

# The End of DLL Hell

Rick Anderson  
Microsoft Corporation

## Summary

Describes three types of DLL Hell. (11 printed pages) Also discusses how the DLL Universal Problem Solver (DUPS) package can be used to resolve DLL compatibility problems.

## Introduction

When someone is introduced to a medical doctor in a social setting, they frequently report their most common ailment and expect the doctor to provide a quick remedy. It doesn't matter if the doctor is a plastic surgeon and they have back pains—to them a doctor is a doctor. I often have the same problem. When I'm introduced to a new crowd, someone relates their computer problem and expects me to instantly solve it. The most common problem reported is known as "DLL Hell": After installing a new application, one or more existing programs quit working.

One day my neighbor, Hank, leaned over the fence and said, "I'm trying to run the drive converter tool so I can have a FAT32 file system, and every time I start the tool an error dialog comes up with the message:

```
"The ordinal 968 could not be located in the dynamic-link library MFC42.dll."
```

A quick inspection of his system showed he had MFC42.dll installed in five directories, all of which were the same ancient version. "There's your problem, Hank," I reported. "Some bogus installer stomped on the system version of MFC42.dll with an old version that doesn't have the functionality you need." Hank excitedly fired back, "Can't we just copy MFC42.dll from your computer onto mine?" "Well, we could," I responded, "but DLLs come as matched sets that need to work together. We should install a service pack that updates this DLL so you have a consistent set." I decided to install Microsoft® Visual Studio® 6.0 Service Pack 3, which updates this DLL. But Visual Studio 6.0 SP3 requires a Visual Studio component, so I had to first install Visual Basic 6.0 on Hank's machine, and then install the service pack. When the SP3 upgrade completed, I uninstalled Visual Basic 6.0. The Visual Basic 6.0 uninstall (like all installers, with the exception of Microsoft® Window NT® service pack installers) did not remove the consistent set of shared system DLLs Hank needed; it only removed the Visual Basic 6.0-specific resources.

Hank's headache was caused by a common DLL demon: a rogue install program that doesn't check versions before it copies DLLs into the system directory. Had that install program compared the existing version of MFC42.dll with the installation version, it would have skipped loading MFC42 and not caused his problem. I call this Type I, or replacing a DLL with a previous version, DLL Hell. This is a very common problem for Windows 9x users, especially those who download free software or copy programs from friends. Modern professional software doesn't cause this problem because it always checks versions before stomping on DLLs.

The most common cause of DLL Hell on Windows NT occurs when a newer DLL has unintended or unanticipated functionality changes. I call this Type II, or side-effect, DLL Hell. DLLs are supposed to be backward-compatible, but it's impossible to guarantee 100 percent backward compatibility.

A well-known example of side-effect-induced problems occurred with Service Pack 4 for Windows NT 4.0. Dozens of customers reported that after installing SP4 their application produced an access violation caused by trying to read an invalid memory location. In virtually every case we investigated, the crashing application was using a memory address that had been freed or moved by a call to realloc. With previous versions of the heap manager, programmers were often able to get away with this coding error without the occurrence of an access violation. With SP4, the bug was quickly exposed. Side-effect DLL Hell is not caused by a bug in the newer DLL, but by an application

# The End of DLL Hell

Rick Anderson

Microsoft Corporation

dependent on a side effect of the DLL that might go away. Sometimes the side effect the application is depending on is not a coding error like that of the SP4 memory bug. Many programmers use undocumented data members of a class. Future versions of the class could make that member private or move it to another structure.

The third source of DLL Hell problems arises when a new DLL version is installed that introduces a new bug. While Type III is the least frequent cause of DLL Hell, it does happen.

Why DLLs?

Before we ask why, we should define what a DLL is. DLL is an acronym for dynamic-link library. A DLL is a software component that an application links to at run time. If you write a program that uses the standard `strlen` function (`strlen` returns the length of the string passed in) and dynamically link to `Msvcrt.dll`, your program's EXE will not contain instructions for `strlen`, but it will contain a call instruction to the address of `strlen` in the `Msvcrt.dll`. When your program runs, `Msvcrt.dll` is loaded the first time a method in `Msvcrt.dll` is called.

If you'd like more information on DLLs, search the MSDN Library. There are several good articles on DLLs.

Unix has traditionally shipped applications as completely linked images. Even though Unix now supports shared libraries (similar to DLLs), many Unix vendors continue to ship statically linked images. Because the application is completely self-contained, installing a new library or application can't break existing programs. Application vendors don't need to worry about other software installations breaking their product. By now you should be asking, "If the DLL approach is less robust than the statically linked image approach, why would you want to ship an app that used DLLs?"

**Claim 1:** DLLs save disk space. Almost every application needs to allocate memory and other resources. If you have 501 applications and utilities on your computer that use `Msvcrt.dll`, statically linking these would waste disk space of at least 500 times the size of `Msvcrt.dll`.

Note The actual wasted space could be much more, depending on the file system you are using. FAT16 file systems store files in integral units of fixed-size file blocks. On average, each file wastes one-half a block of disk space because the file did not completely fill the last block.

Reality: DLLs do save disk space, but disk space is practically free and will only get cheaper. Where common code can be safely shared, DLLs are beneficial.

**Claim 2:** DLLs save memory by utilizing a sharing technique called memory mapping. Windows attempts to load a DLL into the global heap and then map the address range of the DLL into the address space of each application that loads it. Ten different processes, all using `Msvcrt.dll`, can share the same instance of `Msvcrt.dll` instead of loading ten copies of it.

**Reality:** Most processes each load a specific DLL and are able to share one global instance of that DLL. Sharing common DLLs does provide a significant savings on memory load. But Windows is not always able to share one instance of a DLL that is loaded by multiple processes.

If you've ever used Microsoft Visual C++® or another debugger you've probably seen a message similar to this:

**LDR: Automatic DLL Relocation in my.exe**

# The End of DLL Hell

Rick Anderson

Microsoft Corporation

```
LDR: Dll abc.dll base 10000000 relocated due to collision with  
C:\xyz\defg.dll
```

When a DLL is created, a base address is specified by the linker suggesting where Windows should load it in a process's 32-bit address space. The default for DLLs created with Visual C++ is 0x10000000. Consider a shared DLL (call it abc.dll) that specifies 0x20000000 for the requested base address. Application a.exe is currently running with abc.dll loaded at 0x20000000. Application b.exe also uses the same abc.dll but has already loaded def.dll at 0x20000000. The OS must then change the base address (referred to as rebasing) of abc.dll to a unique address in process b. Because process b has rebased abc.dll, the OS cannot share abc.dll with the a.exe process; therefore, in this case the abc.dll provides no memory savings over statically linking with abc.dll.

Most DLLs don't need to be rebased, so DLLs do save memory that static linking would duplicate. However, memory continues to drop in price, so the memory savings advantage will become less important. Given that RAM is approximately ten thousand times faster than disk, DLLs are essential for reasonable performance with many applications.

**Claim 3:** Bug fixes are easier to facilitate because a bug is typically isolated to a single DLL. You don't need to ship the entire image, just the fixed DLL.

Reality: Statically linked images don't need to be one atomic image. You can still have the linked libraries as separate files in the same directory in which the application resides. Indeed, this is one approach to DLL Hell that Windows 2000 takes.

The tradeoff between robustness and efficiency depends on the application, the user, and system resources. An ideal situation would allow application vendors, users, and system administrators to decide which is more important: reliability or economy. Because the cost of computer resources continues to plummet, choosing to economize today might be the wrong decision next year.

## The Windows 2000 Approach to DLL Hell

### Windows File Protection

Windows File Protection (WFP) protects system DLLs from being updated or deleted by unauthorized agents. Applications cannot replace system DLLs, only OS update packages such as service packs can do this. System DLLs that can only be updated by a service pack are referred to as protected DLLs. There are approximately 2,800 protected DLLs in Windows 2000.

If you try to copy over a protected DLL into the system directory (winnt\system32) with a DLL of the same name but different version, the copy will appear to be successful and you won't get an error message. Windows 2000 will replace the new DLL with a copy of the original DLL.

Whenever a DLL is put into the system directory Windows 2000 gets a directory change notification event. It then checks to see if the changed DLL was a protected DLL. If the DLL is protected, it then checks to see if the new DLL has a valid digital signature. If the new DLL doesn't have a valid signature, Windows 2000 will copy the original DLL from winnt\system32\dllcache to winnt\system32. WFP prevents installers from modifying system DLLs. Even Microsoft products like Office and Visual Studio won't be able to upgrade the protected DLLs in the system directory.

WFP completely eliminates Type I problems for protected DLLs, and prevents Type II and III problems caused by upgrading/installing applications.

# The End of DLL Hell

Rick Anderson  
Microsoft Corporation

## Private DLLs

Private DLLs are DLLs that are installed with a specific application and used only by that application. For example, suppose you're responsible for the program SuperApp.exe. You've tested SuperApp.exe with Msvcrt.dll version x.x and Sa.dll version y.y. (Sa.dll is not a Microsoft DLL but a third-party DLL distributed with several different applications.) You want to ensure that SuperApp.exe will always use Msvcrt.dll version x.x and Sa.dll version y.y. To accomplish this, your installer puts SuperApp.exe, Msvcrt.dll version x.x, and Sa.dll version y.y into the ./SuperApp directory. You then notify Windows 2000 that SuperApp.exe should use these private DLLs. When SuperApp is run on a Windows 2000 system, it will look in the ./SuperApp directory for DLLs before it looks in the system and path directories. Future service packs that upgrade Msvcrt.dll can't break SuperApp because it's not using the shared version of Msvcrt.dll. Other applications that install different versions of Sa.dll can't affect SuperApp either, because SuperApp has its private version of Sa.dll.

Private DLLs are also referred to as side-by-side DLLs, because a private copy of a DLL is used in one specific application while the system DLL is used by other applications. If you run WordPad and SuperApp concurrently, two copies of Msvcrt.dll are loaded into memory (hence, the "side-by-side" terminology), even if WordPad and SuperApp use the same version of Msvcrt.dll.

There are two approaches to implementing private DLLs. If you're writing a new application or a new component, you give each version a unique version number. You then register each DLL or component in the application directory where you want a private copy. Your application knows to load a private copy of a shared DLL by version information in the application.

The second side-by-side approach is geared toward existing applications. Suppose C:\SuperApp\SuperApp.exe is an existing application that you want to protect from future DLL upgrades or was broken by a service pack upgrade. You simply copy the DLLs you want private to SuperApp into ./SuperApp and create an empty file in this directory called ".\SuperApp.exe.local." Now when SuperApp fires up and finds the .local file it searches the current directory for DLLs and COM servers before it searches the standard path. If your application is broken by a service pack upgrade, you create an install program with the .local file and the older DLLs you need and provide them to your customers.

Both the version-specific (for new applications) and .local (for old apps) side-by-side approaches have the following characteristics:

- DLLs that are in the apps directory are loaded in lieu of system DLLs, even if the loadlibrary path is hard-coded.
- You cannot redirect the 20 KnownDLLs, which are listed in HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs. Most of them cannot run side by side because they need to maintain cross-process state. For example, kernel32, user32, and ole32 cannot be redirected because they have state (kernel objects, window handles, and running object table for local server) that needs to exist across processes. In a future OS release, several of these DLLs will be implemented so they can run side by side and the KnownDLLs list will shrink.
- Potentially fixes all types of DLL Hell. Existing applications will need to determine which DLLs need to be privatized.

The version approach to private DLLs must be implemented by the component or application author. Administrators or installation programs can create the empty .local file to achieve privatized DLLs.

# The End of DLL Hell

Rick Anderson

Microsoft Corporation

The rarer Type III DLL Hell caused by service pack updates introducing new bugs will be virtually eliminated for those customers selecting the Service Pack Track for Windows 2000. (Windows 2000 updates will have two tracks, an SP track with only bug fixes and no new features, and a Point Release Track for customers who want bug fixes and new features.)

Windows 2000 and Windows 98 Second Edition both implement WFP and side-by-side DLLs. Windows NT 4.0 and Windows 95 folks will need to upgrade if they want this protection. See *Implementing Side-by-Side Component Sharing in Applications* for a thorough discussion of private DLLs.

## My Attack on DLL Hell

Because a significant percentage of our support calls involve DLL Hell, I decided to write a tool that would put the burden of analysis on the computer instead of a support engineer. The result is the DLL Universal Problem Solver (DUPS) package, which is fully documented with complete source code in my Microsoft Knowledge Base article Q247957: "SAMPLE: Using the DUPS Package to Resolve DLL Compatibility Problems."

The DUPS package can be used on a single computer or used to track the DLL history of every DLL on the network. It runs on Windows 95, Windows 98, and Windows 2000, and in its simplest mode has no dependencies. The DUPS package consists of the three C++ utilities and a Visual Basic viewer, detailed as follows:

- C++ data-producing server components
- Dlister. Enumerates all the DLLs on your system, recording name, version, size, and date of check. This information can be logged to a text file or a database (Microsoft Access or SQL Server™).
- Dcomp. Compares all the DLLs listed in two text files and produces a third text file containing the differences.
- Dtxt2DB. Reads the text files created with Dlister and Dcomp into the dllHell database.
- DlgDtxt2DB. A graphical user interface (GUI)-based version of Dtxt2DB. This utility was written to facilitate importing Dlister output text files customers send for analysis.

To audit all the DLLs on a network, install the Dlister and Dcomp images on a shared drive available to all networked machines. From the command prompt on any Windows NT machine, create an "at" job to schedule the DLL auditing. I paste the following text in a command window on each Windows NT box I want to audit:

```
at 3:15 /every:M,T,W,Th,F \\manic2\DllHell\Dlister.exe
```

The preceding at command schedules Dlister to run every workday at 3:15 A.M. If you install more than one product in a day, you can run Dlister manually so each product gets audited separately. On my test box, a 450 MHz PII, it takes approximately 90 seconds for Dlister to complete in Text mode.

## Dlister actions

1. On start up, Dlister reads attributes from the initialization file DLLhell.ini for any customizations to the default behavior. You can specify computer-specific or generic attributes. For example, if you specify the connectionString attribute, Dlister will connect to the database directly and will not use Dcomp and Dtxt2DB. Attributes are documented in the Knowledge Base article just cited.

# The End of DLL Hell

Rick Anderson  
Microsoft Corporation

2. Dlister creates an output file containing all the DLL information on the computer. On its first run, Dlister creates computerName\_DLL.txt ("computerName" is the string returned from the Microsoft Win32® call GetComputerName() made by the Dlister image.) Subsequent runs find an existing computerName\_DLL.txt file, so Dlister creates a new file called computerName\_DLL\_new.txt.
3. On Dlister completion, the Dcomp program is executed. Dcomp compares all the DLLs listed in each file. Any DLL changes are written to the file called "computerName\_DLL\_changes\_DateTime.txt." DateTime is a string of the date and time, so guarantee the file is unique. We call this file the changes file. (On the initial run, there is no computerName\_DLL\_new.txt for comparison, so the computerName\_DLL.txt file is copied to the changes file.)
4. On Dcomp completion, the changes file is copied to the DLLhell server. (The server directory, Dlister, and Dcomp output directories can all be specified in the DLLhell.ini file.)
5. After the changes file is copied to the server, computerName\_DLL\_new.txt (if it exists) is renamed computerName\_DLL.txt. If there were no changes to record, computerName\_DLL\_new.txt is simply deleted.
6. The DLLhell server is signaled when a new file is copied into its changes directory and executes the Dtxt2DB program. Dtxt2DB reads the changes file and updates the DLLhell database. The changes file is then deleted. The server uses the Win32 API ReadDirectoryChangesW so it can efficiently wait for new files to process. The SDK sample Fwatch demonstrates this API.

Whew! That's a lot of steps and complexity just to track DLL changes. If you supply a connection string to Dlister, steps 2 through 6 can be skipped. All you do is run Dlister. Why, then, would I go to all the trouble of the remaining steps?

The average workstation has about 1,200 DLLs requiring 1,200 round trips from the client to the server, returning a few hundred bytes of data for each DLL. Each query is fairly expensive, so running Dlister on more than a few machines will swamp the SQL server and the network. Most computers have few to no DLL changes from day to day, but running Dlister connected to the database requires each DLL to be looked up and compared to the current version.

By taking a distributed client/server approach to monitoring DLL changes, the client machine does the vast majority of work and network traffic is reduced to insignificance. This yields a scalable architecture where an old discarded 120 MHz Pentium can easily handle 50 workstations. My DLLhell database is using about 1 MB per monitored workstation. That includes all the DLLs on each machine and all the DLL changes over the last month.

## Viewing Data in the DLLhell Database

The Dllview application was written with Visual Basic and uses Microsoft ActiveX® Data Objects (ADO) to access the DLLhell database. MDAC 2.1 (or later) and Access or SQL Server is required. If you're auditing more than a few computers, SQL Server is strongly recommended. If you don't have SQL Server, you can download the desktop version for free from <http://msdn.microsoft.com/vstudio/downloads/addins/msde/>.

Selecting Compare mode presents a list of audited computers in the DLLhell database. After selecting the two you want to compare, the viewer presents a complete list of all the DLLs that differ in one grid box and all the identical DLLs in another box.

# The End of DLL Hell

Rick Anderson

Microsoft Corporation

Selecting show details lists the differing DLLs one by one, showing the name, version, date, and size of the DLL on each system. Because there are so many DLLs on even the simplest system, you can easily be overwhelmed trying to compare all the DLLs of two machines. If a particular application is behaving differently on two computers, you only need to compare the DLLs the application uses. By selecting Compare List ... from the File menu, a dialog box lets you navigate to a .txt file listing all the DLLs the application has loaded. A text file listing the loaded DLLs can be created with any of the following utilities:

- depends.exe, the Dependency Walker that comes with the Platform SDK.
- ListDLLs, HandleEx, and DLLView, available at <http://www.sysinternals.com/>.

Figure 1 shows Dllview in Detail+Limit mode.

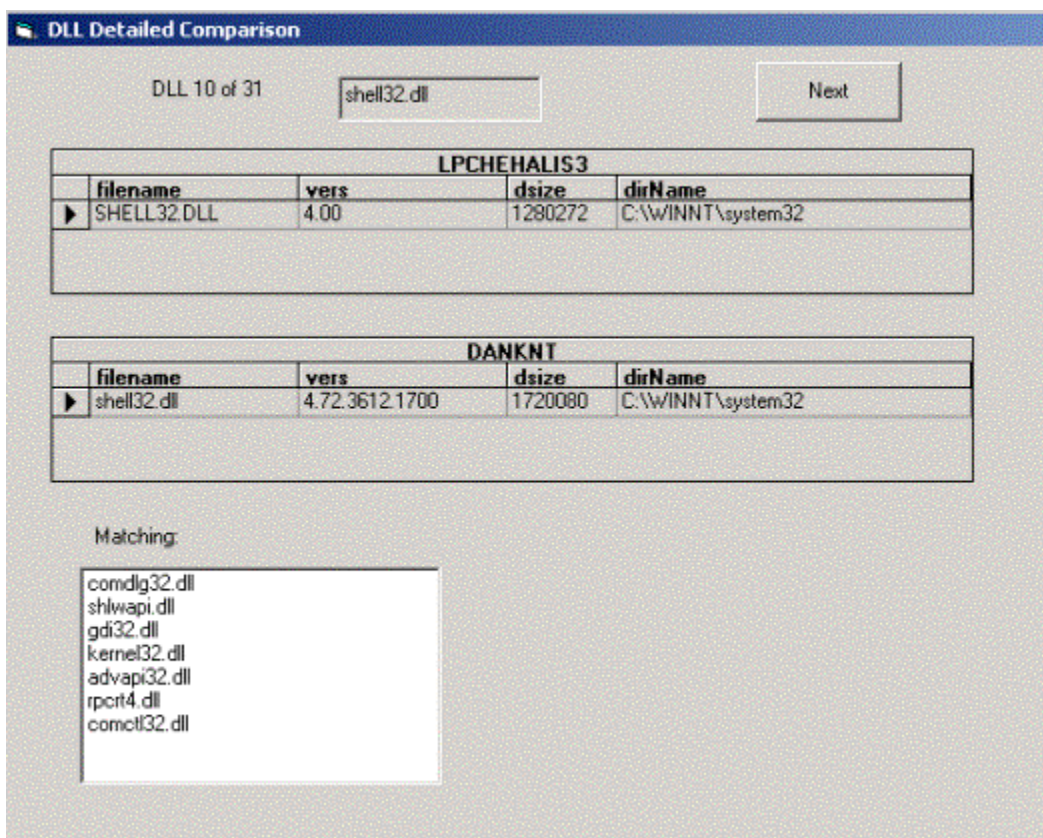


Figure 1. The DLLview application's Detailed View dialog box showing the tenth of 31 DLLs it's comparing from the computers DANKNT and LPCHEHALIS3.

In this view we can see that SHELL32.DLL differs on the two systems, while the Matching list box contains seven DLLs that are identical.

In Time-Date mode, you select a computer and a time or a time range, and the viewer shows all the DLLs that changed for the time selected. This mode is useful in determining which installation caused DLL Hell problems.

# The End of DLL Hell

Rick Anderson

Microsoft Corporation

The Duplicates mode shows all the duplicate DLLs on a system. Most DLL lists customers send me have a large number of duplicates in WINNT\%NtServicePackUninstall, because people generally select the Windows NT SP option to save old DLLs so a complete back out of the service pack is possible. While Windows NT SP upgrades are considered the most robust installers, selecting the uninstall option should set a timer, which a month later asks if it's okay to delete the WINNT\%NtServicePackUninstall directory.

You'll probably be surprised at how many duplicate DLLs are on your computer.

## What DUPS Has Already Accomplished

I've used the DUPS package to solve more than a dozen customer complaints. The easiest problem to solve is when an application works on some computers but not on others. At this point I send the 64 KB DLister and ask the customer to run it on a working and a failing system. This produces two text files (typically 250 KB each) listing all the DLL data for each machine. In order to limit the DLL comparisons to only those DLLs used by the problematic application, I have them send me a text file listing the DLLs loaded by their application using one of the previously listed utilities.

When I receive the three text files, I import the good and bad DLLs list into my DLLhell database and then use DLLviewer to compare each DLL. In about a minute I can report which DLLs differ.

The biggest problem I've had with this approach is that after the customer sends me the DLL data, five minutes later I get the results: The failing computer is using abc.dll version n.2 and the working computer is using abc.dll version n.1. Soon after this I get a glowing e-mail thanking me and saying, "After I copied abc.dll from the working to the failing computer, the problem has been resolved." At this point I'm obligated to call the customer and explain to them that DLLs are not independent, they come in sets. Installing an incomplete set of DLLs may have fixed the current problem but can cause problems in other applications. You've also got an unsupported configuration. The usual response to my pessimistic proclamation is, "Okay, I'll fix it when I have time." For most programmers, "When I have time" is two weeks after never.

## Limitations

The DUPS package cannot solve all inconsistencies where identical images don't execute identically on different computers. It's possible to have two systems that exhibit varying behavior even though they have identical software installed. One possible reason for the inconsistency may be attributed to different hardware. In practice, this is extremely rare if the differing hardware are both using INTEL processors. A bug in a device driver is the more likely reason for the different program behavior.

Thread concurrency bugs are a much more common reason two machines with identical software behave differently. Anytime I suspect a bug is caused by a thread concurrency problem, I immediately run the test program on a dual or quad processor machine. Multithreaded applications with thread concurrency bugs are often very difficult to reproduce on a single processor machine but usually much more easily exposed on a multiprocessor system.

Another common source of application inconsistency between machines occurs when the application reads persistent data. The data could be an .ini file, a database, or the registry. Database and .ini files are usually easy to reconcile, but registry data can be very difficult to compare. If I suspect the registry to be the problem, I run the regmon utility (available for free from <http://www.sysinternals.com/>) and compare the registry values read by the application on each system. Because comparing hundreds of registry values is difficult for normal humans (and virtually impossible for dyslexics like me), my next tool should be a RegistryHell utility.

# The End of DLL Hell

Rick Anderson

Microsoft Corporation

Timing can also influence program outcome and make it difficult or impossible to resolve different program behavior on systems with identical software installed. Many programmers have seen this phenomenon when a faulting program works correctly when run under the debugger. Heap corruption and using uninitialized variables can also be masked by timing changes. In a future article I'll cover PageHeap, the ultimate tool for exposing heap corruption.

DLL Hell is arguably the biggest problem Microsoft faces. Windows 2000 addresses this problem with private DLLs and WFP. Beta testers of WFP and private DLLs have overwhelmingly endorsed these technologies. Private DLLs give Windows 2000 the advantage of shared resources and the flexibility to create essentially static images. This flexibility gives Windows 2000 an advantage over Unix. Microsoft Support has also created the searchable DLL Help database (<http://support.microsoft.com/servicedesks/fileversion/default.asp>) of information about file versions that ship with a selected set of Microsoft products.

## Future Directions

For Windows 2000 systems using NTFS, Dlist2 will simply read the NTFS Change Journal to find all DLL changes that have occurred since the last Dlist2 run. Because the NTFS Change Journal logs all file changes, a Dlist2 using the Change Journal will complete in a matter of seconds instead of the several minutes it now takes to scan all the disks.

## Is DLL Hell Really Over?

The huge increase in robustness WFP and private DLLs provide is one more significant advantage Windows 2000 provides over other operating systems. You now get the resource savings of DLLs with the robustness of static linking. While DLL Hell won't be completely eliminated, it is far less likely to occur. Using DUPS to find the problematic DLL and side-by-side DLLs to fix the problem, DLL version problems will be much easier to find and fix.

None of the code in the DUPS package is very complicated. Because ADO consists of COM objects, I can use the same database API in my low-level/performance-critical C++ routines and in the Visual Basic GUI programs. Where the Visual Basic viewer needed specialized functionality already written in C++ (like reading and parsing the DLL data text files), I was able to wrap the existing C++ code in ATL and call those exposed methods from Visual Basic.