Microsoft Windows™ 32-Bit DLLs

Rex Jaeschke

Microsoft Windows 32-Bit DLLs

© 1996, 2009 Rex Jaeschke.

Edition: 2.0

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
www.RexJaeschke.com
rex@RexJaeschke.com

Preface

This book introduces students to the design, implementation, and use of Dynamic Link Libraries (DLLs) in a Microsoft Windows 32-bit environment. The primary implementation language used is C, with some C++.

All examples have been tested on Visual C++ V6 and Visual C++ .NET.

The screen dumps were generated on Microsoft Vista.

Reader Assumptions

This is *not* a course in programming in C or C++.

I assume that you know how to use your particular text editor, C/C++ compiler, and debugger. Comments on the use of these utility programs will be limited to points of particular interest to the programmer.

Presentation Style

The approach used in this book might be different from that used in many other books and training courses. Having developed and delivered programming language training for some 14 years, I have found that the best approach for my students is an incremental one. I avoid introducing things that are unnecessary at any given time, thus making examples small, simple, and focused. Many books use GUI and numerous non-trivial library facilities in the first few examples, and certainly in the first chapter. I do not care for this approach, either as a reader or as an educator. Instead, I prefer the student to have an excellent chance of understanding and absorbing small amounts of new material, and reinforcing it with lab sessions, as they progress. The intent here is to eliminate any chance of their being overwhelmed, provided, of course, they meet the prerequisites.

Different styles work for different teachers and different students. I do not suggest that my approach is better than any other is; I simply know that my approach works well, and has formed the basis of my successful seminar business.

Exercises and Solutions

The programs shown in the text are available electronically in a directory tree named Source, where each main DLL described has its own subdirectory, within which each program has its own subdirectory.

Each chapter contains exercises, some of which have the character * following their number. For each exercise so marked, a solution is provided electronically in a directory tree named Labs, where each chapter has its own subdirectory, within which each program has its own subdirectory.[1]

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.

Exercises that are not so marked have no general solution and may require experimentation or research in an implementation's documentation.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

Acknowledgments

Many thanks to those people who reviewed all or part of this book. In particular, students in my C seminars provided useful feedback and located numerous typographical errors.

Rex Jaeschke, November 2009

1. Getting Started

1.1 Introduction

Once we have written a function that has some general utility, it makes sense to reuse that function in other applications whenever possible. We achieve this reuse by placing that function's object code in some sort of object library. And while this approach allows the library function to be shared by an arbitrary number of programs, each executable (EXE file) that calls that function has that function's object code physically linked into it. That is, the executable file on disk contains a copy of the library function. Therefore, if we change the library function in certain ways, all programs that use that function must be re-linked against that library if they are to see the changes. When using this traditional object library model, we say that programs are *statically linked* to that library.

A *dynamic link library* (DLL) is a library of functions organized such that a single in-memory copy can be shared by multiple programs simultaneously. When a program is linked against a DLL, the functions within that DLL that are called by the program are *not* copied into the program's EXE file; instead, they remain part of the DLL. All that gets stored in the EXE file is some information about how to get access to the functions from the DLL at runtime. As a result, the size of the EXE is smaller than it would be if the library were statically linked in. When the program executes, the DLL dynamically attaches to it so the program can call functions residing in that DLL. As the DLL remains physically separate from the EXEs to which it attaches itself, the DLL can be replaced by a newer version containing bug fixes and/or enhancements without any programs having to be relinked, provided the calling sequence for each DLL function remains the same. If a number of processes share a DLL at runtime, combined, they take up less memory and are therefore subject to less swapping.

A program can be statically and/or dynamically linked against multiple libraries, and one DLL can call functions in another DLL.

When a thread calls a DLL function, that function executes in the context of the calling thread. (And as multiple threads can be executing a given DLL function at the same time, that function needs to be reentrant.) Therefore, the DLL function can access any handles opened by its parent thread that have been passed to it as an argument, and any handles created by the DLL function can be passed back to its calling thread. Automatic variables in the DLL function are stored on the calling thread's stack. The memory occupied by the DLL function is mapped into the calling thread's process address space as is any memory allocated by the DLL function during its execution. The DLL function can access global symbols in the calling thread's process and the calling thread can access global symbols defined in the DLL.

Each process to which a DLL attaches itself gets its own copy of the DLL's static data. (Static data is global data, data explicitly declared using the storage class keyword `static`, and string literals.)

1.2 A Simple Example in C

Let’s begin with a simple DLL that contains two global symbols: a variable called `globalVal`, and a function called `getMessage` (see Dll01.c in Dl01\Dll01):

```
#include <stdio.h>

__declspec(dllexport) int globalVal;

__declspec(dllexport) void getMessage(char *pbuffer)
{
    sprintf(pbuffer, "Value = %d", globalVal);
}
```

For the most part, the global variable and function are defined as usual. However, we must explicitly export names we wish to make available to users of this DLL. If we do not, these names will be invisible to the outside world. We extend an external object or function's storage class attributes by using the keyword `__declspec`, as shown. This Microsoft-specific extended storage class is used to declare one of a number of options. The one we use here, `dllexport`, indicates our desire to make the associated names, `globalVal` and `getMessage`, available to the outside world; that is, we export them from the DLL.

A DLL cannot be executed directly; it can only execute in the context of some other program. Therefore, when we build a DLL, we must indicate that it is a DLL. (We achieve this via a linker command-line option or by creating a DLL workspace or project in the IDE.) When the example above is built, three files are created: Dll01.dll (the DLL), Dll01.exp (export information), and Dll01.lib (import information).

Consider the following C program (see Test01.c in Dl01\Test01) that uses this DLL:

```
#include <stdio.h>

__declspec(dllimport) int globalVal;
__declspec(dllimport) void getMessage(char *pbuffer);

int main()
{
    char buffer[20];

    getMessage(buffer);
    printf("%d, %s\n", globalVal, buffer);

    globalVal = -53;
    getMessage(buffer);
    printf("%d, %s\n", globalVal, buffer);
```

2. DLL Initialization and Termination

Whenever a DLL attaches to a new process, the DLL can optionally be notified and special handling can be performed as appropriate. For example, the DLL might need to allocate resources, keep track of its users, or perform various other initialization tasks. A DLL can also be notified when it detaches from a process, as well as when it attaches to, or detaches from, a thread within a process. If a DLL needs this notification, it must have an entry-point function called `DllMain`.[1]

2.1 The DLL

The following DLL contains code to handle process and thread attaches and detaches (see Dll03a.c in Dl03a\Dll03):

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

BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, LPVOID lpvReserved)
{
   DWORD processID, threadID;
   char buffer[81];
   static unsigned int count = 0;

   process_ID = GetCurrentProcessId();
   thread_ID = GetCurrentThreadId();
   sprintf(buffer, "Process ID: %#X, Thread ID: %#X, count: %u",
         (unsigned)processID, (unsigned)threadID, ++count);

   switch (fdwReason) {

   case DLL_PROCESS_ATTACH:
         MessageBox(0, buffer, "DLL_PROCESS_ATTACH",
               MB_OK | MB_ICONINFORMATION);
         break;

   case DLL_PROCESS_DETACH:
         MessageBox(0, buffer, "DLL_PROCESS_DETACH",
               MB_OK | MB_ICONINFORMATION);
         break;
```

[1] It is possible to call this function something else by using the linker option /entry; however, that is not recommended.

```
    case DLL_THREAD_ATTACH:
          MessageBox(0, buffer, "DLL_THREAD_ATTACH",
                MB_OK | MB_ICONINFORMATION);
          break;

    case DLL_THREAD_DETACH:
          MessageBox(0, buffer, "DLL_THREAD_DETACH",
                MB_OK | MB_ICONINFORMATION);
          break;
    }

    return TRUE;
}

__declspec(dllexport) char *scopy(char *pdest, const char *psource)
{
    /* source omitted; it's a version of strcpy */
}

__declspec(dllexport) unsigned long slen(const char *pstring)
{
    /* source omitted; it's a version of strlen */
}
```

The entry-point function must have the argument list as shown, and its type must include the storage class modifier `__stdcall`. (We'll discuss more about this in §5.) In this example, this modifier is hidden inside the macro `WINAPI`, which is defined as a result of including the header windows.h. When `DllMain` is called by the system, the second argument passed to it contains the reason for the call. As shown, this integer value is the subject of a switch statement having four case labels, one for each reason. In this example, the DLL does nothing more than display a window showing the reason this function was called, the calling process and thread ID, and the number of times this function has been called by this process. The latter is made possible by the variable `count`. (Being static, it retains its value across calls to its parent function.) Of course, each process to which a DLL attaches itself gets it own private set of static variables (such as `count`).

If the entry-point function handles a process attach successfully, it should return TRUE. If FALSE is returned, the system will immediately call that function again indicating process detach, and then terminate the process.[1] For process detaches and thread attaches and detaches, the entry-point function's return value is ignored. (The macros TRUE and FALSE are defined by including windows.h.)

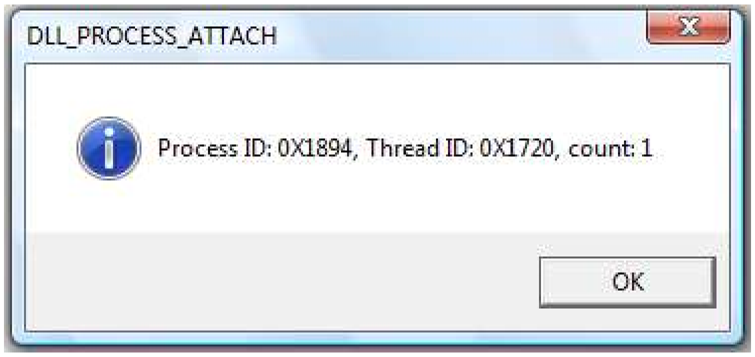
[1] Experiments show that while this is true for the Visual C++ .NET releases, it is *not* the case with Visual C++ V6. (Run program Test01.c in Dl03b\Test01 to see how your implementation handles this situation.)

2.2 A Single-Threaded Application

The following program (see Test01.c in Dl03a\Test01) simply calls the functions `scopy` and `slen`:

```
#include <stdio.h>

__declspec(dllimport) char *scopy(char *pdest, const char *psource);
__declspec(dllimport) unsigned long slen(const char *pstring);

int main()
{
    char buffer[20];

    scopy(buffer, "Hello");
    printf("Text = >%s<, length = %lu\n", buffer, slen(buffer));

    return 0;
}
```

When this program is run, the DLL attaches itself, resulting in the following window's being displayed:


DLL_PROCESS_ATTACH

Process ID: 0X1894, Thread ID: 0X1720, count: 1

And when we acknowledge this, the process writes out the text

```
Text = >Hello<, length = 5
```

4. Run-Time Linking

In the examples thus far, we have used what is called *load-time DLL linking*. Specifically, the DLL has attached itself to a process when that process began executing (that is, when the DLL was loaded into memory) and detached itself from that process when that process terminated. While this approach suffices in most cases, sometimes it is desirable to be able to have a DLL be attached and detached under program control. This is done using *run-time DLL linking*.

With load-time linking, the DLL's LIB file must be made available to the linker, so it can resolve references to public symbols in the DLL. Then when the user process is run, if the DLL is not found, the process is aborted. With run-time linking, the LIB file is not used, as the user process contains no direct references to public symbols in the DLL. When a user process is run, the DLL must exist only if it is loaded manually. And if it can't be found, the process is so notified and it can take the appropriate action. Of course, if the process cannot reasonably continue without the required DLL, it likely will terminate itself, but it does so under programmer control.

Why might we want to load manually a DLL? For the most part, if a process needs a function from a DLL, it might as well be bound to that DLL at link time rather than at runtime. However, consider the case in which the process can be bound to one of several DLLs based on different criteria. While each DLL might have the same set of public names, each DLL performs some different operation. For example, one might support Spanish text, another, German text, and yet another, English text. Or, one might be a production version while another produces debug/trace information.

4.1 An Example

Let's learn the syntax for loading manually a DLL. And along the way, we'll see some more concrete examples of using this approach.

Consider a DLL (see Dll06a.c in Dl06\Dll06a) that contains a simple function called `test`, which takes no arguments and returns no value. This function simply displays a message box saying "Hello from DLL *<DLL-name>*". The `DllMain` function identifies each process and thread, attach and detach, operation by displaying a message box. Consider a second DLL (see Dll06b.c in Dl06\Dll06b) identical to the first except for the DLL-name displayed by function `test`.

The following program (see Test01.c in Dl06\Test01) asks the user to enter the name of the DLL to be loaded at runtime, along with the name of the function within that DLL to be executed:

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

__declspec(dllimport) void test(void);
```

```
int main()
{
   HMODULE dllHandle1, dllHandle2;
   FARPROC dllProcAddress1, dllProcAddress2;
   char dllName1[81], dllName2[81];
   char dllProcName1[81], dllProcName2[81];
   char reply[10];

   printf("Enter path/name of first DLL: ");
   scanf("%80s", dllName1);
   dllHandle1 = LoadLibrary(dllName1);
   if (dllHandle1 == NULL)
   {
         printf("LoadLibrary error loading %s\n", dllName1);
         return 1;
   }
   else
   {
         printf("Loaded DLL %s, dllHandle1 = %p\n", dllName1,
               (void *)dllHandle1);
   }
```

`LoadLibrary` is the function we use to load manually a DLL. Its sole argument is a string containing the DLL's filename, and if that name contains no file type, .DLL is assumed. If `LoadLibrary` fails to find the specified DLL or it finds it and its `DllMain` returns `FALSE`, `LoadLibrary` returns `NULL`, and `GetLastError` reports `ERROR_MOD_NOT_FOUND`. (MOD is an abbreviation for *module*, another term for the library we're trying to load.)

`LoadLibrary` returns a value of type `HMODULE`. On success, the return value is the load address of the DLL.

```
   printf("Enter first DLL's procedure name: ");
   scanf("%80s", dllProcName1);
   dllProcAddress1 = GetProcAddress(dllHandle1, dllProcName1);
   if (dllProcAddress1 == NULL)
   {
         printf("GetProcAddress error on %s", dllProcName1);
         return 2;
   }
   else
   {
         printf("Found procedure %s, dllProcAddress1 = %p\n",
               dllProcName1, (void *)dllProcAddress1);
   }

   dllProcAddress1();      /* same as (*dllProcAddress1)() */
```

Our application contains no direct calls to functions in the DLL. Instead, we have to find out a given function's address by calling GetProcAddress, passing a handle to the DLL and the function's name as a string. If the DLL contains the given function, the address of that function within the DLL is returned. If GetProcAddress cannot find the given symbol in the DLL, it returns NULL, and GetLastError reports ERROR_PROC_NOT_FOUND.

dllProcAddress1 is defined to be an object of type FARPROC. We'll talk more about this type later, but for now, suffice it to say that it's a pointer to a function. Ordinarily, if we have a function pointer p, to call the function to which it points, we use (*p)(), assuming that function has no arguments. However, Standard C permits this to be written as p(); both are equivalent. The abbreviated approach is used in the example above; that is, dllProcAddress1() calls the DLL function to which dllProcAddress1 points.

```
    printf("Free DLL %s (Y/N): ", dllName1);
    scanf("%9s", reply);
    if (reply[0] == 'Y' || reply[0] == 'y')
    {
          if (FreeLibrary(dllHandle1) == FALSE)
          {
                printf("FreeLibrary error on %s\n", dllName1);
                return 3;
          }
          else
          {
                printf("Freed DLL %s\n", dllName1);
          }
    }
```

This program continues by asking us to specify another pair of DLL and function names. However, before doing so, we have the option of releasing the first DLL. This is done by calling FreeLibrary. If FreeLibrary is given a handle to a module that is not currently open (for example, it has already been closed), it fails and returns FALSE, and GetLastError reports ERROR_MOD_NOT_FOUND.

And so the process is repeated as before:

```
    printf("Enter path/name of second DLL: ");
    scanf("%80s", dllName2);
    dllHandle2 = LoadLibrary(dllName2);
    if (dllHandle2 == NULL)
    {
          printf("LoadLibrary error loading %s\n", dllName2);
          return 1;
    }
```