

Multiple Running DLLs with The Same Name

Matt Pietrek

A long time ago, in a column far, far away, I wrote a program for Windows® 3.x called NukeDLL. The purpose of NukeDLL was to let you force a DLL to unload from memory. In those days, it was common for a program to load DLLs via LoadLibrary and subsequently crash, leaving the DLLs orphaned in memory. The problem was especially bad if you were working on the orphaned DLL—the file system wouldn't let you overwrite or delete a file that was in use as a DLL.

The problem of orphaned DLLs exists because, in 16-bit Windows, the list of DLLs is a global entity. Any process can load a DLL, and another process can then use that DLL even though the second process didn't link against or call LoadLibrary on the DLL. The NukeDLL program presented a list of all currently loaded DLLs from which you could select one. NukeDLL would then call FreeLibrary as many times as necessary to remove the DLL from memory.

In Win32®, the orphaned DLL problem really doesn't exist, so there's no need for programs like NukeDLL. Nonetheless, one of the nifty features of NukeDLL was that you could see all loaded DLLs, regardless of who was using them. Win32-based systems associate loaded DLLs with specific processes, so it's possible to see which processes are using the DLLs. The problem is that most utilities only show you a per process module list. As I'll describe later, it can be surprising to see how a DLL that seems completely unneeded by a process still ends up being loaded by the process.

Data Types

With the NukeDLL program as the inspiration, I wanted the basic capability to see a DLL list, along with its associated processes. Why would you need this information? I'll give you just a few reasons, though I'm sure you can come up with more on your own.

First, consider the scenario where you're writing and testing a DLL that includes a hook procedure with systemwide scope (for example, the WH_GETMESSAGE hook that I used for the MouseWheel in my June 1997 column). In this situation, the operating system loads your DLL into any process for which GetMessage is called. This turns out to be quite a few processes, including the shell itself (Explorer.exe).

Now, let's say a bug crops up in your DLL and you fix it. The problem is that when you rebuild the DLL, the linker can't open it to rewrite the file. The DLL is in use by various processes, and is therefore locked by the operating system. Usually, the only recourse is to shut down all of the programs that have your DLL loaded. How do you know you've closed all the necessary programs? You might think that you've covered all the bases and terminated all the relevant processes, but the file system thinks differently. The ability to see which processes have a DLL loaded comes in mighty handy here.

Another need for a list of DLLs and their associated processes is when the classic duplicate DLL problem arises—meaning two DLLs with the same name, but in different directories. For instance, you might be developing a DLL that ultimately will reside in the Windows system directory. However, for the moment you're working on and testing your DLL in a project directory. You make changes to your DLL, it compiles fine, but when you run the test program, your changes don't seem to be in there. After much struggle, you finally slap your forehead in frustration as you realize that the program is loading a different copy of the DLL from another directory. A tool that explicitly shows the loaded DLLs and their associated processes could have short-circuited all that hassle.

Another related scenario is when a program actually has two or more copies of a loaded DLL with the same name, but from different directories. For example, in my September 1997 column I wrote my own WININET.DLL that logged calls to its APIs before passing control to the real WININET.DLL in the system directory. By placing my WININET.DLL in the same directory as Microsoft® Internet Explorer, I caused Internet Explorer to load my DLL rather than the real WININET. My replacement

Multiple Running DLLs with The Same Name

Matt Pietrek

WININET.DLL had to explicitly load the real WININET.DLL. Admittedly, this particular scenario is rare, but it's a real pain to solve problems when they arise in this scenario.

There's also the issue of memory consumption and performance. Normally, when a DLL is used by multiple processes, just one copy of the code, resources, and read-only data needs to be stored in physical memory. Through the magic of paging, the physical pages of memory are shared among the various processes using the DLL. This conservation of memory goes out the window when a DLL loads at different base addresses in multiple processes. The problem is that, because of base relocations in the DLL, most code and data sections are sensitive to where they're loaded. Thus, n copies of a given DLL loaded at n different base addresses will use n times the amount of physical memory. (This is an oversimplification, but I don't want to get sidetracked here.)

Besides additional memory consumption, loading a DLL at different base addresses in multiple processes also increases the time to load it. A DLL has a preferred address where it wants to be loaded. When this happens, the loader doesn't have to apply relocations to the DLL image in memory. When a DLL loads at different addresses in different processes, you're guaranteed that at least one of the DLL instances isn't at the preferred address. Typically a DLL doesn't load at its preferred address because some other DLL has already loaded at that same base address.

The work involved in applying relocations is minimal. Nonetheless, most Microsoft products (including the operating systems) make an effort to prevent DLLs from having conflicting load addresses. The Platform SDK provides the REBASE tool, which allows you to easily prevent DLL load address conflicts in your own DLLs.

With all these justifications for a tool, what's available? Under Win32, the ability to obtain a DLL list, along with associated process information, is strangely absent from the standard programs supplied with SDKs and compilers. Sure, a program like PVIEW can show you a process list from which you can see the DLLs loaded in just that process. Still, this is pretty inconvenient, to say the least!

In the commercial realm, I've only found a few programs that come close to what I want, such as Norton Utilities for Windows NT® and DLL Master by Shaftel Software. There may be similar programs available, but to my knowledge none come with source code. Since this is the Under the Hood column, I decided to show you how to write such a program that works under Windows NT, Windows 95, and Windows 98. It's a great opportunity to revisit some APIs that I've described in previous columns, but that aren't well known.

The ModuleList Program

The program I've written to address the aforementioned issues is called ModuleList. I named the program ModuleList rather than DLLList because, to the operating system, there's no real distinction between EXE files and DLLs. Likewise, there are DLLs that don't use the .DLL extension. Good examples of this are OCXs and Control Panel applets (.CPL files).

The ModuleList user interface is very simple. The majority of the dialog is taken up by a TreeView control (see Figure 1). You can click and drag the borders of the dialog to size it appropriately, and the tree view resizes appropriately. Each top-level node of the tree view contains the name of one instance of a loaded DLL, followed by its parent directory in parentheses. Clicking the Refresh button forces the module list to be rewalked and redisplayed.

When you expand a DLL node in the tree view, you'll get up to three types of subitems: a description of the DLL, the load address and reference count, and the path to an executable that's using the DLL. The description item is only present for DLLs that contain a FileDescription section in their version resources. The reference count is simply the number of referencing processes. Its value is not at all affected if you call LoadLibrary multiple times on a given DLL within a process.

Multiple Running DLLs with The Same Name

Matt Pietrek

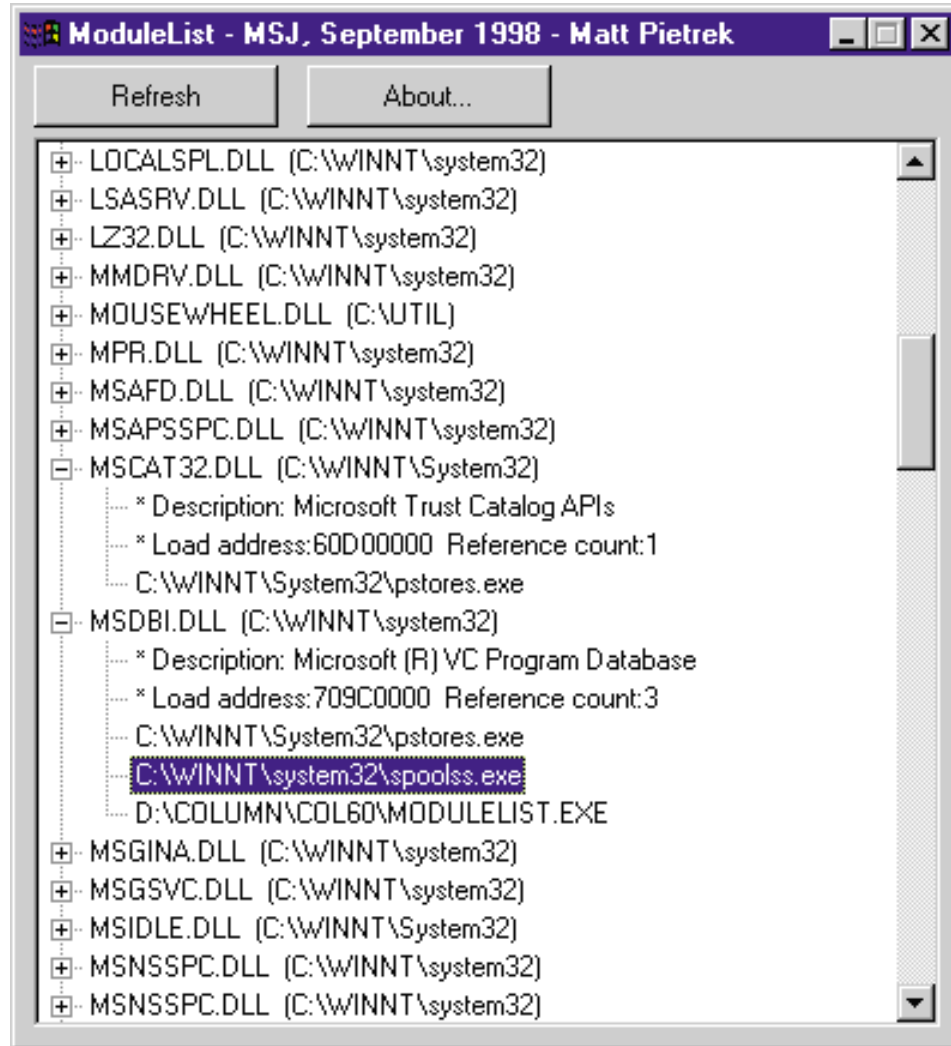


Figure 1 Running ModuleList Tree on Windows NT

Don't be concerned if you see the same DLL appear more than once in the list. ModuleList treats each instance of a DLL with a different load address as a separate instance. I intentionally kept the DLL names in alphabetical order to make it easier to notice when this occurs.

When I set out to write ModuleList, I had a seemingly simple goal: the program should run on as many versions of Windows NT and Windows 9x as possible. This would have been much easier if ModuleList didn't need to run on Windows NT 4.0. However, as I write this column, Windows NT 4.0 is still in widespread use as a development platform.

In an ideal world, I could just use the ToolHelp32 APIs, which provide all the capabilities I need. Alas, ToolHelp32 doesn't exist on Windows NT prior to version 5.0. This presents two problems. First, I can't directly call any ToolHelp32 APIs. Second, a means of getting the equivalent information for Windows NT 4.0 is needed. Luckily, PSAPI.DLL (which I wrote about in my August 1996 column) comes to the rescue.

By using a combination of ToolHelp32 and PSAPI.DLL APIs, I can access all the information I need to

Multiple Running DLLs with The Same Name

Matt Pietrek

build the DLL list. Unfortunately, I can't just call the appropriate APIs directly based upon the operating system ModuleList is running on. By calling the APIs directly, I'd create an implicit reference to those APIs in the ModuleList executable. Since the ToolHelp32 APIs aren't in Windows NT 4.0, and since PSAPI.DLL won't load on Windows 9x, I'd create a program that wouldn't run on any operating system.

The way to overcome this problem is to bite the bullet and go through the drudgery of using GetProcAddress to obtain function pointers to the appropriate APIs, based upon the underlying operating system. In Windows NT 5.0, there's a new feature called Delay Load import descriptors that sounds like it could solve this problem. That is, the operating system lets you put off loading a DLL and hooking up to its APIs until you actually call the API. However, since this feature doesn't exist in Windows 9x and prior versions of Windows NT, it doesn't do you much good now.

The ModuleList Code

The central point for the ModuleList code is ModuleList.CPP (see Figure 2). Most of the code is boilerplate dialog code, which I won't waste time on. Instead, let's focus on the PopulateTree function. After clearing the contents of the tree view, the function resets global instances of a ModuleList class and a ProcessIdToNameMap class. Next, PopulateTree adds data to these class instances by calling the PopulateModuleList_ToolHelp32 or PopulateModuleList_PSAPI functions as necessary. Finally, PopulateTree walks through all the items of the ModuleList class and adds the relevant information to the TreeView control.

Before getting into how the ModuleList and ProcessIdTo-NameMap classes are filled, let's first look at the classes themselves. All of the code for the classes can be found in Module-List-Classes.H and ModuleListClasses.CPP (see Figure 2). The ModuleList class is just a linked list-based container class for ModuleInstance class instances. The ModuleList class has member functions to add a new module, enumerate through all the ModuleInstances, and look up a ModuleInstance given a base address (HMODULE) and file name.

The ModuleInstance class represents each loaded module that has a unique filespec and load address. In addition to storing the HMODULE and filespec, the ModuleInstance class also keeps a list of process IDs that reference the module. There are methods to add a new process ID to the list, enumerate through the process ID list, and retrieve the number of referencing processes.

The final member of this set of classes is the ProcessId-ToNameMap class. Its sole reason for existence is to translate a process ID into a filespec for the executable associated with the process (in essence, the process name). The implementation of this class is rather crude, using a dynamically grown array and brute force scanning algorithms. Yes, using the STL Map class would be more elegant. However, I still spend more time wrestling with STL-induced compiler errors than I save by using the STL in a simple program like this.

The third and final source file from the ModuleList program is ModuleListOSCode.CPP. This is where I isolated all the code that's specific to a particular operating system. There are only two functions in this module: PopulateModuleList_ToolHelp32 and PopulateModuleList_PSAPI. Both functions take references to empty ModuleList and ProcessIdToNameMap class instances and fill them up. Immediately preceding both functions is a series of typedefs. I needed all these typedefs so that I could use GetProcAddress and call the PSAPI and ToolHelp32 APIs through function pointers, thereby avoiding an implicit reference to the APIs.

The PopulateModuleList_ToolHelp32 function looks up the addresses of the five ToolHelp32 APIs it will use. It then creates a ToolHelp32 snapshot of the process list. Using this snapshot, the function iterates through each of the processes. At each stop, it creates a module list snapshot. As the code enumerates through the module list snapshot, it fills in the ModuleList and ProcessIdToNameMap

Multiple Running DLLs with The Same Name

Matt Pietrek

classes that were passed to it. The function is also careful to call Close-Handle on each snapshot when it's done using the snapshot.

The PopulateModuleList_PSAPI function looks up the addresses of the three APIs in PSAPI.DLL that it needs. The code then calls EnumProcesses to obtain an array of process IDs. Next, the code iterates through each of the process IDs and calls OpenProcess to get a corresponding process handle. If a process handle can be obtained (which isn't always the case), the function uses EnumProcess-Modules to get an array of all HMODULEs in the designated process. By itself, an HMODULE in another process is almost useless. Luckily, PSAPI.DLL has the GetModule-FileNameEx API (in both ANSI and Unicode) to retrieve a file name from a process handle and HMODULE combination. I used the ANSI version (GetModuleFileName-ExA) since the rest of the program is ANSI-centric.

In both of these functions, you might notice a minor flaw: not all of the necessary information is collected at one time. With ToolHelp32, multiple module list snapshots are taken during the process enumeration. Likewise, the PSAPI-based function has to use a series of calls to EnumProcess-Modules. The hangup in both cases is that, during the enumeration, a DLL could load or unload. Even worse, a process could start or terminate. Either way, the results wouldn't be entirely consistent. Short of somehow suspending all other processes while the enumeration occurs, this potential loophole can't be avoided.

A DLL Mystery

Take another look at Figure 1. Notice that the highlighted line is spoolss.exe, which is the Windows NT print spooler subsystem. The DLL that it references is MSDBI.DLL. The description for MSDBI.DLL is "Microsoft® VC Program Database." In simpler terms, this is the Visual C++® DLL that reads debug symbol tables. What in the heck would a print spooler need to use a symbol table for? There isn't a reason. As a result, tracking down why spoolss.exe loads MSDBI.DLL is an interesting exercise.

If you're lucky, the program's EXE file or some DLL will implicitly link to the DLL in question. When this happens, you can use a module dependency-listing program to ferret out the connections. One such program is Depends.exe from my February 1997 column. An even better program is Microsoft's own Depends.exe, from the Platform SDK, which I highly recommend. When I wrote my Depends program, I was completely unaware of the Microsoft version. Unlike my version, the Microsoft program has a nice GUI and displays much more information than just module dependencies.

If you run Depends on spoolss.exe, you won't find MSDBI.DLL. That means that the MSDBI.DLL was loaded via LoadLibrary, either directly or indirectly. When I say directly, I mean that somebody explicitly called LoadLibrary on the DLL in question. An indirect load means that LoadLibrary was called for some other DLL, which in turn had an implicit reference to the DLL in question.

With a little thought, you can create numerous scenarios involving a mixture of LoadLibrary calls and implicit references. Figuring out the exact circumstances for a DLL being loaded can be a real nightmare. While a dependency program can help with implicit references, determining DLLs that were loaded via LoadLibrary is trickier. I use a system-level debugger to set a breakpoint on the Load-Library entry point in KERNEL32.DLL. (Under Windows 9x I'd use LoadLibraryA, and under Windows NT, I'd use Load-LibraryExW.) When the breakpoint is hit, you can look at the stack to find the parameter that points to the DLL name being loaded.

Returning to the example at hand, how does MSDBI.DLL get loaded into the spoolss process? When it starts, the initialization code in spoolss calls LoadLibrary to load WIN32SPL.DLL. WIN32SPL.DLL has an implicit reference to LocalSpl.DLL. LocalSpl.DLL uses a single IMAGE-HLP.DLL function, ImageNtHeader. This reference to an IMAGEHLP API is enough to bring in IMAGEHLP.DLL, which in

Multiple Running DLLs with The Same Name

Matt Pietrek

turn implicitly refers to MSDBI.DLL. Quite a twisted path! If you experiment with ModuleList, you'll no doubt find many other strange situations like this. Tracking down the dependencies is a great way to bone up on the various system components and their relationships.

Looking Forward

This issue marks my 60th consecutive monthly column for MSJ. That's five straight years without missing a month (although I've come close on a few occasions). When I first started out, this space was the Windows Questions and Answers column. I covered 16-bit Windows-based programming questions for nearly two years before switching the focus to Win32 programming. In a recent column, I described issues for the forthcoming 64-bit version of Windows NT. That's quite a leap in the evolution of Windows that I've had the privilege to write about.

Under normal circumstances, 60 consecutive months would probably be an MSJ record. However, that honor goes to Paul DiLascia, who started his column before me and is still going strong. I've joked with Paul that someday I'm going to catch up with his streak. However, that won't be happening. For a variety of reasons (all positive), I'm going to cut my column schedule down to once every three months.

Having more time between columns should allow me to focus more of my time on learning, and less on writing. However, in order to make the most of my columns, I need your help. Keep sending me those column topic suggestions. While I won't always be able to help every person with a particular problem, I'm always trying to spot trends where an in-depth Under the Hood column could help out. Thanks for reading!