

2. The build process constructs a `.LIB` library file, which is a *stub* for the actual code and is linked into the calling program at build time, satisfying the function references. The `.LIB` file contains code that loads the DLL at program load time. It also contains a stub for each function, where the stub calls the DLL. This file should be placed in a common user library directory specified to the project.
3. The build process also constructs a `.DLL` file that contains the executable image. This file is typically placed in the same directory as the application that will use it, and the application loads the DLL during its initialization. The alternative search locations are described in the next section.
4. Take care to export the function interfaces in the DLL source, as described next.

### ***Exporting and Importing Interfaces***

The most significant change required to put a function into a DLL is to declare it to be exportable (UNIX and some other systems do not require this explicit step). This is achieved either by using a `.DEF` file or, more simply, with Microsoft C/C++, by using the `__declspec (dllexport)` storage modifier as follows:

```
__declspec (dllexport) DWORD MyFunction (...);
```

The build process will then create a `.DLL` file and a `.LIB` file. The `.LIB` file is the stub library that should be linked with the calling program to satisfy the external references and to create the actual links to the `.DLL` file at load time.

The calling or *client* program should declare that the function is to be imported by using the `__declspec (dllimport)` storage modifier. A standard technique is to write the include file by using a preprocessor variable created by appending the Microsoft Visual C++ project name, in uppercase letters, with `_EXPORTS`.

One further definition is necessary. If the calling (importing) client program is written in C++, `__cplusplus` is defined, and it is necessary to specify the C calling convention, using:

```
extern "C"
```

For example, if `MyFunction` is defined as part of a DLL build in project `MyLibrary`, the header file would contain:

```
#if defined(MYLIBRARY_EXPORTS)
#define LIBSPEC __declspec (dllexport)
#elif defined(__cplusplus)
#define LIBSPEC extern "C" __declspec (dllimport)
```

```
#else
#define LIBSPEC __declspec (dllimport)
#endif
LIBSPEC DWORD MyFunction (...);
```

Visual C/C++ automatically defines `MYLIBRARY_EXPORTS` when invoking the compiler within the `MyLibrary` DLL project. A client project that uses the DLL does not define `MYLIBRARY_EXPORTS`, so the function name is imported from the library.

When building the calling program, specify the `.LIB` file. When executing the calling program, ensure that the `.DLL` file is available to the calling program; this is frequently done by placing the `.DLL` file in the same directory as the executable. As mentioned previously, there is a set of DLL search rules that specify the order in which Windows searches for the specified `.DLL` file *as well as* for all other DLLs or executables that the specified file requires, stopping with the first instance located. The following default *safe DLL search mode* order is used for both explicit and implicit linking:

- The directory containing the loaded application.
- The system directory. You can determine this path with `GetSystemDirectory`; normally its value is `C:\WINDOWS\SYSTEM32`.
- The 16-bit Windows system directory. There is no function to obtain this path, and it is obsolete for our purposes.
- The Windows directory (`GetWindowsDirectory`).
- The current directory.
- Directories specified by the `PATH` environment variable, in the order in which they occur.

Note that the standard order can be modified, as explained in the “Explicit Linking” section. For some additional detailed information on the search strategy, see MSDN and the `SetDllDirectory` function. `LoadLibraryEx`, described in the next section, also alters the search strategy.

You can also export and import variables as well as function entry points, although the examples do not illustrate this capability.

## Explicit Linking

Explicit or *run-time* linking requires the program to request specifically that a DLL be loaded or freed. Next, the program obtains the address of the required entry point and uses that address as the pointer in the function call. The function

is not declared in the calling program; rather, you declare a variable as a pointer to a function. Therefore, there is no need for a library at link time. The three required functions are `LoadLibrary` (or `LoadLibraryEx`), `GetProcAddress`, and `FreeLibrary`. *Note:* The function definitions show their 16-bit legacy through far pointers and different handle types.

The two functions to load a library are `LoadLibrary` and `LoadLibraryEx`.

```
HMODULE LoadLibrary (LPCTSTR lpLibFileName)
```

```
HMODULE LoadLibraryEx (
    LPCTSTR lpLibFileName,
    HANDLE hFile,
    DWORD dwFlags)
```

In both cases, the returned handle (`HMODULE` rather than `HANDLE`; you may see the equivalent macro, `HINSTANCE`) will be `NULL` on failure. The `.DLL` suffix is not required on the file name. `.EXE` files can also be loaded with the `LoadLibrary` functions. Pathnames must use backslashes (`\`); forward slashes (`/`) will not work. The name is the one in the `.DEF` module definition file (see MSDN for details).

*Note:* If you are using C++ and `__declspec`, the decorated name is exported, and the decorated name is required for `GetProcAddress`. Our examples avoid this difficult problem by using C.

Since DLLs are shared, the system maintains a reference count to each DLL (incremented by the two load functions) so that the actual file does not need to be remapped. Even if the DLL file is found, `LoadLibrary` will fail if the DLL is implicitly linked to other DLLs that cannot be located.

`LoadLibraryEx` is similar to `LoadLibrary` but has several flags that are useful for specifying alternative search paths and loading the library as a data file. The `hFile` parameter is reserved for future use. `dwFlags` can specify alternate behavior with one of three values.

1. `LOAD_WITH_ALTERED_SEARCH_PATH` overrides the previously described standard search order, changing just the first step of the search strategy. The pathname specified as part of `lpLibFileName` is used rather than the directory from which the application was loaded.

2. `LOAD_LIBRARY_AS_DATAFILE` allows the file to be data only, and there is no preparation for execution, such as calling `DllMain` (see the “DLL Entry Point” section later in the chapter).
3. `DONT_RESOLVE_DLL_REFERENCE` means that `DllMain` is not called for process and thread initialization, and additional modules referenced within the DLL are not loaded.

When you’re finished with a DLL instance, possibly to load a different version of the DLL, free the library handle, thereby freeing the resources, including virtual address space, allocated to the library. The DLL will, however, remain loaded if the reference count indicates that other processes are still using it.

```
BOOL FreeLibrary (HMODULE hLibModule)
```

After loading a library and before freeing it, you can obtain the address of any entry point using `GetProcAddress`.

```
FARPROC GetProcAddress (
    HMODULE hModule,
    LPCSTR lpProcName)
```

`hModule` is an instance produced by `LoadLibrary` or `GetModuleHandle` (see the next paragraph). `lpProcName`, which cannot be Unicode, is the entry point name. The return result is `NULL` in case of failure. `FARPROC`, like “long pointer,” is an anachronism.

You can obtain the file name associated with an `hModule` handle using `GetModuleFileName`. Conversely, given a file name (either a `.DLL` or `.EXE` file), `GetModuleHandle` will return the handle, if any, associated with this file if the current process has loaded it.

The next example shows how to use the entry point address to invoke a function.

## Example: Explicitly Linking a File Conversion Function

Program 2–3 is an encryption conversion program that calls the function `cci_f` (Program 2–5) to process the file using file I/O. Program 5–3 (`cciMM`) is an alter-

native function that uses memory mapping to perform exactly the same operation. The circumstances under which `cciMM` is faster were described earlier. Furthermore, if you are running on a 32-bit computer, you will not be able to map files larger than about 1.5GB.

Program 5–7 reimplements the calling program so that it can decide which implementation to load at run time. It then loads the DLL and obtains the address of the `cci_f` entry point and calls the function. There is only one entry point in this case, but it would be equally easy to locate multiple entry points. The main program is as before, except that the DLL to use is a command line parameter. Exercise 5–10 suggests that the DLL should be determined on the basis of system and file characteristics. Also notice how the `FARPROC` address is cast to the appropriate function type using the required, but complex, C syntax. The cast even includes `__cdecl`, the linkage type, which is also used by the DLL function. Therefore, there are no assumptions about the build settings for the calling program (“client”) and called function (“server”).

---

### **Program 5–7** `cciEL`: File Conversion with Explicit Linking

---

```
/* Chapter 5. cci Explicit Link version. */

#include "Everything.h"

int _tmain (int argc, LPCTSTR argv[])
{
    BOOL (__cdecl *cci_f) (LPCTSTR, LPCTSTR, DWORD);
    HMODULE hDLL;
    FARPROC pcci;

    /* Load the cipher function. */
    hDLL = LoadLibrary (argv[4]);

    /* Get the entry point address. */
    pcci = GetProcAddress (hDLL, "cci_f");
    cci_f = (BOOL (__cdecl *) (LPCTSTR, LPCTSTR, DWORD)) pcci;

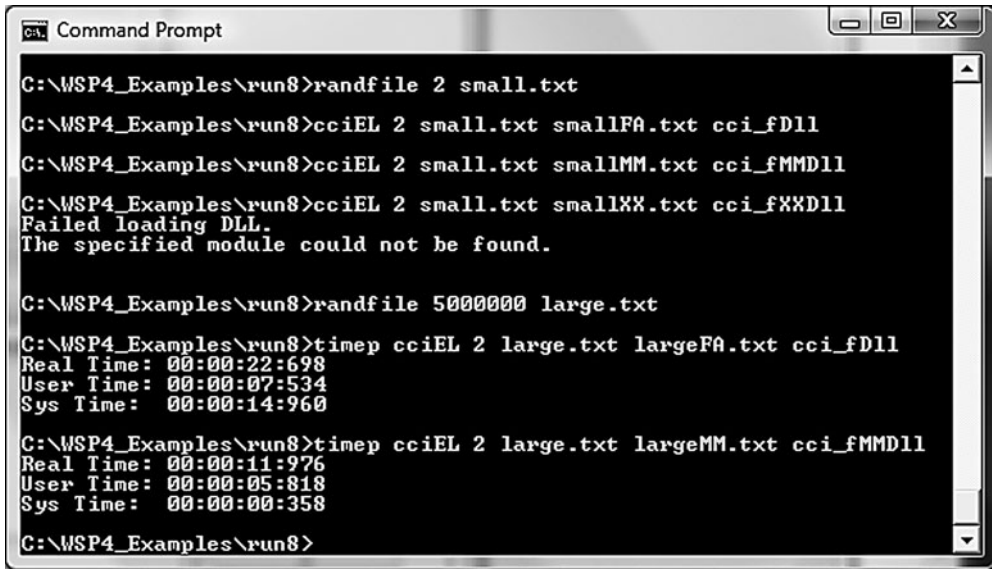
    /* Call the function. */
    if (!cci_f (argv[2], argv[3], atoi(argv[1]) ) ) {
        FreeLibrary (hDLL);
        ReportError (_T ("cci failed."), 6, TRUE);
    }
    FreeLibrary (hDLL);
    return 0;
}
```

---

## Building the cci\_f DLLs

This program was tested with the two file conversion functions, which must be built as DLLs with different names but identical entry points. There is only one entry point in this case. The only significant change in the source code is the addition of a storage modifier, `__declspec (dllexport)`, to export the function.

Run 5-7 shows the results, which are comparable to Run 5-3.



```

C:\WSP4_Examples\run8>randfile 2 small.txt
C:\WSP4_Examples\run8>cciEL 2 small.txt smallFA.txt cci_fDll
C:\WSP4_Examples\run8>cciEL 2 small.txt smallMM.txt cci_fMMDll
C:\WSP4_Examples\run8>cciEL 2 small.txt smallXX.txt cci_fXXDll
Failed loading DLL.
The specified module could not be found.

C:\WSP4_Examples\run8>randfile 5000000 large.txt
C:\WSP4_Examples\run8>timep cciEL 2 large.txt largeFA.txt cci_fDll
Real Time: 00:00:22:698
User Time: 00:00:07:534
Sys Time: 00:00:14:960

C:\WSP4_Examples\run8>timep cciEL 2 large.txt largeMM.txt cci_fMMDll
Real Time: 00:00:11:976
User Time: 00:00:05:818
Sys Time: 00:00:00:358
C:\WSP4_Examples\run8>
  
```

**Run 5-7** cciEL: Explicit Linking to a DLL

## The DLL Entry Point

Optionally, you can specify an entry point for every DLL you create, and this entry point is normally invoked automatically every time a process attaches or detaches the DLL. `LoadLibraryEx`, however, allows you to prevent entry point execution. For implicitly linked (load-time) DLLs, process attachment and detachment occur when the process starts and terminates. In the case of explicitly linked DLLs, `LoadLibrary`, `LoadLibraryEx`, and `FreeLibrary` cause the attachment and detachment calls.

The entry point is also invoked when new threads (Chapter 7) are created or terminated by the process.

The DLL entry point, `DllMain`, is introduced here but will not be fully exploited until Chapter 12 (Program 12-5), where it provides a convenient way for

threads to manage resources and so-called Thread Local Storage (TLS) in a thread-safe DLL.

```

BOOL DllMain (
    HINSTANCE hDll,
    DWORD Reason,
    LPVOID lpReserved)

```

The `hDll` value corresponds to the instance obtained from `LoadLibrary`. `lpReserved`, if `NULL`, indicates that the process attachment was caused by `LoadLibrary`; otherwise, it was caused by implicit load-time linking. Likewise, `FreeLibrary` gives a `NULL` value for process detachment.

`Reason` will have one of four values: `DLL_PROCESS_ATTACH`, `DLL_THREAD_ATTACH`, `DLL_THREAD_DETACH`, and `DLL_PROCESS_DETACH`. DLL entry point functions are normally written as `switch` statements and return `TRUE` to indicate correct operation.

The system serializes calls to `DllMain` so that only one thread at a time can execute it (Chapter 7 introduces threads). This serialization is essential because `DllMain` must perform initializations that must complete without interruption. As a consequence, however, there should not be any blocking calls, such as I/O or wait functions (see Chapter 8), within the entry point, because they would prevent other threads from entering. Furthermore, you cannot call other DLLs from `DllMain` (there are a few exceptions, such as `InitializeCriticalSection`).

`LoadLibrary` and `LoadLibraryEx`, in particular, should never be called from a DLL entry point, as that would create additional DLL entry point calls.

An advanced function, `DisableThreadLibraryCalls`, will disable thread attach/detach calls for a specified DLL instance. As a result, Windows does not need to load the DLL's initialization or termination code every time a thread is created or terminates. This can be useful if the DLL is only used by some of the threads.

## DLL Version Management

A common problem with DLLs concerns difficulties that occur as a library is upgraded with new symbols and features are added. A major DLL advantage is that multiple applications can share a single implementation. This power, however, leads to compatibility complications, such as the following.

- A new version may change behavior or interfaces, causing problems to existing applications that have not been updated.
- Applications that depend on new DLL functionality sometimes link with older DLL versions.

DLL version compatibility problems, popularly referred to as “DLL hell,” can be irreconcilable if only one version of the DLL is to be maintained in a single directory. However, it is not necessarily simple to provide distinct version-specific directories for different versions. There are several solutions.

- Use the DLL version number as part of the .DLL and .LIB file names, usually as a suffix. For example, `Utility_4_0.DLL` and `Utility_4_0.LIB` are used in the *Examples* projects to correspond with the book edition number. By using either explicit or implicit linking, applications can then determine their version requirements and access files with distinct names. This solution is commonly used with UNIX applications.
- Microsoft introduced the concept of side-by-side DLLs or assemblies and components. This solution requires adding a manifest, written in XML, to the application so as to define the DLL requirements. This topic is beyond the book’s scope, but additional information can be found on the MSDN Web site.
- The .NET Framework provides additional support for side-by-side execution.

The first approach, including the version number as part of the file name, is used in the *Examples* file, as mentioned in the first bullet.

To provide additional support so that applications can determine additional DLL information beyond just the version number, `DllGetVersion` is a user-provided callback function; many Microsoft DLLs support this callback function as a standard method to obtain version information dynamically. The function operates as follows:

```
HRESULT CALLBACK DllGetVersion(
    DLLVERSIONINFO *pdvi
)
```



- Information about the DLL is returned in the `DLLVERSIONINFO` structure, which contains `DWORD` fields for `cbSize` (the structure size), `dwMajorVersion`, `dwMinorVersion`, `dwBuildNumber`, and `dwPlatformID`.
- The last field, `dwPlatformID`, can be set to `DLLVER_PLATFORM_NT` if the DLL cannot run on Windows 9x (this should no longer be an issue!) or to `DLLVER_PLATFORM_WINDOWS` if there are no restrictions.
- The `cbSize` field should be set to `sizeof(DLLVERSIONINFO)`. The normal return value is `NOERROR`.
- `Utility_4_0` implements `DllGetVersion`.

## Summary

Windows memory management includes the following features.

- Logic can be simplified by allowing the Windows heap management and exception handlers to detect and process allocation errors.
- Multiple independent heaps provide several advantages over allocation from a single heap, but there is a cost of extra complexity to assure that blocks are freed, or resized, from the correct heap.
- Memory-mapped files, also available with UNIX but not with the C library, allow files to be processed in memory, as illustrated by several examples. File mapping is independent of heap management, and it can simplify many programming tasks. Appendix C shows the performance advantage of using memory-mapped files.
- DLLs are an essential special case of mapped files, and DLLs can be loaded either explicitly or implicitly. DLLs used by numerous applications should provide version information.

## Looking Ahead

This completes coverage of what can be achieved within a single process. The next step is to learn how to manage concurrent processing, first with processes (Chapter 6) and then with threads (Chapter 7). Subsequent chapters show how to synchronize and communicate between concurrent processing activities.

## Additional Reading

### *Memory Mapping, Virtual Memory, and Page Faults*

Russinovich, Solomon, and Ionescu (*Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition*) describe the important concepts, and most OS texts provide good in-depth discussion.

### *Data Structures and Algorithms*

Search trees and sort algorithms are explained in numerous texts, including Cormen, Leiserson, Rivest, and Stein's *Introduction to Algorithms*.

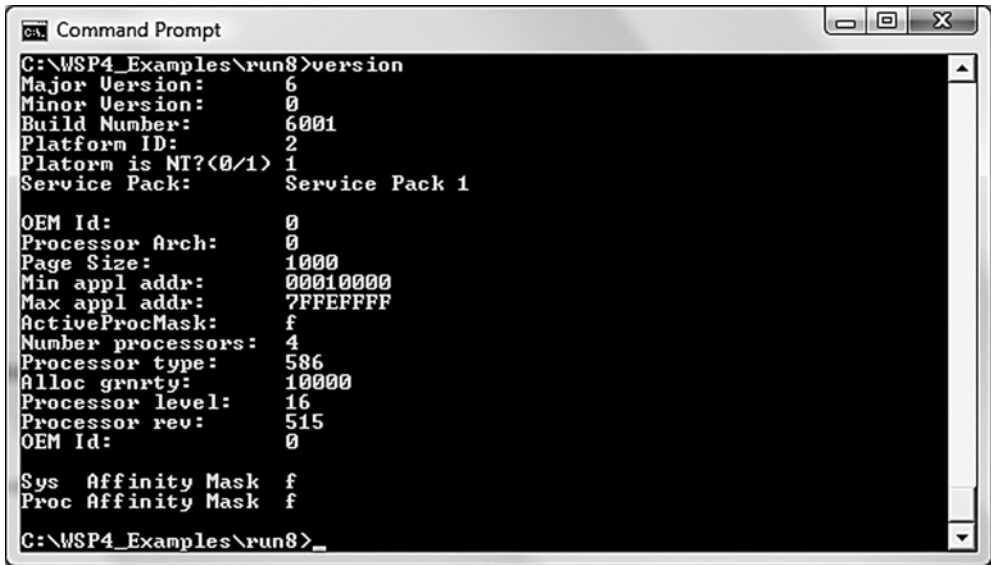
### *Using Explicit Linking*

DLLs and explicit linking are fundamental to the operation of COM, which is widely used in Windows software development. Chapter 1 of Don Box's *Essential COM* shows the importance of `LoadLibrary` and `GetProcAddress`.

## Exercises

- 5-1. Design and carry out experiments to evaluate the performance gains from the `HEAP_NO_SERIALIZE` flag with `HeapCreate` and `HeapAlloc`. How are the gains affected by the heap size and by the block size? Are there differences under different Windows versions? The *Examples* file contains a program, `HeapNoSr.c`, to help you get started on this exercise and the next one.
- 5-2. Modify the test in the preceding exercise to determine whether `malloc` generates exceptions or returns a null pointer when there is no memory. Is this the correct behavior? Also compare `malloc` performance with the results from the preceding exercise.
- 5-3. Windows versions differ significantly in terms of the overhead memory in a heap. Design and carry out an experiment to measure how many fixed-size blocks each system will give in a single heap. Using SEH to detect when all blocks have been allocated makes the program easier. A test program, `clear.c`, in the *Examples* file will show this behavior.
- 5-4. Modify `sortFL` (Program 5-4) to create `sortHP`, which allocates a memory buffer large enough to hold the file, and read the file into that buffer. There is no memory mapping. Compare the performance of the two programs.

- 5-5. Compare random file access performance using conventional file access (Chapter 3's `RecordAccess`) and memory mapping (`RecordAccessMM`).
- 5-6. Program 5-5 exploits the `__based` pointers that are specific to Microsoft C. If you have a compiler that does not support this feature (or simply for the exercise), reimplement Program 5-5 with a macro, arrays, or some other mechanism to generate the based pointer values.
- 5-7. Write a search program that will find a record with a specified key in a file that has been indexed by Program 5-5. The C library `bsearch` function would be convenient here.
- 5-8. Enhance `sortMM` (Programs 5-5 and 5-6) to remove all implicit alignment assumptions in the index file. See the comments after the program listings.
- 5-9. Implement the `tail` program from Chapter 3 with memory mapping.
- 5-10. Modify Program 5-7 so that the decision as to which DLL to use is based on the file size and system configuration. The `.LIB` file is not necessary, so figure out how to suppress `.LIB` file generation. Use `GetVolumeInformation` to determine the file system type. Create additional DLLs for the conversion function, each version using a different file processing technique, and extend the calling program to decide when to use each version.
- 5-11. Put the `ReportError`, `PrintStrings`, `PrintMsg`, and `ConsolePrompt` utility functions into a DLL and rebuild some of the earlier programs. Do the same with `Options` and `GetArgs`, the command line option and argument processing functions. It is important that both the utility DLL and the calling program also use the C library in DLL form. Within Visual Studio, for instance, you can select "Use Run-Time Library (Multithreaded DLL)" in the project settings. Note that DLLs must, in general, be multithreaded because they will be used by threads from several processes. See the `Utilities_4_0` project in the *Examples* file for a solution.
- 5-12. Build project `Version` (in the *Examples* file), which uses `version.c`. Run the program on as many different Windows versions as you can access. What are the major and minor version numbers for those systems, and what other information is available? The following screenshot, Exercise Run 5-12, shows the result on a Vista computer with four processors. The "Max appl addr" value is wrong, as this is a 64-bit system. Can you fix this defect?



```
C:\WSP4_Examples\run8>version
Major Version:      6
Minor Version:      0
Build Number:       6001
Platform ID:        2
Platform is NT?(<0/1>) 1
Service Pack:       Service Pack 1

OEM Id:             0
Processor Arch:      0
Page Size:          1000
Min appl addr:       00010000
Max appl addr:       7FFEFFFF
ActiveProcMask:      f
Number processors:   4
Processor type:      586
Alloc grnrtty:       10000
Processor level:     16
Processor rev:       515
OEM Id:             0

Sys Affinity Mask    f
Proc Affinity Mask    f

C:\WSP4_Examples\run8>_
```

**Exercise Run 5-12** version: System Version and Other Information