DLLs for Beginners

Debabrata Sarma

Microsoft Developer Support

November 1996

Contents

Introduction

The objective of this article is to introduce the concepts of the Dynamic Link Library (DLL) and the mechanism of writing a DLL for Microsoft® Windows® applications. This discussion is limited to 32-bit applications using Microsoft Visual C++® version 4.0 or later and versions 2.*x*. Appendix A contains information about 16-bit DLL implementation and porting to 32-bit DLLs.

Most references used in this article are available in the Microsoft Developer Network Library (MSDN Library). The articles cited in the text that are followed by a "Q" number can be found either in the MSDN Library or in the Microsoft Personal Support Center at http://support.microsoft.com/support/default.asp.

Basic Concepts

A Windows program is an executable file that generally creates one or more windows and uses a message loop to receive user input. Dynamic link libraries are generally not directly executable, and they do not receive messages. They are separate files containing functions that can be called by programs and other DLLs to perform certain computations or functions.

Static Linking

In high-level programming languages such as C, Pascal, and FORTRAN, an application's source code is compiled and linked to various libraries to create an executable file. These libraries contain object files of precompiled functions that are called to accomplish common tasks, such as computing the square root of a number or allocating memory. When these library functions are linked to an application, they become a permanent part of the application's executable file. All calls to the library functions are resolved at link time—thus the name *static linking*.

Dynamic Linking

Dynamic linking provides a mechanism to link applications to libraries at run time. The libraries reside in their own executable files and are not copied into applications' executable files as with static linking. These libraries are called *dynamic-link libraries* (DLLs) to emphasize that they are linked to an application when it is loaded and executed, rather than when it is linked. When an application uses a DLL, the operating system loads the DLL into memory, resolves references to functions in the DLL so that they can be called by the application, and unloads the DLL when it is no longer needed. This dynamic linking mechanism can be performed explicitly by applications or implicitly by the operating system.

Differences Between Static-Link Libraries and Windows DLLs

Windows DLLs differ considerably from static-link libraries. Basic differences between the two are as follows:

- Static-link libraries reside in .LIB files, which are basically collections of object files, whereas dynamic-link libraries reside in separate executable files that are loaded by Windows into memory when they are needed.
- Each application that uses a static-link library has its own copy of the library. However, Windows supports multiple applications simultaneously using one copy of the same DLL.
- Static-link libraries contain only code and data because they are stored as a collection of object files. Windows DLLs, on the other hand, can contain code, data, and resources such as bitmaps, icons, and cursors, because they are stored as executable program files.
- Static-link libraries must use the data space of the application, whereas DLLs may (and often do) have their own data address space mapped into the address space of the process.

Differences Between Windows-Based Applications and DLLs

This section first defines some key terms in Windows programming and then illustrates the concepts behind those terms, and finally explains the specific differences between applications and DLLs.

Definitions

Let's review some basic terms that are thrown around quite a bit in programming for Windows.

- An *executable* is a file with an .EXE or .DLL extension containing executable code and/or resources for an application or DLL.
- An *application* is a Windows-based program residing in an .EXE file.
- A *DLL* is a Windows dynamic link library residing in a .DLL file. System DLLs may have .EXE extensions, for example, USER.EXE, GDI.EXE, KRNL286.EXE, and KRNL386.EXE. Various device drivers have a .DRV extension, for example, MOUSE.DRV and KEYBOARD.DRV. Only dynamic link libraries with a .DLL extension will be loaded automatically by the Windows operating system. If the file has another extension, the program must explicitly load the module using the **LoadLibrary** function.

Before we go further, it is necessary that we have some understanding of how the DLL and the application is mapped in memory.

One of the more significant changes to DLLs for Win32® is the location in memory where a DLL's code and data reside. In Win32, each application runs within the context of its own 32-bit linear address space. In this way, every application has its own private address space that can only be addressed from code within this process. (In Win32, each application's private address space is referred to as the *process* for that application.) All of an application's code, data, resources, and dynamic memory also reside within an application's process. Further, it is not possible for an application to address data or code residing outside of its own process. Because of this, when a DLL is loaded it must somehow reside in the process of the application that loaded the DLL; if more than one application loads the same DLL, it must reside in each application's process.

So, to satisfy the above requirement—that is, to be reentrant (accessible from more than one thread at a time) and to have only one copy of the DLL physically loaded into memory—Win32 uses memory mapping. Through memory mapping, Win32 is able to load the DLL once into the global heap and then map the address range of the DLL into the address space of each application that loads it. Figure 1 depicts how a DLL is mapped to the address space of two different applications simultaneously (Process 1 and Process 2).

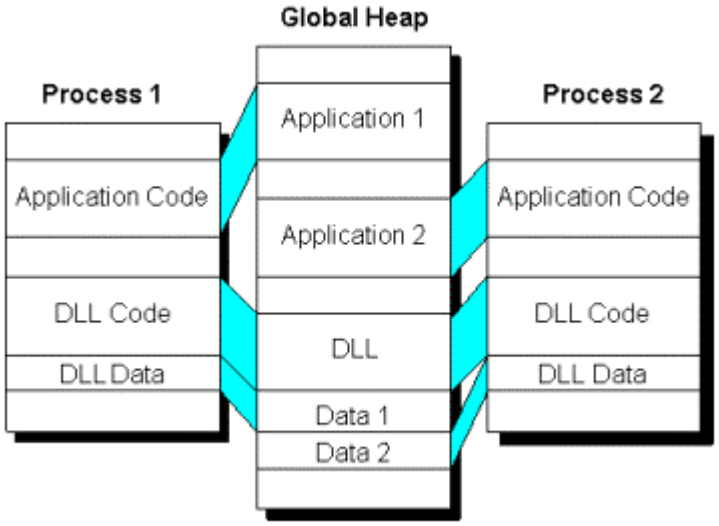


Figure 1. A DLL mapped to the address space of two different applications simultaneously

Notice that the DLL is physically loaded into the global heap only once, yet both applications share the code and resources of the DLL.

In Microsoft Windows version 3.1 and earlier, DLLs have a common set of data that is shared among all applications that load the DLL. This behavior is one of the more difficult obstacles to writing source code in DLLs. Win32 breaks this barrier by permitting DLLs to have separate sets of data, one for each of the applications that load a DLL. These sets of data are also allocated from the global heap, but they are not shared among applications. Figure 1 depicts each DLL data set and how it is mapped to the application to which it belongs. This new feature makes it much easier to write code for DLLs because the DLL does not have to guard against two applications accessing the same global variable(s).

While it is easier to write code for DLLs shared among different applications, there is still a potential conflict in how global variables are used in a DLL. The scope of a DLL's data is the entire process of the application that loaded the DLL. So each thread in an application with multiple threads can access all the global variables in the DLL's data set. Since each thread executes independently of the others, a potential conflict exists if more than one thread is using the same global variable(s). Exercise caution when using global variables in a DLL that can be accessed by multiple threads in an application, and employ the use of semaphores, mutexes (mutual exclusions), wait events, and critical sections where necessary.

Since a DLL is mapped into the process of the application that loaded it, some confusion may exist as to who owns the current process when executing code in a DLL. Calling the **GetModuleHandle** function with a null module name parameter returns the module handle for the application, not the DLL, when called from the code of a DLL. It is a good idea to keep a copy of the DLL's module handle in a static variable for future reference when accessing the DLL's resources (the module handle is passed to the **DllEntryPoint** function). The only other way to obtain the handle again is to use **GetModuleHandle** with a valid string identifying the path and filename of the DLL.

How applications and DLLs differ

Even though DLLs and applications are both executable program modules, they differ in several ways. To the end user, the most obvious difference is that DLLs are not programs that are directly executed from the Program Manager or other shell program. From the system's point of view, there are two fundamental differences between applications and DLLs:

- An application can have multiple instances of itself running on the system simultaneously, whereas a DLL can have only one instance. Each instance of an application has its own automatic data space, but all instances of the application share a single copy of the executable code and resources. On the other hand, no matter how many times a DLL is loaded, it has exactly one instance. For a 32-bit operating system, each loading of a DLL will have its own instance; therefore, sharing of data segment among multiple processes is not straightforward and not recommended for 32-bit DLLs.

- An application can "own" things, but a DLL cannot. Only processes are capable of "ownership," and only application instances are processes. In a 32-bit operating system, DLLs attach themselves to processes, memory can be owned only by an individual process. DLLs are not entities by themselves anymore.

DLLs are program modules separate from applications. On disk, they reside in their own special executable files, which may contain code, data, and resources (read-only data) such as bitmaps and cursors. When a process loads the DLL, the system maps the code and data for the DLL into the address space of the process. For a 32-bit operating system, DLLs do not become part of the operating system, instead they become part of the process that loads the DLL. Any memory allocation calls made by functions in the DLL cause memory to be allocated from the process's address space; no other process has access to this allocated memory. The global and static variables allocated by a DLL are also not shared among multiple mappings of the DLL.

When a process loads a DLL for the first time, the usage count for that DLL becomes 1. If that process calls **LoadLibrary** to load the same DLL a second time, the usage count for the library with respect to the process becomes 2. If another process calls **LoadLibrary** to load a DLL that is being used by another process, the system maps the code and data for the DLL into the calling process's address space and increments the DLLs usage count (with respect to this process) to 1. The **FreeLibrary** function decrements the usage count of the DLL; if the usage count reaches 0, the DLL is unmapped from the process's address space.

Advantages of Using a DLL

DLLs are compiled and linked independently of the applications that use them; they can be updated without requiring applications to be recompiled or relinked.

If several applications work together as a system and all share common DLLs, the entire system can be improved by replacing the common DLLs with enhanced versions. A bug fix in one of the DLLs fixes the bug in all applications that use it. Likewise, speed improvements or new functionality benefit all applications that use the DLLs.

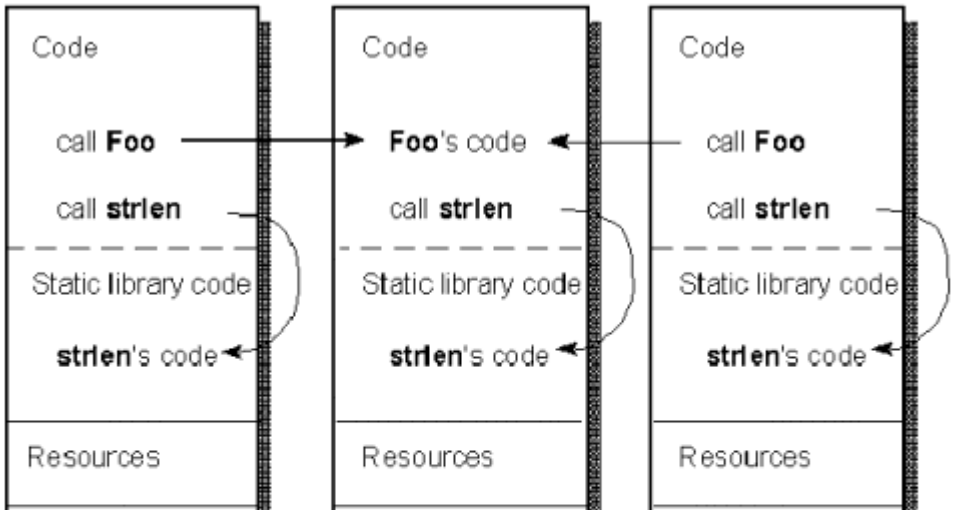
DLLs can reduce memory and disk space requirements by sharing a single copy of common code and resources among multiple applications; the memory and disk space saved increases as more applications use the DLL. If multiple applications use a static-link library, there are several identical copies of the library on the disk. If the applications were all run simultaneously, then there would be several identical copies in memory. As the number of applications increases, the number of identical copies increases as well. These identical copies are redundant and waste space. If a DLL is used instead of static-link libraries, then only one copy of the code and resources is needed, no matter how many different applications use it.

Any Disadvantages?

The primary disadvantage of using DLLs over static-link libraries is that DLLs can be more difficult to develop.

Sometimes, using DLLs can actually increase memory and disk space usage, especially if only a single application uses the DLL. This happens when both the application and the DLL use the same static-link library functions, as in the case of C run-time library functions. If the DLL and the application each have a copy of a static-link library function linked into it, there will be two copies of the library function in memory and on disk, which wastes space. To avoid this problem, the distribution of functions and function calls must either be properly modularized to avoid duplication or use the DLL version of the C run-time library.

In Figure 2, we have two applications and a DLL using a static-link library function, **strlen**; additionally, the two applications use a dynamic-link library function, **Foo**, which also uses **strlen**. Note that all three executable modules have their own copy of **strlen**'s code (total of three static libraries), whereas **Foo** is reused by the two applications.



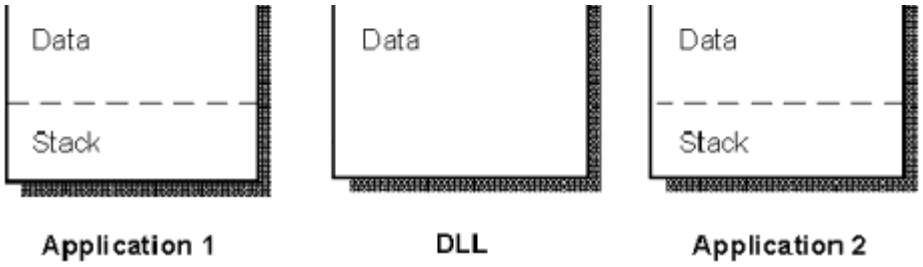


Figure 2. Two applications and a DLL using a static-link library function and a dynamic-link library function

Implementation of DLLs

With basic concepts behind us, it is time to get to the business of writing a DLL. A DLL can be implicitly loaded by the operating system or explicitly loaded by the application.

- *Implicit dynamic linking* is performed at load time by the operating system. When the operating system loads an application or DLL, it first loads any dependent DLLs (that is, DLLs that are used by the application or DLL).
- *Explicit dynamic linking* is performed at run time by the application or DLL itself by making calls to the operating system to load the DLL using the **LoadLibrary** function.

Implementing a 32-bit DLL

Let us first build a static library and then convert it to a DLL. That way if you have not implemented a static library before, then you are going to get that experience now. Let us also get into the habit of writing a header file for function prototypes instead of declaring the prototypes in the implementation files (.C or .CPP). Third-party vendors supply header files with their static or dynamic link libraries for inclusion in the application that uses their libraries.

Building a Static Library

The static library example we are going to build will have a global variable, local variables, functions to be called by application, a local function, and a callback function in the application. The application will be a Console Application. The library uses one header file for its local function prototypes and another for the shared functions and variables (this header file may also be used as is by the application). The header files and the source file are shown below. Create these files in a directory of your choice.

In the Visual Workbench, choose New from the File menu. Choose Project and then choose a project name (normally in the same directory as your files are created) of project type "Static Library." Choose Create.

```
-------------------------------------------------------------------
// Library header  file  lib1.h
int libglobal;
int callappfunc(int * ptrlib);
void libfuncA(char * buffer);
int  libfuncB(int const * mainint);
-------------------------------------------------------------------
// This is a header file of the library local functions-- lib2.h
 int liblocalA(int x, int y);
-------------------------------------------------------------------
```

```
// This is  the library source file lib1.c
#include <stdio.h>
#include "lib1.h"
#include "lib2.h"
libglobal = 20;
void libfuncA(char * buffer)
 {
    int i =0;
    printf("Printing in Library \n%s\n\n", buffer);
    printf("Value of i: %d\n", i);
 }

int libfuncB(int const * mainint)
 {
    int returnvalue;
    int  localint;
    int libvalue = 3;
    localint = *mainint;
    returnvalue = liblocalA(localint, libglobal);

    libvalue = callappfunc(&libvalue);
    printf("Print return value from app callback :%d\n", libvalue);
    return returnvalue;

 }
int liblocalA(int x, int y)
 {
       return (x + y);
 }
-----------------------------------------------------------------------
```

Now build the library for the debug version. You will find that a subdirectory Debug (Windebug subdirectory if Visual C++ version 2.*x* is used) is created for the project directory. In that directory, you will find the static library just built with the name Libname.lib, where libname is the name you chose for the project earlier. This library is known as "Object Library," where the library contains the object code of the functions included in the library. If you want to know how the library functions and variables are listed (or decorated) inside the library, you can use the Dumpbin utility from the DOS command prompt as follows:

```
DUMPBIN  /ALL  libname.lib
```

The output of the above command gives you the decorated names for functions, which is very useful in solving linker problems like LNK2001 error (Unresolved external). You also get the information of the default libraries used, which is useful in solving LNK2005 error (Symbol already defined in another library). Run the above command on the library just created and find out what default libraries are used and how the functions and variables are listed (or decorated). Other useful DUMPBIN switches are /IMPORTS and /EXPORTS. You may refer to the following knowledge base articles for more information:

- "The /Mx Compiler Options and the LIBC, LIBCMT, MSVCRT Libs" (Q128641)
- "Troubleshooting LNK2001 or L2029 Unresolved External Errors" (Q138400)
- "How to Link with the Correct C Run-Time (CRT) Library" (Q140584)
- "PRB: Error Msg: LNK2001 on _beginthreadex and _endthreadex" (Q126646)
- "How to Determine Default Libraries for an .OBJ Object File" (Q143072)

We are now ready to build our application, which will use the library we just built. The following

header and source files are used. Note that we are using the same header file as used by the library. Create the source file in a directory of your choice (try a directory other than the one you created for your library) and copy the header file Lib1.h to this directory. Create a new project of type "Console Application." Add to this project the source file you just created. To include the library in your project, you may either add the library in your project from Insert - Files into Project menu or add in the Build—Settings—Link—Input—Object/Library Modules edit box. Provide the complete path of the library. Build your project for debug mode.

```
-----------------------------------------------------------------------
// This is the application source file  -- appsource.c
#include "lib1.h"
#include <stdio.h>
int mainglobal;
int callappfunc(int * ptrlib)
 {
    int m =  *ptrlib;
    return m = m * m  - libglobal;

 }
 void main(void)
  {
      int libreturn;
    char  buf[] = "Let us print this line in library";
    printf ("We are starting in the application\n");
    libfuncA(buf);
    printf("Let us increment the library global and print in app :%d\n",
               ++libglobal);

    mainglobal = 10;
    libreturn = libfuncB( &mainglobal);
    printf("Print the library returned value :%d\n", libreturn);
    printf("Demonstration of library use is complete\n");
  }
-----------------------------------------------------------------------
```

As our project is built, we may try to debug it and see whether we are able to step through all the functions in the library and our application. Instead of setting a breakpoint in the code and pressing the F5 key (Debug Go), as this is a small application, we can as well press the F11 (Step Into) key (F8 key if Visual C++ version 2.*x* is used). This will bring us to the curly brace of the main function. From that point, we can press F11 and step through all the code, including library functions. We can verify the output by using ALT+TAB to go back and forth between the console command line window and the Visual C Workbench source debug window. Once you reach the last right curly brace in the main function, you can press F5 to terminate debugging or Stop Debug (or ALT+F5) from debug menu. A breakpoint can be set by placing the cursor on a line of code in the source file and clicking on the hand symbol button on the toolbar—a red dot appears on the line to indicate a breakpoint. Pressing F5 will stop execution of program at this breakpoint.

Note that an application can be built using the release version of a library. You can verify this by building the library in the release mode and then adding this library in your application, which is then built for the debug mode. If you now debug your code, you will find that any time a library function is called, it does not step through the code; instead, the library function is properly executed and you get the expected behavior of your code. Although we could get by without using the **extern** keyword for the *libglobal* variable declaration, it is a good programming practice to use **extern** for variables declared outside the module.

We implemented a static library written in C and used by a C application. The same static library should be usable from a C++ application also. But, if you now rename your Appsource.c file to appsource.cpp and add this file to your project after removing the appsource.c file and build it, you get the following linker error. This is because C++ decorates (or mangles) the

name of variables and functions.

```
appsource.obj : error LNK2001: unresolved external symbol "?libfuncA@@YAXPAD@Z (void  __cdecl
appsource.obj : error LNK2001: unresolved external symbol "?libfuncB@@YAHPBH@Z (int  __cdecl libfu
```

We can instruct the compiler to keep the name decoration used by C by declaring the functions and variables in the following way.

```
extern "C"  functionname /variablename ;
```

We can also group several declarations together. The modified header file is shown below. You can build and test the application.

```
-----------------------------------------------------------------------
// This is the header file for the application for C++ -- applib.h
#ifdef __cplusplus
extern "C" {
#endif
extern int libglobal;
int callappfunc(int * ptrlib);
void libfuncA(char * buffer);
int  libfuncB(int const * mainint);
#ifdef __cplusplus
}
#endif
-----------------------------------------------------------------------
```

Building a DLL

Implicit dynamic linking

Using the __declspec attribute

Let us now convert the static library we built to a Dynamic Link Library (DLL). We can build a DLL using the **dllexport** attribute or use a separate module-definition-file (.DEF file). We will discuss both methods. First, we will discuss DLL implementation using the **dllexport** attribute. Using the **dllimport** attribute improves efficiency and enables you to import data and objects as well as code. Because they are attributes and not keywords, **dllexport** and **dllimport** must be used in conjunction with the **__declspec** keyword. The syntax for using **__declspec** is:

__declspec(*attribute*) variable-declaration

For example, the following definition exports an integer, using **dllexport** as the attribute:

```
__declspec(dllexport) int Sum = 0;
```

Using the above syntax, let us now change the prototype declaration in our header file that we used in the static library. The header file for the DLL now looks like the one shown below. For the time being, we will comment out all references to the callback function in the DLL; that is, we are not going to call from the DLL a function in the application. The corresponding call to **callappfunc()** in the lib1.c is therefore removed.

```
-----------------------------------------------------------------------
// Library header  lib1.h
__declspec(dllexport) int libglobal;
__declspec(dllexport) void libfuncA(char * buffer);
__declspec(dllexport) int  libfuncB(int const * mainint);
```

```
-----------------------------------------------------------------
```

Now create a new project, give it the name Libname, and choose its type as "Dynamic-LinkLibrary". Insert the modified file Lib1.c into the project, and build the project for debug mode. You now have in the Debug subdirectory two files named Libname.lib and LIBNAME.DLL, where Libname is the name you chose for the project. The file Libname.lib is known as the "Import Library" for the DLL (LIBNAME.DLL) that we just built. This file does not contain any code, it contains all the references to the functions and other declarations needed by the linker to link with the application. The object code of the functions are in the file LIBNAME.DLL.

We are now ready to build our application. We will use the same source and header files we used while building the static library. This time, the header file is not the same as the one used in the DLL because you were using the header file used in the static library. You may give a different name to this header file to indicate the difference; for this application, call it Applib.h.

```
-----------------------------------------------------------------
// This is the header file for the application when using the  DLL   -- applib.h
int libglobal;
void libfuncA(char * buffer);
int  libfuncB(int const * mainint);
-----------------------------------------------------------------
```

Again, create a project named Console Project and add the source file Appsource2.c in the project. As before, we can add to this project the import library we just built with the full path in the Insert -Files into Project menu or add in the Build—Settings—Link—Input—Object/Library Modules edit box. Build the project.

```
-----------------------------------------------------------------   //
This is the application source file   -- appsource2.c
#include "applib.h"
#include <stdio.h>
int mainglobal;
void main(void)
  {
      int libreturn;
    char  buf[] = "Let us print this line in library";
    printf ("We are starting in the application\n");
    libfuncA(buf);
    printf("Let us increment the library global and print in app :%d\n",
               ++libglobal);

    mainglobal = 10;
    libreturn = libfuncB( &mainglobal);
    printf("Print the library returned value :%d\n", libreturn);
    printf("Demonstration of library use is complete\n");
  }
-----------------------------------------------------------------
```

Debugging

While our project is being built, we can start debugging by pressing the F11 key. But this time, we get an error message in a message box indicating that the dynamic link library LIBNAME.DLL cannot be found in the specified path and lists the directories it searched to find the DLL. This means that the operating system is trying to load the DLL from some specific directories and that the places it looks for are the current directory, the directory of the executable, the windows system directory, the windows directory, and all the directories specified in the path environment variable. That means we need to move the DLL we built to one of these directories. In the 16-bit Visual C environment, the message is not as clear as above; see the article "PRB: Error 'Could not load debuggee. File not found (2)' " (Q125616).

Let us move the LIBNAME.DLL file to the Debug subdirectory of the application project. Now press F11 again to start the debugger and step into the application. You can see that you are able to step through the functions in the DLL, and everything works fine except that we are getting a value of 1 when we print the *libglobal* variable in the application. This behavior is different compared to the static library example. This is because the static library and the application had one data segment for the library and the application, whereas now the application has its own data segment and the DLL has its own data segment. So, *libglobal* is treated by the application as its own global variable and, likewise, the DLL is treating it as its own global variable. So our intention of exporting *libglobal* from the DLL to the application is not working. To make it work the intended way, we need to modify our declaration of *libglobal* in the application header file. So the modified application header file looks like the following:

```
-------------------------------------------------------------------
// This is the header file for the application when using the  DLL   -- applib.h
__declspec( dllimport) int libglobal;
void libfuncA(char * buffer);
int  libfuncB(int const * mainint);
-------------------------------------------------------------------
```

This means that any time a data type or object needs to be exported from a DLL, the application needs to use the **__declspec(dllimport)** attribute as shown above. Build the application again and try debugging as before. This time, everything will work fine.

The above debugging session was started from the application's project workspace. It is also possible to debug a DLL from the project workspace of the DLL. To do this, set a breakpoint in your DLL function. Type the full path of your application that uses this DLL in the menu Build—Settings—Debug—General—Executable For Debug Session edit box. Start the debugger by pressing the F5 key; the debugger will stop at the breakpoint set in the DLL. You can now step through your DLL code. You can debug the DLL even if the application is not built with debug information. Thus, if applications like Word or Microsoft Excel calls into a user-supplied DLL (built with debug information), you can debug the DLL with this technique. If you forgot to type the name of your application and pressed F5, a message box will appear asking you to enter the full path of the application.

Use __declspec(dllimport)

Although we have not used **__declspec(dllimport)** in the functions exported by the DLL, doing so will improve the efficiency of the code. The article "Under the Hood" by Matt Pietrek in the November 1995 issue of the *Microsoft Systems Journal* has a detailed explanation. If we do so, the header file will be as follows. Try it out.

```
-------------------------------------------------------------------
// This is the header file for the application when building a DLL   -- applib.h
__declspec( dllimport) int libglobal;
__declspec( dllimport) void libfuncA(char * buffer);
__declspec( dllimport) int  libfuncB(int const * mainint);
-------------------------------------------------------------------
```

Use the Same Header File

How about using the same header file for the DLL as well as for the application? Try the following header file by typing the line **#define MAKE_A_DLL** at the very top of your DLL source file. Add this header file to both the DLL and the application, this means replacing the Applib.h header file with this Lib1.h header file in the application's source file.

```
-------------------------------------------------------------------
```

```
// Common Library header for DLL and application  -- lib1.h
#ifdef MAKE_A_DLL
#define LINKDLL __declspec( dllexport)
#else
#define LINKDLL __declspec( dllimport)
#endif
LINKDLL int libglobal;
LINKDLL void libfuncA(char * buffer);
LINKDLL int  libfuncB(int const * mainint);
-----------------------------------------------------------------------
```

Using the DEF File

Earlier we mentioned that a DLL can be implemented using the module definition file (.DEF extension) without using **__declspec(dllexport)**. There may be some situations where you may have to build a DLL by using the module definition file. We will discuss those situations later. But you must remember that the client application must use **__declspec(dllimport)** when importing data items from a DLL.

Let us first build the DLL by using the .DEF file. We need to take out all the **declspec** specifiers from the header file and add a module definition file. The two files now appear as follows:

```
-----------------------------------------------------------------------
// Header for DLL using .DEF -- lib1.h
int libglobal;
void libfuncA(char * buffer);
int  libfuncB(int const * mainint);
-----------------------------------------------------------------------
;lib1.def
; The LIBRARY entry must be same as the name of your DLL, the name of
; our DLL is lib1.dll
LIBRARY lib1
EXPORTS

       libglobal
       libfuncA
       libfuncB
-----------------------------------------------------------------------
```

The **EXPORTS** section lists all the exported definitions.

The **LIBRARY** statement identifies the module-definition file as belonging to a DLL. This must be the first statement in the file. The name specified in the **LIBRARY** statement identifies the library as the DLL's *import library*.

Add the Lib1.def file to the DLL project. Build the DLL.

To build the application, we have to change the library header file included in the application. The following header file is used. Everything else remains same as before. Note the use of the **declspec** specifier for the data member (which is a must).

```
-----------------------------------------------------------------------
//  Library (DLL) header for application -- applib.h
__declspec( dllimport) int libglobal;
void libfuncA(char * buffer);
int  libfuncB(int const * mainint);
-----------------------------------------------------------------------
```

Build the application and then run and debug it. You get the expected result.

The 32-bit version of Visual C++ compiler does not have the Implib utility; because of that, it is no longer possible to build the import library from a DLL if it is not provided. The article "How to Create 32-bit Import Libraries Without .OBJs or Source" (Q131313) describes a technique by which the import library can be built from a DLL.

A DEF file can also be used to call an exported function with a different name. Thus you can implement a DLL where a function can be called with its original name in the DLL or a different name. Before giving an example, I would like to introduce the meaning of the **__cdecl** and **__stdcall** calling conventions used in function calls.

__cdecl is the default calling convention for C and C++ programs. Because the stack is cleaned up by the caller, it can do **vararg** functions. The **__cdecl** calling convention creates larger executables than **__stdcall**, because it requires each function call to include stack cleanup code. The C name decoration scheme is to decorate the function name with an underscore. For the following function declaration, the decorated name is _Myfunc. The use of **__cdecl** is optional.

```
int  __cdecl Myfunc (int a, double  b)
```

The **__stdcall** calling convention is used to call Win32 API functions. The callee cleans the stack, so the compiler automatically converts a variable argument (**vararg**) function to **__cdecl**. The C name decoration scheme is to decorate the function name with an underscore followed by an @*#nn*, where *#nn* is the total number of bytes (in multiples of 4) in the argument list in decimal, or the number of bytes needed by the arguments in the stack. For the following function declaration, the decorated name is Myfunc@12.

```
int  __stdcall Myfunc (int a, double  b)
```

For both **__cdecl** and **__stdcall**, the arguments are passed in the stack from right to left. In WINDOWS.H header file, WINAPI, PASCAL, and CALLBACK are defined as **__stdcall**. Use WINAPI where you previously used PASCAL or **__far __pascal**.

The PASCAL name decoration of a function is to convert the function name to all uppercase letters. Languages like Visual Basic® follows the PASCAL convention. Thus, the **MyFunc** function shown above will have the PASCAL decorated name of MYFUNC. For a Visual Basic program to call a C DLL, we can use the DEF file to call the same function with different (decorated) names. The following example shows how to achieve this by using the DEF file. The calling application in this example is a C program.

Create a project called Pascaldll as a project type of Dynamic Link Library in the Developer Studio. Add the .CPP and .DEF file shown below to the project. When built, this project will create a DLL called PASCALDLL.DLL and an import library called Pascaldll.lib.

```
-------------------------------------------------------------------
// DLL source file pascal.cpp
#include"pascal.h"
double __stdcall MyFunc(int a, double b)
{
    return a*b;
}
int CdeclFunc(int  a)
{
    return  2*a;
}
-------------------------------------------------------------------
//Header file  pascal.h
```

```
extern "C" double __stdcall MyFunc(int a, double b);
extern "C" int  CdeclFunc(int a);   //By default this is a __cdecl convetion
-------------------------------------------------------------------
; DEF file for the  DLL
EXPORTS
   MYFUNC =_MyFunc@12 ; Alias for function call as MYFUNC for pascal calls
   _MyFunc@12                  ;  To call as MyFunc in C in __stdcall
  CdeclFunc                    ;   Called from C program in __cdecl
  CMyFunc = CdeclFunc          ;  Calling CdeclFunc with different name
-------------------------------------------------------------------
```

After building this DLL, if you run DUMPBIN /EXPORTS utility on this DLL, you will see the following functions exported.

```
    CMyFunc
             CdeclFunc
             MYFUNC
             _MyFunc@12
```

Now, build a console application called Pascalmain and add the following .CPP file to the project and the Pascaldll.lib import library as obtained for the DLL built above.

```
-------------------------------------------------------------------
// Application .CPP file pascalmain.cpp
#include "pascal.h"
#include "pascal2.h"
#include <iostream.h>
voidmain(void)
{
    int x = 3;
    double y = 2.3;
        int a;
    double b;
    b = MyFunc(x,y);
    cout<< "b=" <<b <<endl;
        a = CdeclFunc(x);
        cout << a <<endl;
        a = CMyFunc(a*x);       // Calling with a different name
        cout << a <<endl;
}
-------------------------------------------------------------------
// Header file pascal2.h, needed to resolve CMyFunc
extern "C"  int  CMyFunc(int a);
-------------------------------------------------------------------
```

Build this application and run it. It will run as expected.

A very good discussion of calling a C DLL from Visual Basic can be found in the file VB4DLL.TXT in the VB directory.

Other useful features of the DEF file is use of the ordinal number and the NONAME flag. Make the following changes in the DEF file and build the DLL again. The ordinal number can be any decimal number usually set to indicate the number of functions in the DLL, which in turn provides a shorthand method for getting the address of the function in the DLL when **GetProcAddress** function is used (discussed in the following section on explicit linking).

```
-------------------------------------------------------------------
; DEF file for the  DLL
EXPORTS
   MYFUNC =_MyFunc@12
  _MyFunc@12                    @1
  CdeclFunc                     @2  NONAME
```

```
  CMyFunc = CdeclFunc
------------------------------------------------------------------------
```

The NONAME flag indicates that you do not want the function name to be exported, only the ordinal value. The linker will not add the function name in the function table inside the DLL. This means the application will not be able to call **GetProcAddress** passing in the name of a function.

Specifying the NONAME flag does not cause the names of the functions in the DLL removed from the import library. Otherwise, we will not be able to link our application to a DLL that implicitly uses function names.

After building the DLL, if you run the DUMPBIN/EXPORTS utility on this DLL, you will see the following functions exported. Notice the absence of the function name **CdeclFunc**.

```
ordinal    hint      name
   4       0     CMyFunc
   3           1    MYFUNC
   1       2    _MyFunc@12
```

You need to build (or link) the application again to account for the above changes in the import library. The application will run as usual.

The advantage of using the NONAME flag in a DLL is that instead of putting function names as strings in a DLL, only the ordinal number is used. This saves a lot of space if your DLL contains hundreds of functions like the USER32.DLL, which exports about 550 functions.

Exporting Classes

Exporting C++ is as simple as exporting functions and data members, as we showed above. You can use the **declspec** attribute or the DEF file. The article "__declspec(dllexport) Replaces __export in 32-bit Visual C++ " (Q107501) shows how to use the **declspec** attribute to export classes. To use the DEF file approach, copy the C++ decorated names of the items you want to export from the .map file, which you can generate by choosing the corresponding link option or from the link settings for the General category in the Developer Studio. Use these decorated names in your DEF file.

If you are exporting a derived class, you need to export its base class also. The same applies to an embedded class. Another article worth reading is "Exporting with Class" by Dale Rogerson (MSDN Library Archive Edition).

There is a new/delete mismatching problem while creating and deleting classes across the EXE and the DLL, which may cause run-time errors. To avoid this situation, follow the suggestions provided in the article "BUG: Wrong Operator Delete Called for Exported Class" (Q122675).

Exporting Template Class and Functions

Visual C++ does not support export of template class or functions. One reason for this arises from the notion of a template itself. A class template is an abstract representation of similar, but distinct, types. There is no code or data that exists to be exported until an instance of a class template is generated. One workaround (not recommended) is to declare the template class or functions with the **declspec** attribute in the DLL and instantiate all possible types in the DLL source. As you can see, this defeats the very purpose of using templates.

For the same reason, you cannot export the Standard Template Library (STL) classes or

functions.

Explicit Dynamic Linking

Explicit dynamic linking is used when you want to load the DLL only when you are ready to call the functions in the DLL in your application. When you are done using the functions, you can unload the DLL from memory, thus saving memory space when the application is running, and load another DLL for some other function calls. The two functions used to achieve this are **LoadLibrary** and **FreeLibrary**. To get the function addresses **GetProcAddress** is used. **GetProcAddress** can use the actual function name or the ordinal number of the function as associated in the DEF file, which we discussed before. The use of the ordinal number lets **GetProcAddress** locate the function in the DLL function table quickly instead of searching the table with a function name string. If the NONAME flag is used in the DEF file, **GetProcAddress** must use the ordinal number. To get the function address through the ordinal number, the MAKEINTRESOURCE macro is used. The following example shows how to write the code on the application side to achieve this. Notice the absence of the header files Pascal.h and Pascal2.h. Note that as the DLL is loaded explicitly, the application does not link with the DLL import library. But, the DLL has to be in the directories specified for **LoadLibray** to succeed. In our example below, we copied the pascaldll.dll into the application's EXE directory. Use the following source file for the application. Build the application without the import library pascaldll.lib.

```
-------------------------------------------------------------------------
// Application .CPP file  explicitmain.cpp
#include <windows.h>
#include <iostream.h>
void main(void)
{
        typedef int ( * lpFunc1)(int);
    typedef double ( * lpFunc2)(int, double);
    HINSTANCE  hLibrary;
    lpFunc1 Func1, Func2;
    lpFunc2 Func3;

    int x = 3;
    double y = 2.3;
        int a,c;
    double b;

    hLibrary = LoadLibrary("pascaldll.dll"); // Load the DLL now

    if (hLibrary != NULL)
    {
        Func1 = (lpFunc1) GetProcAddress(hLibrary, "CMyFunc");
        if (Func1 != NULL)
            a = ((Func1)(x ));
        else cout << "Error in Func1 call" << endl;
        Func2 = (lpFunc1) GetProcAddress(hLibrary,
                    MAKEINTRESOURCE(2));
        if (Func2 != NULL)
            c = ((Func2)(a*x ));
        else cout << "Error in Func2 call" << endl;

        Func3 = (lpFunc2) GetProcAddress(hLibrary,
                    MAKEINTRESOURCE(1));
        if (Func3 != NULL)
            b = ((Func3)( x, y ));
        else cout << "Error in Func3 call" << endl;
    }
    else cout << "Error in Load Library" << endl;
    cout << "b=" << b << endl;
        cout << "a=" << a << endl;
```

```
            cout << "c=" << c << endl;
    FreeLibrary(hLibrary);         // Unload DLL from memory
}
-----------------------------------------------------------------
```

Run the application. It will show expected output.

While debugging, you will not be able to set a breakpoint at any of the DLL functions unless **LoadLibrary** loads the DLL. To circumvent this situation, open the **Settings** dialog box from the **Build** menu, click the **Debug** tab, choose **Additional DLLs** from the **Category** drop down list box and enter the dynamically loaded DLL's name in the **Local Name** field. Now, when you start the debugger by pressing F11 or F5, you will see, in the Debug output window, the message "Loaded symbols for mydllname.dll". You can open your DLL source file and set a breakpoint even though the **LoadLibrary** call has not been made yet.

If you are running your application on the Windows NT® operating system, you can find out which DLLs are loaded during the application's run time by using the PVIEW utility. In PVIEW, highlight your application, and then click on the **Memory Detail** button. Then, click on the arrow in the drop down list box displaying Total Commit. You will see the list of DLLs loaded for your application.

The DLL Entry Point

Earlier we mentioned the DLL entry point function. So far in our implementation of the example DLLs, we have not included this function, but still our DLLs worked. This is because the C/C++ run-time library provides, by default, a DLL entry point function called **DllMain**. The compiler looks for this function in your DLL code, if it does not find any it uses this default **DllMain** function. This function performs the initialization and termination of the DLL. The actual name of the C/C++ run-time library DLL entrypoint function is **_DllMainCRTStartup**, which initializes the C/C++ run-time library (calling the **_CRT_INIT** function) and then calls **DllMain**. You can provide your own **DllMain** in your DLL code. You may even choose to have your own entrypoint function. In that case, you need to do the job performed by the **_DllMainCRTStartup** function (properly initialize the C/C++ run-time library) and use the linker switch /ENTRY:FunctionName (or, if you are using the Visual C++ Developer Studio, open the **Settings** dialog box from the **Build** menu, click the **Link** tab, select **Output** from the **Category** drop down list, and type the function name in the **Entry-point symbol** text box). The entrypoint function takes three parameters, a module handle, a reason for call (four reasons possible), and a reserved parameter. It returns TRUE(1) to indicate success. The following code is a basic skeleton showing what the definition of **DllMain** might look like. You can add this code as is to all the previous DLLs we implemented, but you will not see any difference in execution because it is not doing anything different from the default **DllMain.**

```
    #include <windows.h>
BOOL WINAPI DllMain( HANDLE hModule,
                     DWORD fdwreason,  LPVOID lpReserved )
{
    switch(fdwreason) {
    case DLL_PROCESS_ATTACH:
    // The DLL is being mapped into process's address space
    //  Do any required initialization on a per application basis, return FALSE if failed
    break;
    case DLL_THREAD_ATTACH:
    // A thread is created. Do any required initialization on a per thread basis
    break;
    case DLL_THREAD_DETACH:
    // Thread exits with  cleanup
    break;
    case DLL_PROCESS_DETACH:
    // The DLL unmapped from process's address space. Do necessary cleanup
```

```
    break;
    }
    return TRUE;
}
```

The *hModule* parameter contains the instance handle of the DLL. You may save this parameter in a global variable so that you can use it in calls that load resources such as **LoadString**.

The *fdwreason* parameter indicates the reason why the operating system is calling **DllMain.** The four reasons are DLL_PROCESS_ATTACH, DLL_THREAD_ATTACH, DLL_THREAD_DETACH, and DLL_PROCESS_DETACH. Each time a DLL is loaded by a new process, **DllMain** is called with DLL_PROCESS_ATTACH. If in this process a thread calls **LoadLibrary** on this DLL, then **DllMain** is called with DLL_THREAD_ATTACH. A **FreeLibrary** call in this thread will call **DllMain** with DLL_THREAD_DETACH. When an application frees the DLL, **DllMain** is called with DLL_PROCESS_DETACH. Refer to online help and "DLLs in Win32" (MSDN Library)

The *lpReserved* parameter is reserved and usually passes a NULL for normal process exiting, such as a call to *FreeLibrary*, unless *ExitProcess* is called.

Let us write some code to demonstrate the use of the **DllMain** function. We are going to modify the pascal.cpp file and build the pascal.dll with the **DllMain** function. The file is modified as shown below.

```
-------------------------------------------------------------------
// File showing use of DllMain function
// DLL source file pascal.cpp
#include <windows.h>
#include <iostream.h>
#include "pascal.h"
HANDLE dllHandle;
BOOL WINAPI DllMain( HANDLE hModule,
                      DWORD fdwreason,  LPVOID lpReserved )
{
    dllHandle = hModule; // Saved for later use
   switch(fdwreason) {
    case DLL_PROCESS_ATTACH:
    // The DLL is being mapped into process's address space
    //  Do any required initialization on a per application basis,
    //  return FALSE if failed
    MessageBox(NULL, "DLL Process Attach", "DLL Message 1", MB_OK);
    break;
    case DLL_THREAD_ATTACH:
    // A thread is created. Do any required initialization on a per
    // thread basis
    MessageBox(NULL, "DLL Thread Attach", "DLL Message 2", MB_OK);
    break;
    case DLL_THREAD_DETACH:
    // Thread exits with  cleanup
    MessageBox(NULL, "DLL Thread Detach", "DLL Message 3", MB_OK);
    break;
    case DLL_PROCESS_DETACH:
    // The DLL unmapped from process's address space. Do necessary
    // cleanup
    MessageBox(NULL, "DLL Process Detach", "DLL Message 4", MB_OK);
    break;
    }
    return TRUE;
}
double __stdcall MyFunc(int a, double b){
return a*b;
}
int CdeclFunc(int a){
return 2*a;
```

```
}
-----------------------------------------------------------------------
```

Build the DLL. Now let us implement a console application with a thread that uses the above DLL with the following source file. For simplicity, the job done in the thread is kept almost the same as the main process. Since we are using the **_beginthread** function, build this application with the multithreaded C run-time library (LIBCMT[D].LIB) or the multithreaded DLL version of the C run-time library(MSVCRT[D].LIB). [D] is for the debug version of the library. You can choose this option in Developer Studio in the project settings for the C/C++ Code Generation category. When you run this application, you will see the message boxes pop up as the process and the thread loads the DLL.

```
-------------------------------------------------------------------------------
// File processattach.cpp
#include <windows.h>
#include <process.h>
#include <iostream.h>
void firstthread(void* dummy);
void main(void)
{
    typedef int ( * lpFunc1)(int);
    typedef double ( * lpFunc2)(int, double);
    HINSTANCE  hLibrary;
    lpFunc1 Func1, Func2;
    lpFunc2 Func3;

    int x=3;
    double y=2.3;
    int a,c;
    double b;

    hLibrary = LoadLibrary("pascaldll.dll"); //Load DLL in main

    if (hLibrary != NULL)
    {
        Func1 =(lpFunc1) GetProcAddress(hLibrary, "CMyFunc");
        if (Func1 != NULL)
            a = ((Func1)(x ));
         else cout << "Error in Func1 call" << endl;
        Func2 =(lpFunc1) GetProcAddress(hLibrary,
                    MAKEINTRESOURCE(2));
        if (Func2 != NULL)
            c = ((Func2)(a*x ));
        else cout << "Error in Func2 call" << endl;

        Func3 =(lpFunc2) GetProcAddress(hLibrary,
                    MAKEINTRESOURCE(1));
        if (Func3 != NULL)
            b = ((Func3)( x, y ));
        else cout << "Error in Func3 call" << endl;
    }
    else cout << "Error in Load Library" << endl;
    cout << "b=" << b << endl;
        cout << "a=" << a << endl;
              cout << "c=" << c << endl;

    _beginthread(firstthread,0,NULL); // Start a thread
    Sleep(10000L);
    cout << "Exit Main Process" << endl;
    FreeLibrary(hLibrary); //Free Library in main
}
void firstthread(void* dummy)
{
        typedef int ( * lpFunc1)(int);
    typedef double ( * lpFunc2)(int, double);
```

```
    HINSTANCE  hLibrary;
    lpFunc1 Func1, Func2;
    char buf[10];

    int x=10;
    int a,c;

    hLibrary = LoadLibrary("pascaldll.dll"); //Load DLL in thread

    if (hLibrary != NULL)
    {
        Func1 =(lpFunc1) GetProcAddress(hLibrary, "CMyFunc");
        if (Func1 != NULL)
            a = ((Func1)(x ));
        else cout << "Error in Func1 call" << endl;
        Func2 =(lpFunc1) GetProcAddress(hLibrary,
                    MAKEINTRESOURCE(2));
        if (Func2 != NULL)
            c = ((Func2)(a*x ));
        else cout << "Error in Func2 call" << endl;

    }
    else cout << "Error in Load Library" << endl;

    _itoa( c, buf, 10 );
    MessageBox(NULL, buf, "Threadtest Result", MB_OK);

    cout << "Now output other values" << endl;
    cout << "a=" <<  a  << endl;
              cout << "Leaving firstthread" << endl;

    FreeLibrary(hLibrary); //Free Library in thread
}

-------------------------------------------------------------------------
```

Applications Sharing Data in DLL

Unlike the 16-bit Windows operating system, the 32-bit Windows operating system does not allow sharing of data in DLLs from different applications. This is because applications are not prevented from sharing memory directly between processes. Two methods may be used for sharing data between processes. One is use of memory-mapped files and the other is creation of a shared data segment. Using memory-mapped files is the recommended method. *Advanced Windows NT*, *The Developer's Guide to the Win32 Application Programming Interface* by J. Richter and "DLLs in Win32" by Randy Kath (MSDN Library) are recommended for further details. The following articles are also helpful.

- "How to Share Data Between Different Mappings of a DLL" (Q125677)
- "Sharing All Data in a DLL" (Q109619)
- "How to Specify Shared and Nonshared Data in a DLL" (Q89817)

Mutual Imports, DLLs and EXE

Exporting and importing between DLLs and EXEs presents some complications because the import library of the other is needed before one is built. But, while building a DLL or EXE that imports from another DLL or EXE, the import library and an export file with the .EXP extension is created when the **dllexport** or the DEF file is used. So, if a DLL or EXE exports a function, you get an import library and a .EXP file even though it generates a linker unresolved symbol

error and no EXE or DLL is produced. But, with the help of this import library you will be able to build the next DLL, which in turn will generate its import library and its DLL file. Now, you can use this import library along with the .EXP file and build the previous DLL or EXE until you get to the first one. If you build from Developer Studio, the .EXP file is automatically used. If you build from the command line, you need to specify the .EXP file in the linker input. In the following example, we start with building the application first, which gives us an import library and a .EXP file, but no EXE. With this import library, we build DLL1, which gives us an import library DLL1.LIB and DLL1.EXP. Now, we can build DLL2 because we have DLL1.LIB. So, we get DLL2.DLL and DLL2.LIB. With DLL2.LIB and DLL1.EXP we can now build DLL1.DLL. As we already have DLL1.LIB, we can also build the EXE. In our example, the application calls DLL1, and DLL1 in turn calls DLL2 and the application, and DLL2 calls DLL1. Here are the files that are used to build the project. This is like going forward until success, and then coming back in reverse order until all modules are built.

```
-------------------------------------------------------------------
// This is the header file for the application when using DLL1--
// applib.h
// DLL1 calls callappfunc function from this application as a callback
// function.
__declspec(dllexport) int callappfunc(int * ptrlib);
__declspec(dllimport) int libglobal;
__declspec(dllimport) void libfuncA(char * buffer);
__declspec(dllimport) int  libfuncB(int const * mainint);
-------------------------------------------------------------------
// This is the application source file  -- appsource.c
// The callappfunc function will be called by DLL1.
// Application calls libfuncA and libfuncB ib DLL1.
#include "applib.h"
#include <stdio.h>
int mainglobal;
int callappfunc(int * ptrlib)
 {
    static int x = 0;
    int m =  *ptrlib;
    printf ("We are starting in the application callappfunc round %d\n",
             ++x);

    m =  libglobal - m * m ;
    printf ("Application callappfunc value: %d\n", m);
    return m;
 }
 void main(void)
  {
       int libreturn;
    char  buf[] = "Let us print this line in DLL1";
    printf ("We are starting in the application\n");
    libfuncA(buf);
    printf("Let us increment the library global and print in app
              :%d\n", ++libglobal);

    mainglobal = 10;
    libreturn = libfuncB( &mainglobal);
    printf("\nPrint the DLL1 returned value in app :%d\n", libreturn);
    printf("Demonstration of Mutual DLL (import) use is complete\n");
  }
-------------------------------------------------------------------
// Library header for DLL1-- lib1.h
// DLL1 calls dllfuncA function in DLL2 and callappfunc in application
__declspec(dllexport) int libglobal;
__declspec(dllexport) void libfuncA(char * buffer);
__declspec(dllexport) int  libfuncB(int const * mainint);
__declspec(dllimport) int callappfunc(int * ptrlib);
__declspec(dllimport) void dllfuncA(char * buffer);
-------------------------------------------------------------------
```

```
// This is a header file of the DLL1 local functions-- lib2.h
 int liblocalA(int x, int y);
-----------------------------------------------------------------------
// This is the DLL source file for DLL1 ---lib1.c
// DLL1 calls callappfunc function in application and
// dllfuncA function in DLL2.
#include <stdio.h>
#include "lib1.h"
#include "lib2.h"
libglobal = 20;
void libfuncA(char * buffer)
 {
      int i =1;
        printf("Printing in DLL1 \n%s\n\n", buffer);
     printf("Value of i in DLL1: %d\n", i);
    buffer = "Changed value(in DLL1) of original content";
    dllfuncA(buffer);
 }
int libfuncB(int const * mainint)
 {
        int returnvalue;
        int  localint;
        int libvalue = 3;
    localint = *mainint;
    returnvalue = liblocalA(localint, libglobal);

    libvalue = callappfunc(&libvalue);
    printf("\nrint returnvalue and libvalue in DLL1:%d, %d\n",
               returnvalue, libvalue);
    return returnvalue;

 }
int liblocalA(int x, int y)
 {
   return (x + y);
 }
-----------------------------------------------------------------------
// Library header lib2A.h for DLL2
// DLL2 calls libfuncB function in DLL1.
__declspec(dllexport) int dllglobal;
__declspec(dllexport) void dllfuncA(char * buffer);
__declspec(dllimport) int  libfuncB(int const * mainint);
-----------------------------------------------------------------------
// This is the  source file lib2.c for DLL2.
// Calls libfuncB function in DLL1.
#include <stdio.h>
#include "lib2A.h"
dllglobal = 35;
void dllfuncA(char * buffer)
 {
        int i =2;
        printf ("We are starting in DLL2\n");
        printf("Printing in DLL2 \n%s\n\n", buffer);
        printf("Value of i in DLL2: %d\n dllglobal:%d\n", i,dllglobal);

        printf ("We are calling into DLL1 from DLL2\n");
        libfuncB(&dllglobal);
 }
-----------------------------------------------------------------------
```

DLL Base Address Conflict

All the system DLLs are usually loaded at the same virtual address regardless of the process. Sometimes there may be a conflict in the DLL base address and you may encounter the following message when a DLL used by your application is being loaded:

```
LDR: Dll xxxx.DLL base 10000000 relocated due to collision with
yyyy.DLL
```

This relocation causes a performance penalty in your application, but as it is a onetime occurrence, the penalty may not be of concern. Each time a DLL is relocated, the fix-ups for that DLL have to be recalculated. This is performed internally within the operating system and cannot be controlled by applications using these DLLs. For new DLLs you are creating, 10000000(hex) is the default base address used by Visual C++. Use the following LINK command on your DLL and look at the entry "image base" to find the base address of your DLL:

```
link -dump -headers  your.dll
```

The base address can be changed by using the linker switch /BASE:address. You should group the base address of your DLLs. You should use an address range that is >0x10000000 and <0x60000000. You can also use the EDITBIN utility and use the /REBASE switch. Query online help on "base" for further information. You may also refer to "Rebasing Win32 DLLs: The Whole Story" (MSDN Library) and to "Dynamically Loading Dynamic-Link Libraries in Windows NT" (Q100635).

Conclusion

In this article, the basic concepts of building a C/C++ DLL with the help of easy-to-understand examples is introduced. A list of references is provided for further studies. The purpose of the article is to break down the initial barrier a new programmer faces when trying to implement a DLL. I hope this article served this purpose.

References

Asche, Ruediger R. "Rebasing Win32 DLLs: The Whole Story." September 1995. (MSDN Library)

Dynamic Link Libraries (DLLs), Microsoft Visual C++ 4.0 Online Documentation.

Gellock, Scot. "Writing DLLs in Win32." (MSDN Library Archive)

Kath, Randy. "DLLs in Win32." September 1992. (MSDN Library)

Richter, J. *Advanced Windows NT*, *The Developer's Guide to the Win32 Application Programming Interface.* Redmond, Washington: Microsoft Press, 1994.

Richter, J. *Advanced Windows, The Developer's Guide to the Win32 API for Windows NT 3.5 and Windows 95.* Redmond, Washington: Microsoft Press, 1995.

Appendix A

The 16-bit DLLs have some difference in implementation considerations in the way they work. First of all, if more than one application loads a DLL, there will be only one data segment. For that reason, the global variables in a DLL can be shared by all the applications using that DLL, unlike the 32-bit DLLs. Since DLLs do not have their own stack segment, while creating a DLL the programmer has to remember that stack segment is not equal to data segment (SS!=DS). The article "How to Export Functions from a 16-bit DLL" (Q148832) describes how to implement

a 16-bit DLL.

Two articles contain more information about 16-bit DLLs and porting 16-bit DLLs to 32-bit DLLs: "General Information Regarding Dynamic-Link Libraries" (Q76476) and "Portable DLL Example Using _declspec() and _export" (Q123870).

References

Gunderson, Bob. "Loading, Initializing, and Terminating a DLL." March 1992. (MSDN Library Archive)

Gunderson, Bob. "Modules, Instances, and Tasks." March 1992. (MSDN Library Archive)

Long, David. "Designing DLLs for Multiple Client Applications, Part 1: Strategy." April 1993. (MSDN Library Archive)

Long, David, and Dan Ruder. "Introduction to Microsoft Windows Dynamic-Link Libraries." August 1992. (MSDN Library Archive)

Long, David, and Dan Ruder. "Mechanics of Dynamic Linking." January 1993. (MSDN Library Archive)

Rogerson, Dale. "Exporting with Class." October 1992. (MSDN Library Archive)

Ruder, Dan. "DLL Anatomy—What's a DLL Made Of?" May 1993. (MSDN Library Archive)
