,                   // low offset: beginning
    0);                  // default: map entire file
if (lpvMem == NULL)
    return FALSE;
// Initialize memory if this is the first process
if (fInit)
    memset(lpvMem, '\0', SHMEMSIZE);

break;

// The attached process creates a new thread
case DLL_THREAD_ATTACH:
    break;

// The thread of the attached process terminates
case DLL_THREAD_DETACH:
    break;

// DLL unload due to process termination or FreeLibrary
case DLL_PROCESS_DETACH:

    // Unmap shared memory from the process's address space
    fIgnore = UnmapViewOfFile(lpvMem);

    // Close the process's handle to the file-mapping object
    fIgnore = CloseHandle(hMapObject);

    break;

default:
    break;
}
return TRUE;
UNREFERENCED_PARAMETER(hinstDLL);
UNREFERENCED_PARAMETER(lpvReserved);
// The export mechanism used here is the __declspec(export)
// method supported by Microsoft Visual Studio, but any
// other export method supported by your development
// environment may be substituted.
#ifdef __cplusplus
// If used by C++ code,
extern "C" {
    // we need to export the C interface
#endif

// SetSharedMem sets the contents of the shared memory
__declspec(dllexport) void SetSharedMem(LPWSTR lpszBuf)
{
    LPWSTR lpszTmp;
    DWORD dwCount=1;

    // Get the address of the shared memory block
    lpszTmp = (LPWSTR) lpvMem;

    // Copy the null-terminated string into shared memory
    while (*lpszBuf && dwCount<SHMEMSIZE)
    {
        *lpszTmp++ = *lpszBuf++;
        dwCount++;
    }
    *lpszTmp = '\0';
}

// GetSharedMem gets the contents of the shared memory
__declspec(dllexport) void GetSharedMem(LPWSTR lpszBuf, DWORD cchSize)
{
    LPWSTR lpszTmp;

    // Get the address of the shared memory block
    lpszTmp = (LPWSTR) lpvMem;

    // Copy from shared memory into the caller's buffer
    while (*lpszTmp && --cchSize)
    {
        *lpszBuf++ = *lpszTmp++;
        *lpszBuf = '\0';
    }
#ifdef __cplusplus
}
#endif

Build and run the project. The generated DLL is ready to be used as shown in the following screenshot.
Shared memory can be mapped to a different address in each process. For this reason, each process has its own instance of lpvMem, which is declared as a global variable so that it is available to all DLL functions. The example assumes that the DLL global data is not shared, so each process that loads the DLL has its own instance of lpvMem. Note that the shared memory is released when the last handle to the file-mapping object is closed. To create persistent shared memory, you would need to ensure that some process always has an open handle to the file-mapping object.

Processes that Use the Shared Memory Program Example

The following processes use the shared memory provided by the DLL defined above. The first process calls SetSharedMem() to write a string while the second process calls GetSharedMem() to retrieve this string.

This process uses the SetSharedMem() function implemented by the DLL to write the string "This is a test string" to the shared memory. It also starts a child process that will read the string from the shared memory.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
### New Project

#### Project types:
- Visual C++
  - ATL
  - CLR
  - General
  - MFC
  - Smart Device
  - Test
  - Win32
- Other Languages
- Other Project Types
- Test Projects

#### Templates:
- .NET Framework 3.5

#### Visual Studio installed templates
- Win32 Console Application
- Win32 Project

#### My Templates
- Search Online Templates...

A project for creating a Win32 console application

<table>
<thead>
<tr>
<th>Name</th>
<th>UseSharedMemDll</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>\anadir</td>
</tr>
<tr>
<td>Solution Name</td>
<td>UseSharedMemDll</td>
</tr>
</tbody>
</table>

- OK
- Cancel
Then, add the source file and give it a suitable name.
Next, add the following source code.

```c
// Parent process
#include <windows.h>
#include <stdio.h>

extern "C" void SetSharedMem(LPWSTR lpszBuf);

HANDLE CreateChildProcess(LPTSTR szCmdline)
{
    PROCESS_INFORMATION piProcInfo;
    STARTUPINFO siStartInfo;
    BOOL bFuncRetn = FALSE;

    // Set up members of the PROCESS_INFORMATION structure.
    ZeroMemory( &piProcInfo, sizeof(PROCESS_INFORMATION) );

    // Set up members of the STARTUPINFO structure.
    ZeroMemory( &siStartInfo, sizeof(STARTUPINFO) );
    siStartInfo.cb = sizeof(STARTUPINFO);

    // Create the child process.
    bFuncRetn = CreateProcess(NULL, 
        szCmdline,  // command line
        NULL,       // process security attributes
        NULL,       // primary thread security attributes
```
TRUE,       // handles are inherited
0,          // creation flags
NULL,       // use parent's environment
NULL,       // use parent's current directory
&siStartInfo, // STARTUPINFO pointer
&piProcInfo); // receives PROCESS_INFORMATION

if (bFuncRetn == 0)
{
    wprintf(L"CreateProcess() failed, error %d\n", GetLastError());
    return INVALID_HANDLE_VALUE;
} else
{
    CloseHandle(piProcInfo.hThread);
    return piProcInfo.hProcess;
}

void wmain(int argc, WCHAR *argv[])
{
    HANDLE hProcess;

    if (argc == 1)
    {
        wprintf(L"Please specify an input file\n");
        ExitProcess(0);
    }

    // Call the DLL function
    wprintf(L"\nProcess is writing to shared memory...\n\n");
    SetSharedMem(L"This is a test string");

    // Start the child process that will read the memory
    hProcess = CreateChildProcess(argv[1]);

    // Ensure this process is around until the child process terminates
    if (INVALID_HANDLE_VALUE != hProcess)
    {
        WaitForSingleObject(hProcess, INFINITE);
        CloseHandle(hProcess);
    }
}

Before building this program we need to tell the compiler where to find the SharedMemDll.lib file and a copy of the previously created DLL, SharedMemDll.dll must be available to this project. Set the library location path and the name as done previously through the projects' properties page.
Then, copy the DLL to the project's Debug folder so that the file can be found physically.
Finally, run the project. The following sample outputs show the application run without and with an argument.

The following is another program example that supposed to use the shared memory. The process uses the GetSharedMem() function implemented by the DLL to read a string from the shared memory. It is started by the parent process above.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
A project for creating a Win32 console application

Name: UseSharedMemDll2
Location: C:\amad
Solution Name: UseSharedMemDll2

Create directory for solution
Then, add the source file and give it a suitable name.
Next, add the following source code.

```c
// Child process
#include <windows.h>
#include <stdio.h>

extern "C" void GetSharedMem(LPWSTR lpzBuf, DWORD cchSize);

void wmain()
{
    WCHAR cBuf[MAX_PATH];
    GetSharedMem(cBuf, MAX_PATH);
    wprintf(L"Child process read from shared memory: %s\n", cBuf);
}
```

Next add the path to the DLL/library and the library file name.
Build the project.
Then we need to copy the DLL into the project’s Debug folder so that it can be found during the run-time else the following error screen will be displayed.
For the sake of demonstration, copy the DLL into the application Debug’s project folder and re-run the project.

The following is the sample output.

Using Thread Local Storage in a Dynamic-Link Library Program Example
This section shows the use of a DLL entry-point function to set up a thread local storage (TLS) index to provide private storage for each thread of a multithreaded process. The TLS index is stored in a global variable, making it available to all of the DLL functions. This example assumes that the DLL’s global data is not shared, because the TLS index is not necessarily the same for each process that loads the DLL.

The entry-point function uses the TlsAlloc() function to allocate a TLS index whenever a process loads the DLL. Each thread can then use this index to store a pointer to its own block of memory. When the entry-point function is called with the DLL_PROCESS_ATTACH value, the code performs the following actions:

1. Uses the TlsAlloc() function to allocate a TLS index.
2. Allocates a block of memory to be used exclusively by the initial thread of the process.
3. Uses the TLS index in a call to the TlsSetValue() function to store the address of the memory block in the TLS slot associated with the index.

Each time the process creates a new thread, the entry-point function is called with the DLL_THREAD_ATTACH value. The entry-point function then allocates a block of memory for the new thread and stores a pointer to it by using the TLS index.

When a function requires access to the data associated with a TLS index, specify the index in a call to the TlsGetValue() function. This retrieves the contents of the TLS slot for the calling thread, which in this case is a pointer to the memory block for the data. When a process uses load-time linking with this DLL, the entry-point function is sufficient to manage the thread local storage. Problems can occur with a process that uses run-time linking because the entry-point function is not called for threads that exist before the LoadLibrary() function is called, so TLS memory is not allocated for these threads. This example solves this problem by checking the value returned by the TlsGetValue() function and allocating memory if the value indicates that the TLS slot for this thread is not set.

When each thread no longer needs to use a TLS index, it must free the memory whose pointer is stored in the TLS slot. When all threads have finished using a TLS index, use the TlsFree() function to release the index.

When a thread terminates, the entry-point function is called with the DLL_THREAD_DETACH value and the memory for that thread is freed. When a process terminates, the entry-point function is called with the DLL_PROCESS_DETACH value and the memory referenced by the pointer in the TLS index is freed.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
Select the DLL and empty project as shown below.
Then, add the source file and give it a suitable name.
Next, add the following source code.

```c
// The DLL code
#include <windows.h>

static DWORD dwTlsIndex; // address of shared memory

// DllMain() is the entry-point function for this DLL
BOOL WINAPI DllMain(HINSTANCE hinstDLL, // DLL module handle
                    DWORD fdwReason, // reason called
                    LPVOID lpvReserved) // reserved
{
    LPVOID lpvData;
    BOOL fIgnore;

    switch (fdwReason)
    {
        // The DLL is loading due to process
        // initialization or a call to LoadLibrary
        case DLL_PROCESS_ATTACH:
            // Allocate a TLS index
            if ((dwTlsIndex = TlsAlloc()) == TLS_OUT_OF_INDEXES)
                return FALSE;
            // No break: Initialize the index for first thread.
            // The attached process creates a new thread.
            break;
        case DLL_THREAD_ATTACH:
```
// Initialize the TLS index for this thread.
lpvData = (LPVOID) LocalAlloc(LPTR, 256);
if (lpvData != NULL)
    fIgnore = TlsSetValue(dwTlsIndex, lpvData);
break;
// The thread of the attached process terminates.
case DLL_THREAD_DETACH:
// Release the allocated memory for this thread
lpvData = TlsGetValue(dwTlsIndex);
if (lpvData != NULL)
    LocalFree((HLOCAL) lpvData);
break;
// DLL unload due to process termination or FreeLibrary
case DLL_PROCESS_DETACH:
// Release the allocated memory for this thread
lpvData = TlsGetValue(dwTlsIndex);
if (lpvData != NULL)
    LocalFree((HLOCAL) lpvData);
// Release the TLS index
TlsFree(dwTlsIndex);
break;
default:
    break;
}
return TRUE;
UNREFERENCED_PARAMETER(hinstDLL);
UNREFERENCED_PARAMETER(lpvReserved);
}

// The export mechanism used here is the __declspec(export)
// method supported by Microsoft Visual Studio, but any
// other export method supported by your development
// environment may be substituted.
#ifdef __cplusplus
    // If used by C++ code,
extern "C" {}
#else
_declspec(dllexport) BOOL WINAPI StoreData(DWORD dw)
{
    LPVOID lpvData;
    DWORD * pData; // The stored memory pointer

    lpvData = TlsGetValue(dwTlsIndex);
    if (lpvData == NULL)
    {
        lpvData = (LPVOID) LocalAlloc(LPTR, 256);
        if (lpvData == NULL)
            return FALSE;
        if (!TlsSetValue(dwTlsIndex, lpvData))
            return FALSE;
    }
    pData = (DWORD *) lpvData; // Cast to my data type.
    // In this example, it is only a pointer to a DWORD

    return TRUE;
#endif
// but it can be a structure pointer to contain more complicated data.
(*pData) = dw;
return TRUE;
}

__declspec(dllexport) BOOL WINAPI GetData(DWORD *pdw)
{
    LPVOID lpvData;
    DWORD * pData; // The stored memory pointer

    lpvData = TlsGetValue(dwTlsIndex);
    if (lpvData == NULL)
        return FALSE;

    pData = (DWORD *) lpvData;
    (*pdw) = (*pData);
    return TRUE;
}
#endif

Build the project. The DLL and lib files should be generated.

The following application code example demonstrates the use of the DLL functions defined in the previous example.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
Then, add the source file and give it a suitable name.
Next, add the following source code.

```c
#include <windows.h>
#include <stdio.h>

#define THREADCOUNT 4
#define DLL_NAME L"TheTLSDLL"

void ErrorExit(LPSTR);

extern "C" BOOL WINAPI StoreData(DWORD dw);
extern "C" BOOL WINAPI GetData(DWORD *pdw);

DWORD WINAPI ThreadFunc(void)
{
    int i;

    if(!StoreData(GetCurrentThreadId()))
        ErrorExit("StoreData() error");

    for(i=0; i<THREADCOUNT; i++)
    {
        DWORD dwOut;
        if(!GetData(&dwOut))
            ErrorExit("GetData() error");
        if( dwOut != GetCurrentThreadId() )
```
wprintf(L"thread %d: data is incorrect (%d)\n", GetCurrentThreadId(), dwOut);
else
    wprintf(L"thread Id %d: data is correct\n", GetCurrentThreadId());
    Sleep(0);
}

return 0;
}

int main(void)
{
    DWORD IDThread;
    HANDLE hThread[THREADCOUNT];
    int i;
    HMODULE hm;

    // Load the DLL
    hm = LoadLibrary(DLL_NAME);
    if (!hm)
    {
        ErrorExit("DLL (LoadLibrary()) failed to load!\n");
    }
    wprintf(L"LoadLibrary() is fine, the %s DLL was loaded..\n", DLL_NAME);

    // Create multiple threads
    for (i = 0; i < THREADCOUNT; i++)
    {
        hThread[i] = CreateThread(NULL, // default security attributes
            0, // use default stack size
            (LPTHREAD_START_ROUTINE) ThreadFunc, // thread function
            NULL, // no thread function argument
            0, // use default creation flags
            &IDThread); // returns thread identifier

        // Check the return value for success.
        if (hThread[i] == NULL)
            ErrorExit("CreateThread() error!\n");
    }
    WaitForMultipleObjects(THREADCOUNT, hThread, TRUE, INFINITE);
    FreeLibrary(hm);
    return 0;
}

void ErrorExit (LPSTR lpszMessage)
{
    fprintf(stderr, "%s\n", lpszMessage);
    ExitProcess(0);
}

When you build, the following error should be expected.

1>------ Build started: Project: UseTheTLSDLL, Configuration: Debug Win32 ------
1>Compiling...
Add the path to the DLL/lib and the library file name.
Build and run the application project. The following is a sample output.

![Sample Output](image)

Dynamic-Link Library Functions

The following functions are used in dynamic linking.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DisableThreadLibraryCalls()</td>
<td>Disables thread attach and thread detach notifications for the specified DLL.</td>
</tr>
<tr>
<td>DllMain()</td>
<td>An optional entry point into a DLL.</td>
</tr>
<tr>
<td>FreeLibrary()</td>
<td>Decrements the reference count of the loaded DLL. When the reference count reaches zero, the module is unmapped from the address space of the calling process.</td>
</tr>
<tr>
<td>FreeLibraryAndExitThread()</td>
<td>Decrements the reference count of a loaded DLL by one, and then calls ExitThread() to terminate the calling thread.</td>
</tr>
<tr>
<td>GetDllDirectory()</td>
<td>Retrieves the application-specific portion of the search path used to locate DLLs for the application.</td>
</tr>
<tr>
<td>GetModuleFileName()</td>
<td>Retrieves the fully-qualified path for the file containing the specified module.</td>
</tr>
<tr>
<td>GetModuleFileNameEx()</td>
<td>Retrieves the fully-qualified path for the file containing the specified module.</td>
</tr>
<tr>
<td>GetModuleHandle()</td>
<td>Retrieves a module handle for the specified module.</td>
</tr>
<tr>
<td>GetModuleHandleEx()</td>
<td>Retrieves a module handle for the specified module.</td>
</tr>
<tr>
<td>GetProcAddress()</td>
<td>Retrieves the address of an exported function or variable from the specified DLL.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LoadLibrary()</td>
<td>Maps the specified executable module into the address space of the calling process.</td>
</tr>
<tr>
<td>LoadLibraryEx()</td>
<td>Maps the specified executable module into the address space of the calling process.</td>
</tr>
<tr>
<td>SetDllDirectory()</td>
<td>Modifies the search path used to locate DLLs for the application.</td>
</tr>
</tbody>
</table>

**Obsolete Functions**

The LoadModule() function is provided only for compatibility with 16-bit versions of Windows.