Inside Windows: An In-Depth Look into the Win32

By: Matt Pietrek

Referenced Index:

- **Inside Windows -** An In-Depth Look into the Win32 Portable Executable File Format – Part1. Pag **02**
  - Bridging the Gap Pag **02**
  - Overview of the PE File Format Pag **03**
  - PE File Sections Pag **05**
  - Relative Virtual Addresses Pag **07**
  - The Data Directory Pag **07**
  - Importing Functions Pag **08**
  - PE File Structure Pag **10**
  - The MS-DOS Header Pag **10**
  - The IMAGE_NT_HEADERS Header Pag **10**
  - The Section Table Pag **11**
  - Wrap-up Pag **11**
- **Inside Windows -** An In-Depth Look into the Win32 Portable Executable File Format – Part2. Pag **12**
  - The Exports Section Pag **12**
  - Export Forwarding Pag **14**
  - The Imports Section Pag **14**
  - Binding Pag **16**
  - Delayload Data Pag **17**
  - The Resources Section Pag **18**
  - Base Relocations Pag **19**
  - The Debug Directory Pag **20**
  - The .NET Header Pag **21**
  - TLS Initialization Pag **21**
  - Program Exception Data Pag **23**
  - The PEDUMP Program Pag **23**
  - Wrap-up Pag **24**

- **Appendix** Pag **25**
  - IMAGE_DATA_DIRECTORY Values (Figure 2) Pag **25**
  - IMAGE_FILE_HEADER (Figure 3) Pag **26**
  - IMAGE_FILE_XXX (Figure 4) Pag **26**
  - IMAGE_OPTIONAL_HEADER (Figure 5) Pag **27**
  - The IMAGE_SECTION_HEADER (Figure 6) Pag **29**
  - Flags (Figure 7) Pag **29**
  - Section Names (Figure 8) Pag **30**
  - IMAGE_EXPORT_DIRECTORY Structure Members (Figure 9) Pag **31**
  - KERNEL32 Exports (Figure 10) Pag **31**
  - IMAGE_IMPORT_DESCRIPTOR Structure (Figure 11) Pag **32**
  - ImgDelayDescr Structure (Figure 12) Pag **32**
  - Resources from ADVAPI32.DLL (Figure 13) Pag **32**
  - Fields of IMAGE_DEBUG_DIRECTORY (Figure 14) Pag **33**
  - IMAGE_COR20_HEADER Structure (Figure 15) Pag **33**
  - IMAGE_TLS_DIRECTORY Structure (Figure 16) Pag **34**
  - Command-line Options (Figure 17) Pag **34**
  - RunInit.cpp (Figure 18) Pag **34**
  - ShowSnaps (Figure 19) Pag **40**
  - TLSInit.cpp (Figure 20) Pag **34**
  - Loader APIs (Figure 21) Pag **45**
  - Private Loader APIs (Figure 22) Pag **45**
  - Internal Loader Routines (Figure 23) Pag **46**
  - APIs Forwarded to NTDLL (Figure 24) Pag **48**
  - Binding via LdrpSnapIAT and LdrpSnapThunk for Forwarded.DLL (Figure 25) Pag **48**
  - Pre-binding using SDK Bind (Figure 26) Pag **50**

- **Extras** Pag **52**
  - Under the Hood (Complete Article) Pag **52**
  - Peering Inside the PE: A Tour of the Win32 Portable Executable File Format (Complete Article) Pag **60**
  - What Goes On Inside Windows 2000: Solving the Mysteries of the Loader (Complete Article) Pag **91**

- **References Links** Pag **104**
- **About the Authors** Pag **104**

From the February 2002 issue of MSDN Magazine

Inside Windows

An In-Depth Look into the Win32 Portable Executable File Format – Part1

Matt Pietrek

This article assumes you're familiar with C++ and Win32

Download the code for this article: PE.exe (98KB)

SUMMARY A good understanding of the Portable Executable (PE) file format leads to a good understanding of the operating system. If you know what's in your DLLs and EXEs, you'll be a more knowledgeable programmer. This article, the first of a two-part series, looks at the changes to the PE format that have occurred over the last few years, along with an overview of the format itself.

After this update, the author discusses how the PE format fits into applications written for .NET, PE file sections, RVAs, the DataDirectory, and the importing of functions. An appendix includes lists of the relevant image header structures and their descriptions.

 long time ago, in a galaxy far away, I wrote one of my first articles for *Microsoft Systems Journal* (now *MSDN® Magazine*). The article, "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format," turned out to be more popular than I had expected. To this day, I still hear from people (even within Microsoft) who use that article, which is still available from the MSDN Library. Unfortunately, the problem with articles is that they're static. The world of Win32® has changed quite a bit in the intervening years, and the article is severely dated. I'll remedy that situation in a two-part article starting this month.

You might be wondering why you should care about the executable file format. The answer is the same now as it was then: an operating system's executable format and data structures reveal quite a bit about the underlying operating system. By understanding what's in your EXEs and DLLs, you'll find that you've become a better programmer all around.

Sure, you could learn a lot of what I'll tell you by reading the Microsoft specification. However, like most specs, it sacrifices readability for completeness. My focus in this article will be to explain the most relevant parts of the story, while filling in the hows and whys that don't fit neatly into a formal specification. In addition, I have some goodies in this article that don't seem to appear in any official Microsoft documentation.

Bridging the Gap

Let me give you just a few examples of what has changed since I wrote the article in 1994. Since 16-bit Windows® is history, there's no need to compare and contrast the format to the Win16 New Executable format. Another welcome departure from the scene is Win32s®. This was the abomination that ran Win32 binaries very shakily atop Windows 3.1. Back then, Windows 95 (codenamed "Chicago" at the time) wasn't even released. Windows NT® was still at version 3.5, and the linker gurus at Microsoft hadn't yet started getting aggressive with their optimizations. However, there were MIPS and DEC Alpha implementations of Windows NT that added to the story.

And what about all the new things that have come along since that article? 64-bit Windows introduces its own variation of the Portable Executable (PE) format. Windows CE adds all sorts of new processor types. Optimizations such as delay loading of DLLs, section merging, and binding were still over the horizon. There are many new things to shoehorn into the story.

And let's not forget about Microsoft® .NET. Where does it fit in? To the operating system, .NET executables are just plain old Win32 executable files. However, the .NET runtime recognizes data within these executable files as the metadata and intermediate language that are so central to .NET. In this article, I'll knock on the door of the .NET metadata format, but save a thorough survey of its full splendor for a subsequent article.

And if all these additions and subtractions to the world of Win32 weren't enough justification to remake the article with modern day special effects, there are also errors in the original piece that make me cringe. For example, my description of Thread Local Storage (TLS) support was way out in left field. Likewise, my description of the date/time stamp DWORD used throughout the file format is accurate only if you live in the Pacific time zone!

In addition, many things that were true then are incorrect now. I had stated that the .rdata section wasn't really used for anything important. Today, it certainly is. I also said that the .idata section is a read/write section, which has been found to be most untrue by people trying to do API interception today.

Along with a complete update of the PE format story in this article, I've also overhauled the PEDUMP program, which displays the contents of PE files. PEDUMP can be compiled and run on both the x86 and IA-64 platforms, and can dump both 32 and 64-bit PE files. Most importantly, full source code for PEDUMP is available for download fropm the link at the top of this article, so you have a working example of the concepts and data structures described here.

Overview of the PE File Format

Microsoft introduced the PE File format, more commonly known as the PE format, as part of the original Win32 specifications. However, PE files are derived from the earlier Common Object File Format (COFF) found on VAX/VMS. This makes sense since much of the original Windows NT team came from Digital Equipment Corporation. It was natural for these developers to use existing code to quickly bootstrap the new Windows NT platform.

The term "Portable Executable" was chosen because the intent was to have a common file format for all flavors of Windows, on all supported CPUs. To a large extent, this goal has been achieved with the same format used on Windows NT and descendants, Windows 95 and descendants, and Windows CE.

OBJ files emitted by Microsoft compilers use the COFF format. You can get an idea of how old the COFF format is by looking at some of its fields, which use octal encoding! COFF OBJ files have many data structures and enumerations in common with PE files, and I'll mention some of them as I go along.

The addition of 64-bit Windows required just a few modifications to the PE format. This new format is called PE32+. No new fields were added, and only one field in the PE format was deleted. The remaining changes are simply the widening of certain fields from 32 bits to 64 bits. In most of these cases, you can write code that simply works with both 32 and 64-bit PE files. The Windows header files have the magic pixie dust to make the differences invisible to most C++-based code.

The distinction between EXE and DLL files is entirely one of semantics. They both use the exact same PE format. The only difference is a single bit that indicates if the file should be treated as an EXE or as a DLL. Even the DLL file extension is artificial. You can have DLLs with entirely different extensions for instance .OCX controls and Control Panel applets (.CPL files) are DLLs.

A very handy aspect of PE files is that the data structures on disk are the same data structures used in memory. Loading an executable into memory (for example, by calling LoadLibrary) is primarily a matter of mapping certain ranges of a PE file into the address space. Thus, a data structure like the IMAGE_NT_HEADERS *(which I'll examine later)* is identical on disk and in memory. The key point is that if you know how to find something in a PE file, you can almost certainly find the same information when the file is loaded in memory.

It's important to note that PE files are not just mapped into memory as a single memory-mapped file. Instead, the Windows loader looks at the PE file and decides what portions of the file to map in. This mapping is consistent in that higher offsets in the file correspond to higher memory addresses when mapped into memory. The offset of an item in the disk file may differ from its offset once loaded into memory. However, all the information is present to allow you to make the translation from disk offset to memory offset (see **Figure 1**).

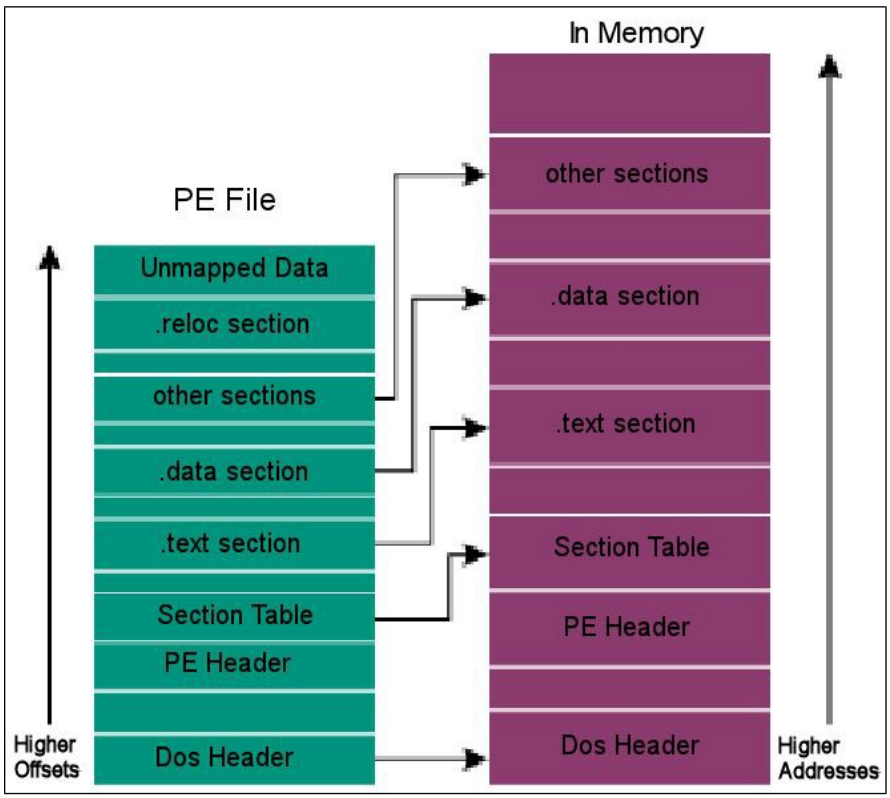

Figure 1 Offsets

When PE files are loaded into memory via the Windows loader, the in-memory version is known as a module. The starting address where the file mapping begins is called an HMODULE. This is a point worth remembering: given an HMODULE, you know what data structure to expect at that address, and you can use that knowledge to find all the other data structures in memory. This powerful capability can be exploited for other purposes such as API interception. (To be completely accurate, an HMODULE isn't the same as the load address under Windows CE, but that's a story for yet another day.)

A module in memory represents all the code, data, and resources from an executable file that is needed by a process. Other parts of a PE file may be read, but not mapped in (for instance, relocations). Some parts may not be mapped in at all, for example, when debug information is placed at the end of the file. A field in the PE header tells the system how much memory needs to be set aside for mapping the executable into memory. Data that won't be mapped in is placed at the end of the file, past any parts that will be mapped in.

The central location where the PE format *(as well as COFF files)* is described is WINNT.H. Within this header file, you'll find nearly every structure definition, enumeration, and #define needed to work with PE files or the equivalent structures in memory. Sure, there is documentation elsewhere. MSDN has the "Microsoft Portable Executable and Common Object File Format Specification," for instance (see the October 2001 MSDN CD under Specifications). But WINNT.H is the final word on what PE files look like.

There are many tools for examining PE files. Among them are Dumpbin from Visual Studio, and Depends from the Platform SDK. I particularly like Depends because it has a very succinct way of examining a file's imports and exports. A great free PE viewer is PEBrowse Professional, from Smidgeonsoft (http://www.smidgeonsoft.com). The PEDUMP program included with this article is also very comprehensive, and does almost everything Dumpbin does.
From an API standpoint, the primary mechanism provided by Microsoft for reading and modifying PE files is IMAGEHLP.DLL.

Before I start looking at the specifics of PE files, it's worthwhile to first review a few basic concepts that thread their way through the entire subject of PE files. In the following sections, I will discuss PE file sections, relative virtual addresses (RVAs), the data directory, and how functions are imported.

PE File Sections

A PE file section represents code or data of some sort. While code is just code, there are multiple types of data. Besides read/write program data (such as global variables), other types of data in sections include API import and export tables, resources, and relocations. Each section has its own set of in-memory attributes, including whether the section contains code, whether it's read-only or read/write, and whether the data in the section is shared between all processes using the executable.

Generally speaking, all the code or data in a section is logically related in some way. At a minimum, there are usually at least two sections in a PE file: one for code, the other for data. Commonly, there's at least one other type of data section in a PE file. I'll look at the various kinds of sections in Part 2 of this article next month.

Each section has a distinct name. This name is intended to convey the purpose of the section. For example, a section called .rdata indicates a read-only data section. Section names are used solely for the benefit of humans, and are insignificant to the operating system. A section named FOOBAR is just as valid as a section called .text. Microsoft typically prefixes their section names with a period, but it's not a requirement. For years, the Borland linker used section names like CODE and DATA.

While compilers have a standard set of sections that they generate, there's nothing magical about them. You can create and name your own sections, and the linker happily includes them in the executable. In Visual C++, you can tell the compiler to insert code or data into a section that you name with #pragma statements. For instance, the statement

```
#pragma data_seg( "MY_DATA" )
```

causes all data emitted by Visual C++ to go into a section called MY_DATA, rather than the default .data section. Most programs are fine using the default sections emitted by the compiler, but occasionally you may have funky requirements which necessitate putting code or data into a separate section.

Sections don't spring fully formed from the linker; rather, they start out in OBJ files, usually placed there by the compiler. The linker's job is to combine all the required sections from OBJ files and libraries into the appropriate final section in the PE file. For example, each OBJ file in your project probably has at least a .text section, which contains code. The linker takes all the sections named .text from the various OBJ files and combines them into a single .text section in the PE file. Likewise, all the sections named .data from the various OBJs are combined into a single .data section in the PE file. Code and data from .LIB files are also typically included in an executable, but that subject is outside the scope of this article.

There is a rather complete set of rules that linkers follow to decide which sections to combine and how. I gave an introduction to the linker algorithms in the July 1997 Under The Hood column in *MSJ*. A section in an OBJ file may be intended for the linker's use, and not make it into the final executable. A section like this would be intended for the compiler to pass information to the linker.

Sections have two alignment values, one within the disk file and the other in memory. The PE file header specifies both of these values, which can differ. Each section starts at an offset that's some multiple of the alignment value. For instance, in the PE file, a typical alignment would be 0x200. Thus, every section begins at a file offset that's a multiple of 0x200.

Once mapped into memory, sections always start on at least a page boundary. That is, when a PE section is mapped into memory, the first byte of each section corresponds to a memory page. On x86 CPUs, pages are 4KB aligned, while on the IA-64, they're 8KB aligned.

The following code shows a snippet of PEDUMP output for the .text and .data section of the Windows XP KERNEL32.DLL. Section Table

1. .text VirtSize: 00074658 VirtAddr: 00001000 raw data offs: 00000400 raw data size: 00074800
2. .data VirtSize: 000028CA VirtAddr: 00076000 raw data offs: 00074C00 raw data size: 00002400

The .text section is at offset 0x400 in the PE file and will be 0x1000 bytes above the load address of KERNEL32 in memory. Likewise, the .data section is at file offset 0x74C00 and will be 0x76000 bytes above KERNEL32's load address in memory.

It's possible to create PE files in which the sections start at the same offset in the file as they start from the load address in memory.

This makes for larger executables, but can speed loading under Windows 9*x* or Windows Me. The default /OPT:WIN98 linker option (introduced in Visual Studio 6.0) causes PE files to be created this way. In Visual Studio® .NET, the linker may or may not use /OPT:NOWIN98, depending on whether the file is small enough.

An interesting linker feature is the ability to merge sections. If two sections have similar, compatible attributes, they can usually be combined into a single section at link time. This is done via the linker /merge switch. For instance, the following linker option combines the .rdata and .text sections into a single section called .text:

```
/MERGE:.rdata=.text
```

The advantage to merging sections is that it saves space, both on disk and in memory. At a minimum, each section occupies one page in memory. If you can reduce the number of sections in an executable from four to three, there's a decent chance you'll use one less page of memory. Of course, this depends on whether the unused space at the end of the two merged sections adds up to a page.

Things can get interesting when you're merging sections, as there are no hard and fast rules as to what's allowed. For example, it's OK to merge .rdata into .text, but you shouldn't merge .rsrc, .reloc, or .pdata into other sections. Prior to Visual Studio .NET, you could merge .idata into other sections. In Visual Studio .NET, this is not allowed, but the linker often merges parts of the .idata into other sections, such as .rdata, when doing a release build.

Since portions of the imports data are written to by the Windows loader when they are loaded into memory, you might wonder how they can be put in a read-only section. This situation works because at load time the system can temporarily set the attributes of the pages containing the imports data to read/write. Once the imports table is initialized, the pages are then set back to their original protection attributes.

Relative Virtual Addresses

In an executable file, there are many places where an in-memory address needs to be specified. For instance, the address of a global variable is needed when referencing it. PE files can load just about anywhere in the process address space. While they do have a preferred load address, you can't rely on the executable file actually loading there. For this reason, it's important to have some way of specifying addresses that are independent of where the executable file loads.

To avoid having hardcoded memory addresses in PE files, RVAs are used. An RVA is simply an offset in memory, relative to where the PE file was loaded. For instance, consider an EXE file loaded at address 0x400000, with its code section at address 0x401000. The RVA of the code section would be:

(target address) 0x401000 - (load address)0x400000 = (RVA)0x1000.

To convert an RVA to an actual address, simply reverse the process: add the RVA to the actual load address to find the actual memory address. Incidentally, the actual memory address is called a Virtual Address (VA) in PE parlance. Another way to think of a VA is that it's an RVA with the preferred load address added in. Don't forget the earlier point I made that a load address is the same as the HMODULE.

Want to go spelunking through some arbitrary DLL's data structures in memory? Here's how. Call GetModuleHandle with the name of the DLL. The HMODULE that's returned is just a load address; you can apply your knowledge of the PE file structures to find anything you want within the module.

The Data Directory

There are many data structures within executable files that need to be quickly located. Some obvious examples are the imports, exports, resources, and base relocations. All of these well-known data structures are found in a consistent manner, and the location is known as the DataDirectory.

The DataDirectory is an array of 16 structures. Each array entry has a predefined meaning for what it refers to. The IMAGE_DIRECTORY_ENTRY_ *xxx* #defines are array indexes into the DataDirectory (from 0 to 15). Figure 2 describes what each of the IMAGE_DATA_DIRECTORY_*xxx* values refers to. A more detailed description of many of the pointed-to data structures will be included in Part 2 of this article.

Importing Functions

When you use code or data from another DLL, you're importing it. When any PE file loads, one of the jobs of the Windows loader is to locate all the imported functions and data and make those addresses available to the file being loaded. I'll save the detailed discussion of data structures used to accomplish this for Part 2 of this article, but it's worth going over the concepts here at a high level.

When you link directly against the code and data of another DLL, you're implicitly linking against the DLL. You don't have to do anything to make the addresses of the imported APIs available to your code. The loader takes care of it all. The alternative is explicit linking. This means explicitly making sure that the target DLL is loaded and then looking up the address of the APIs. This is almost always done via the LoadLibrary and GetProcAddress APIs.

When you implicitly link against an API, LoadLibrary and GetProcAddress-like code still executes, but the loader does it for you automatically. The loader also ensures that any additional DLLs needed by the PE file being loaded are also loaded. For instance, every normal program created with Visual C++® links against KERNEL32.DLL. KERNEL32.DLL in turn imports functions from NTDLL.DLL. Likewise, if you import from GDI32.DLL, it will have dependencies on the USER32, ADVAPI32, NTDLL, and KERNEL32 DLLs, which the loader makes sure are loaded and all imports resolved. (Visual Basic 6.0 and the Microsoft .NET executables directly link against a different DLL than KERNEL32, but the same principles apply.)

When implicitly linking, the resolution process for the main EXE file and all its dependent DLLs occurs when the program first starts. If there are any problems (for example, a referenced DLL that can't be found), the process is aborted.

Visual C++ 6.0 added the delayload feature, which is a hybrid between implicit linking and explicit linking. When you delayload against a DLL, the linker emits something that looks very similar to the data for a regular imported DLL. However, the operating system ignores this data. Instead, the first time a call to one of the delayloaded APIs occurs, special stubs added by the linker cause the DLL to be loaded (if it's not already in memory), followed by a call to GetProcAddress to locate the called API. Additional magic makes it so that subsequent calls to the API are just as efficient as if the API had been imported normally.

Within a PE file, there's an array of data structures, one per imported DLL. Each of these structures gives the name of the imported DLL and points to an array of function pointers. The array of function pointers is known as the import address table (IAT). Each imported API has its own reserved spot in the IAT where the address of the imported function is written by the Windows loader. This last point is particularly important: once a module is loaded, the IAT contains the address that is invoked when calling imported APIs.

The beauty of the IAT is that there's just one place in a PE file where an imported API's address is stored. No matter how many source files you scatter calls to a given API through, all the calls go through the same function pointer in the IAT. Let's examine what the call to an imported API looks like. There are two cases to consider: the efficient way and inefficient way. In the best case, a call to an imported API looks like this:

```
CALL DWORD PTR [0x00405030]
```

If you're not familiar with x86 assembly language, this is a call through a function pointer. Whatever DWORD-sized value is at 0x405030 is where the CALL instruction will send control. In the previous example, address 0x405030 lies within the IAT. The less efficient call to an imported API looks like this:

```
CALL 0x0040100C
(•••)
 0x0040100C:
JMP    DWORD PTR [0x00405030]
```

In this situation, the CALL transfers control to a small stub. The stub is a JMP to the address whose value is at 0x405030. Again, remember that 0x405030 is an entry within the IAT. In a nutshell, the less efficient imported API call uses five bytes of additional code, and takes longer to execute because of the extra JMP.
You're probably wondering why the less efficient method would ever be used. There's a good explanation. Left to its own devices, the compiler can't distinguish between imported API calls and ordinary functions within the same module. As such, the compiler emits a CALL instruction of the form

CALL XXXXXXXX

where *XXXXXXXX* is an actual code address that will be filled in by the linker later. Note that this last CALL instruction isn't through a function pointer. Rather, it's an actual code address. To keep the cosmic karma in balance, the linker needs to have a chunk of code to substitute for *XXXXXXXX*. The simplest way to do this is to make the call point to a JMP stub, like you just saw.

Where does the JMP stub come from? Surprisingly, it comes from the import library for the imported function. If you were to examine an import library, and examine the code associated with the imported API name, you'd see that it's a JMP stub like the one just shown.

What this means is that by default, in the absence of any intervention, imported API calls will use the less efficient form.

Logically, the next question to ask is how to get the optimized form. The answer comes in the form of a hint you give to the compiler.

The declspec(dllimport) function modifier tells the compiler that the function resides in another DLL and that the compiler should generate this instruction

CALL DWORD PTR [XXXXXXXX]

rather than this one: *CALL XXXXXXXX*

In addition, the compiler emits information telling the linker to resolve the function pointer portion of the instruction to a symbol named imp_functionname. For instance, if you were calling MyFunction, the symbol name would be imp_MyFunction. Looking in an import library, you'll see that in addition to the regular symbol name, there's also a symbol with the imp prefix on it. This imp symbol resolves directly to the IAT entry, rather than to the JMP stub.

So what does this mean in your everyday life? If you're writing exported functions and providing a .H file for them, remember to use the declspec(dllimport) modifier with the function:

declspec(dllimport) void Foo(void);

If you look at the Windows system header files, you'll find that they use declspec(dllimport) for the Windows APIs. It's not easy to see this, but if you search for the DECLSPEC_IMPORT macro defined in WINNT.H, and which is used in files such as WinBase.H, you'll see how declspec(dllimport) is prepended to the system API declarations.

PE File Structure

Now let's dig into the actual format of PE files. I'll start from the beginning of the file, and describe the data structures that are present in every PE file. Afterwards, I'll describe the more specialized data structures *(such as imports or resources)* that reside within a PE's sections. All of the data structures that I'll discuss below are defined in WINNT.H, unless otherwise noted.

In many cases, there are matching 32 and 64-bit data structures for example, IMAGE_NT_HEADERS32 and IMAGE_NT_HEADERS64.

These structures are almost always identical, except for some widened fields in the 64-bit versions. If you're trying to write portable code, there are #defines in WINNT.H which select the appropriate 32 or 64-bit structures and alias them to a size-agnostic name (in the previous example, it would be IMAGE_NT_HEADERS).

The structure selected depends on which mode you're compiling for (specifically, whether _WIN64 is defined or not).

You should only need to use the 32 or 64-bit specific versions of the structures if you're working with a PE file with size characteristics that are different from those of the platform you're compiling for.

The MS-DOS Header

Every PE file begins with a small MS-DOS® executable. The need for this stub executable arose in the early days of Windows, before a significant number of consumers were running it. When executed on a machine without Windows, the program could at least print out a message saying that Windows was required to run the executable.

The first bytes of a PE file begin with the traditional MS-DOS header, called an IMAGE_DOS_HEADER. The only two values of any importance are e_magic and e_lfanew. The e_lfanew field contains the file offset of the PE header. The e_magic field (a WORD) needs to be set to the value 0x5A4D. There's a #define for this value, named IMAGE_DOS_SIGNATURE. In ASCII representation, 0x5A4D is MZ, the initials of Mark Zbikowski, one of the original architects of MS-DOS.

The IMAGE_NT_HEADERS Header

The IMAGE_NT_HEADERS structure is the primary location where specifics of the PE file are stored. Its offset is given by the e_lfanew field in the IMAGE_DOS_HEADER at the beginning of the file. There are actually two versions of the IMAGE_NT_HEADER structure, one for 32-bit executables and the other for 64-bit versions. The differences are so minor that I'll consider them to be the same for the purposes of this discussion. The only correct, Microsoft-approved way of differentiating between the two formats is via the value of the Magic field in the IMAGE_OPTIONAL_HEADER (described shortly).

An IMAGE_NT_HEADER is comprised of three fields:

```
typedef struct _IMAGE_NT_HEADERS {
 DWORD Signature;
 IMAGE_FILE_HEADER FileHeader;
 IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

In a valid PE file, the Signature field is set to the value 0x00004550, which in ASCII is "PE00".
A #define, IMAGE_NT_SIGNATURE, is defined for this value. The second field, a struct of type IMAGE_FILE_HEADER, predates PE files. It contains some basic information about the file; most importantly, a field describing the size of the optional data that follows it. In PE files, this optional data is very much required, but is still called the IMAGE_OPTIONAL_HEADER.

Figure 3 shows the fields of the IMAGE_FILE_HEADER structure, with additional notes for the fields. This structure can also be found at the very beginning of COFF OBJ files. Figure 4 lists the common values of IMAGE_FILE_*xxx*. Figure 5 shows the members of the IMAGE_OPTIONAL_HEADER structure.

The DataDirectory array at the end of the IMAGE_OPTIONAL_HEADERs is the address book for important locations within the executable. Each DataDirectory entry looks like this:

```
typedef struct _IMAGE_DATA_DIRECTORY {
 DWORD VirtualAddress; // RVA of the data
 DWORD Size;// Size of the data
};
```

The Section Table

Immediately following the IMAGE_NT_HEADERS is the section table. The section table is an array of IMAGE_SECTION_HEADERs structures. An IMAGE_SECTION_HEADER provides information about its associated section, including location, length, and characteristics. Figure 6 contains a description of the IMAGE_SECTION_HEADER fields. The number of IMAGE_SECTION_HEADER structures is given by the IMAGE_NT_HEADERS.FileHeader.NumberOfSections field.

The file alignment of sections in the executable file can have a significant impact on the resulting file size. In Visual Studio 6.0, the linker defaulted to a section alignment of 4KB, unless /OPT:NOWIN98 or the /ALIGN switch was used. The Visual Studio .NET linker, while still defaulting to /OPT:WIN98, determines if the executable is below a certain size and if that is the case uses 0x200-byte alignment.

Another interesting alignment comes from the .NET file specification. It says that .NET executables should have an in-memory alignment of 8KB, rather than the expected 4KB for x86 binaries. This is to ensure that .NET executables built with x86 entry point code can still run under IA-64. If the in-memory section alignment were 4KB, the IA-64 loader wouldn't be able to load the file, since pages are 8KB on 64-bit Windows.

Wrap-up

That's it for the headers of PE files. In Part 2 of this article I'll continue the tour of portable executable files by looking at commonly encountered sections. Then I'll describe the major data structures within those sections, including imports, exports, and resources. And finally, I'll go over the source for the updated and vastly improved PEDUMP.

From the March 2002 issue of MSDN Magazine

Inside Windows

An In-Depth Look into the Win32 Portable Executable File Format – Part2

Matt Pietrek

This article assumes you're familiar with C++ and Win32

SUMMARY The Win32 Portable Executable File Format (PE) was designed to be a standard executable format for use on all versions of the operating systems on all supported processors. Since its introduction, the PE format has undergone incremental changes, and the introduction of 64-bit Windows has required a few more. Part 1 of this series presented an overview and covered RVAs, the data directory, and the headers. This month in Part 2 the various sections of the executable are explored. The discussion includes the exports section, export forwarding, binding, and delayloading. The debug directory, thread local storage, and the resources sections are also covered..

Last month in Part 1 of this article, I began a comprehensive tour of Portable Executable (PE) files. I described the history of PE files and the data structures that make up the headers, including the section table. The PE headers and section table tell you what kind of code and data exists in the executable and where you should look to find it.

This month I'll describe the more commonly encountered sections. I'll talk a bit about my updated and improved PEDUMP program, available in the February 2002 download. If you're not familiar with basic PE file concepts, you should read Part 1 of this article first.

Last month I described how a section is a chunk of code or data that logically belongs together. For example, all the data that comprises an executable's import tables are in a section. Let's look at some of the sections you'll encounter in executables and OBJs. Unless otherwise stated, the section names in Figure 8 come from Microsoft tools.

The Exports Section

When an EXE exports code or data, it's making functions or variables usable by other EXEs. To keep things simple, I'll refer to exported functions and exported variables by the term "symbols."

At a minimum, to export something, the address of an exported symbol needs to be obtainable in a defined manner. Each exported symbol has an ordinal number associated with it that can be used to look it up. Also, there is almost always an ASCII name associated with the symbol. Traditionally, the exported symbol name is the same as the name of the function or variable in the originating source file, although they can also be made to differ.

Typically, when an executable imports a symbol, it uses the symbol name rather than its ordinal. However, when importing by name, the system just uses the name to look up the export ordinal of the desired symbol, and retrieves the address using the ordinal value. It would be slightly faster if an ordinal had been used in the first place. Exporting and importing by name is solely a convenience for programmers.

The use of the ORDINAL keyword in the Exports section of a .DEF file tells the linker to create an import library that forces an API to be imported by ordinal, not by name.

I'll begin with the IMAGE_EXPORT_DIRECTORY structure, which is shown in Figure 9. The exports directory points to three arrays and a table of ASCII strings. The only required array is the Export Address Table (EAT), which is an array of function pointers that contain the address of an exported function. An export ordinal is simply an index into this array (see **Figure 3**).

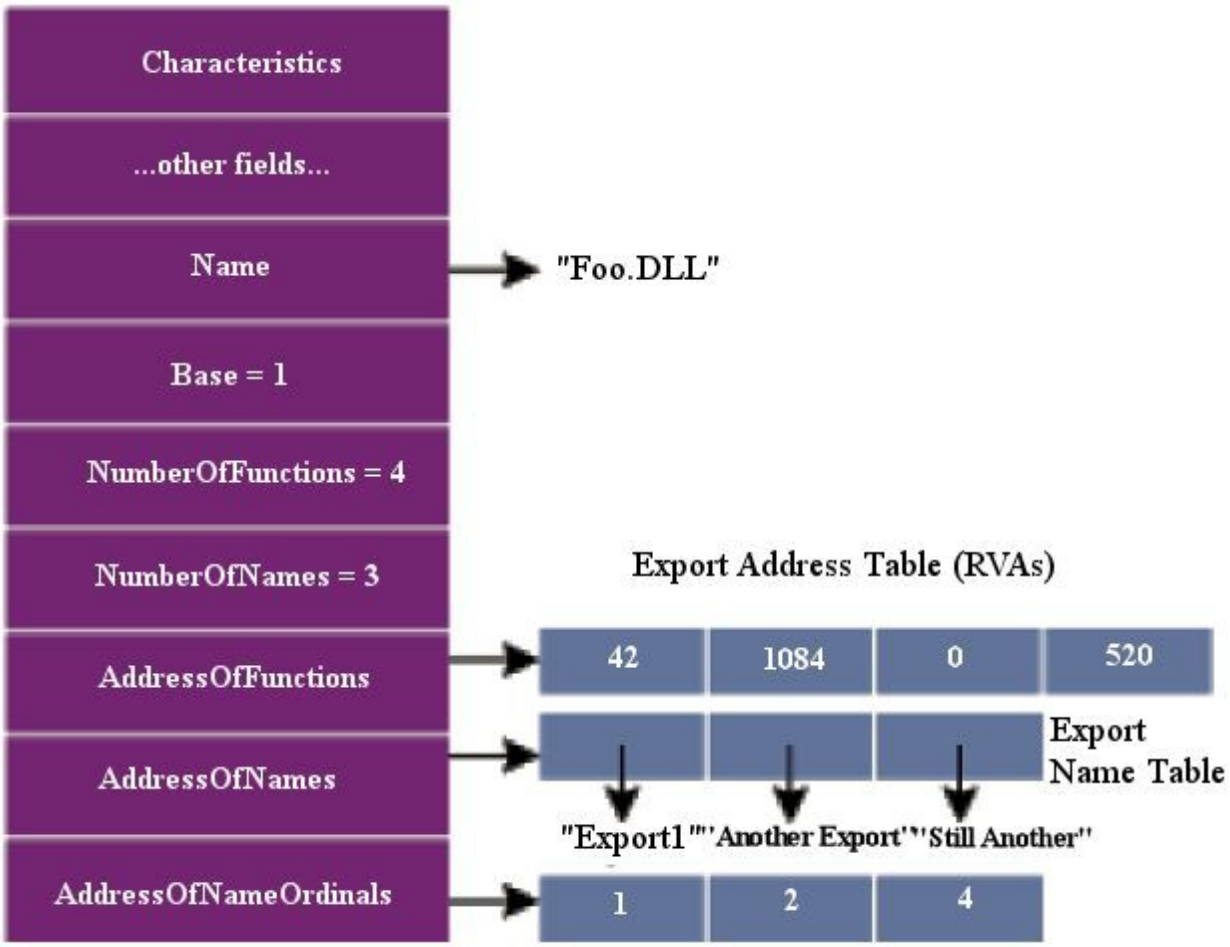

Figure 3 The IMAGE_EXPORT_DIRECTORY Structure

Let's go through an example to show exports at work. Figure 10 shows some of the exports from KERNEL32.DLL.

Let's say you've called GetProcAddress on the AddAtomA API in KERNEL32. The system begins by locating KERNEL32's IMAGE_EXPORT_DIRECTORY. From that, it obtains the start address of the Export Names Table (ENT). Knowing that there are 0x3A0 entries in the array, it does a binary search of the names until it finds the string "AddAtomA".

Let's say that the loader finds AddAtomA to be the second array entry. The loader then reads the corresponding second value from the export ordinal table. This value is the export ordinal of AddAtomA.

Using the export ordinal as an index into the EAT *(and taking into account the Base field value)*, it turns out that AddAtomA is at a relative virtual address *(RVA)* of 0x82C2. Adding 0x82C2 to the load address of KERNEL32 yields the actual address of AddAtomA.

Export Forwarding

A particularly slick feature of exports is the ability to "forward" an export to another DLL. For example, in Windows NT®, Windows® 2000, and Windows XP, the KERNEL32 HeapAlloc function is forwarded to the RtlAllocHeap function exported by NTDLL. Forwarding is performed at link time by a special syntax in the EXPORTS section of the .DEF file. Using HeapAlloc as an example, KERNEL32's DEF file would contain:

```
EXPORTS
•••
HeapAlloc = NTDLL.RtlAllocHeap
```

How can you tell if a function is forwarded rather than exported normally? It's somewhat tricky. Normally, the EAT contains the RVA of the exported symbol. However, if the function's RVA is inside the exports section *(as given by the VirtualAddress and Size fields in the DataDirectory)*, the symbol is forwarded.
When a symbol is forwarded, its RVA obviously can't be a code or data address in the current module. Instead, the RVA points to an ASCII string of the DLL and symbol name to which it is forwarded. In the prior example, it would be NTDLL.RtlAllocHeap.

The Imports Section

The opposite of exporting a function or variable is importing it. In keeping with the prior section, I'll use the term *"symbol"* to collectively refer to imported functions and imported variables.

The anchor of the imports data is the IMAGE_IMPORT_DESCRIPTOR structure. The DataDirectory entry for imports points to an array of these structures. There's one IMAGE_IMPORT_DESCRIPTOR for each imported executable. The end of the IMAGE_IMPORT_DESCRIPTOR array is indicated by an entry with fields all set to 0. Figure 11 shows the contents of an IMAGE_IMPORT_DESCRIPTOR.

Each IMAGE_IMPORT_DESCRIPTOR typically points to two essentially identical arrays. These arrays have been called by several names, but the two most common names are the Import Address Table (IAT) and the Import Name Table (INT). Figure 6 shows an executable importing some APIs from USER32.DLL.

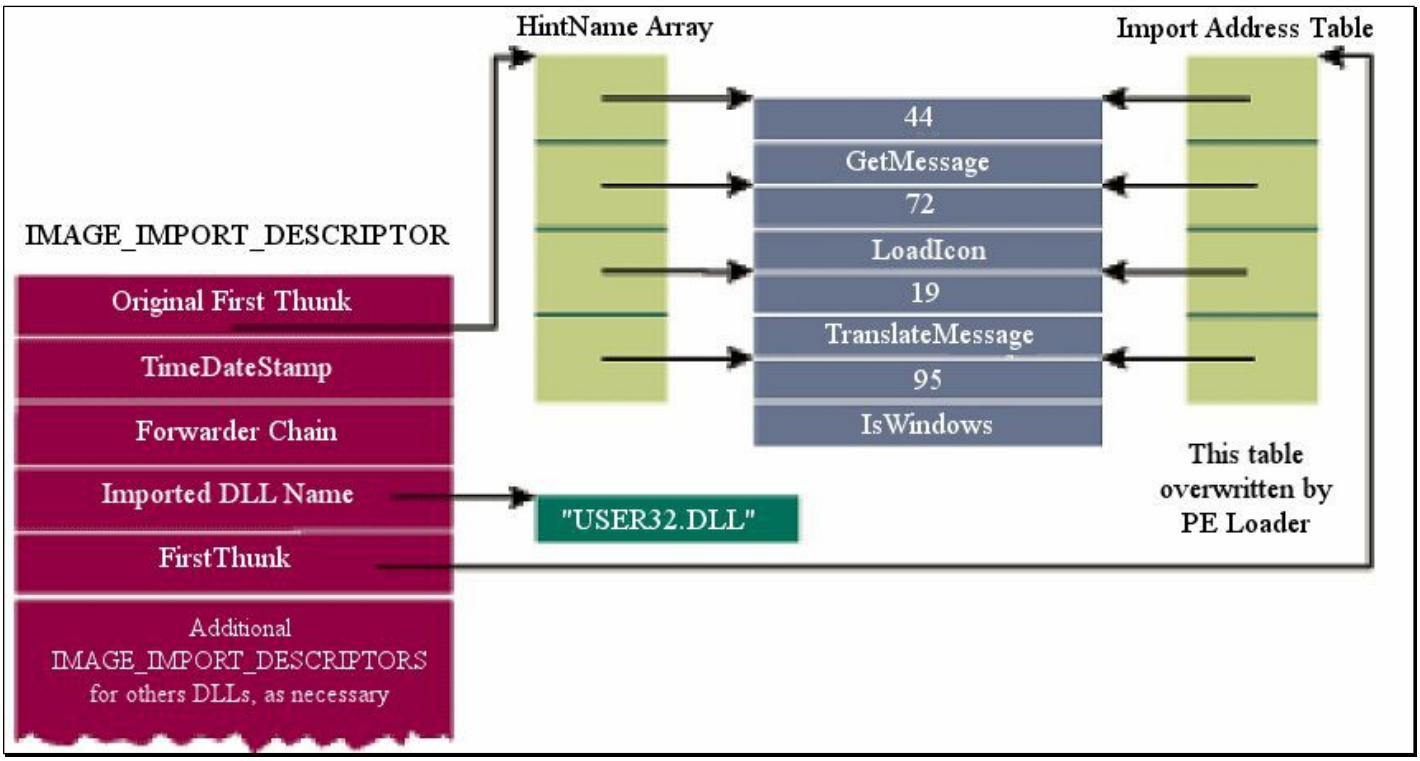

Figure 6 Two Parallel Arrays of Pointers

Both arrays have elements of type IMAGE_THUNK_DATA, which is a pointer-sized union. Each IMAGE_THUNK_DATA element corresponds to one imported function from the executable. The ends of both arrays are indicated by an IMAGE_THUNK_DATA element with a value of zero.

The IMAGE_THUNK_DATA union is a DWORD with these interpretations:

```
DWORD Function; // Memory address of the imported function
DWORD Ordinal; // Ordinal value of imported API
DWORD AddressOfData; // RVA to an IMAGE_IMPORT_BY_NAME with the imported API name
DWORD ForwarderString;// RVA to a forwarder string
```

The IMAGE_THUNK_DATA structures within the IAT lead a dual-purpose life. In the executable file, they contain either the ordinal of the imported API or an RVA to an IMAGE_IMPORT_BY_NAME structure. The IMAGE_IMPORT_BY_NAME structure is just a WORD, followed by a string naming the imported API. The WORD value is a "hint" to the loader as to what the ordinal of the imported API might be. When the loader brings in the executable, it overwrites each IAT entry with the actual address of the imported function. This a key point to understand before proceeding. I highly recommend reading Russell Osterlund's article in this issue which describes the steps that the Windows loader takes.
Before the executable is loaded, is there a way you can tell if an IMAGE_THUNK_DATA structure contains an import ordinal, as opposed to an RVA to an IMAGE_IMPORT_BY_NAME structure?

The key is the high bit of the IMAGE_THUNK_DATA value. If set, the bottom 31 bits *(or 63 bits for a 64-bit executable)* is treated as an ordinal value. If the high bit isn't set, the IMAGE_THUNK_ DATA value is an RVA to the IMAGE_IMPORT_BY_NAME.

The other array, the INT, is essentially identical to the IAT. It's also an array of IMAGE_THUNK_DATA structures. The key difference is that the INT isn't overwritten by the loader when brought into memory. Why have two parallel arrays for each set of APIs imported from a DLL? The answer is in a concept called binding. When the binding process rewrites the IAT in the file (I'll describe this process later), some way of getting the original information needs to remain. The INT, which is a duplicate copy of the information, is just the ticket.

An INT isn't required for an executable to load. However, if not present, the executable cannot be bound. The Microsoft linker seems to always emit an INT, but for a long time, the Borland linker (TLINK) did not. The Borland-created files could not be bound.

In early Microsoft linkers, the imports section wasn't all that special to the linker. All the data that made up an executable's imports came from import libraries. You could see this for yourself by running Dumpbin or PEDUMP on an import library. You'd find sections with names like .idata$3 and .idata$4. The linker simply followed its rules for combining sections, and all the structures and arrays magically fell into place. A few years back, Microsoft introduced a new import library format that creates significantly smaller import libraries at the cost of the linker taking a more active role in creating the import data.

Binding

When an executable is bound *(via the Bind program, for instance)*, the IMAGE_THUNK_DATA structures in the IAT are overwritten with the actual address of the imported function. The executable file on disk has the actual in-memory addresses of APIs in other DLLs in its IAT. When loading a bound executable, the Windows loader can bypass the step of looking up each imported API and writing it to the IAT. The correct address is already there! This only happens if the stars align properly, however. My May 2000 column contains some benchmarks on just how much load-time speed increase you can get from binding executables.

You probably have a healthy skepticism about the safety of executable binding. After all, what if you bind your executable and the DLLs that it imports change? When this happens, all the addresses in the IAT are invalid. The loader checks for this situation and reacts accordingly. If the addresses in the IAT are stale, the loader still has all the necessary information from the INT to resolve the addresses of the imported APIs.

Binding your programs at installation time is the best possible scenario. The BindImage action of the Windows installer will do this for you. Alternatively, IMAGEHLP.DLL provides the BindImageEx API. Either way, binding is good idea. If the loader determines that the binding information is current, executables load faster. If the binding information becomes stale, you're no worse off than if you hadn't bound in the first place.

One of the key steps in making binding effective is for the loader to determine if the binding information in the IAT is current. When an executable is bound, information about the referenced DLLs is placed into the executable. The loader checks this information to make a quick determination of the binding validity. This information wasn't added with the first implementation of binding. Thus, an executable can be bound in the old way or the new way. The new way is what I'll describe here.

The key data structure in determining the validity of bound imports is an IMAGE_BOUND_IMPORT_DESCRIPTOR.

A bound executable contains a list of these structures. Each IMAGE_BOUND_IMPORT_DESCRIPTOR structure represents the time/date stamp of one imported DLL that has been bound against. The RVA of the list is given by the IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT element in the DataDirectory.

The elements of the IMAGE_BOUND_IMPORT_DESCRIPTOR are:

- **TimeDateStamp:** *a DWORD that contains the time/date stamp of the imported DLL.*
- **OffsetModuleName:** *a WORD that contains an offset to a string with the name of the imported DLL. This field is an offset (not an RVA) from the first IMAGE_BOUND_IMPORT_DESCRIPTOR.*
- **NumberOfModuleForwarderRefs:** *a WORD that contains the number of IMAGE_BOUND_FORWARDER_REF structures that immediately follow this structure.*

These structures are identical to the IMAGE_BOUND_IMPORT_DESCRIPTOR except that the last WORD (the NumberOfModuleForwarderRefs) is reserved.

In a simple world, the IMAGE_BOUND_IMPORT_DESCRIPTORs for each imported DLL would be a simple array. But, when binding against an API that's forwarded to another DLL, the validity of the forwarded DLL has to be checked too. Thus, the IMAGE_BOUND_FORWARDER_REF structures are interleaved with the IMAGE_BOUND_IMPORT_DESCRIPTORs.

Let's say you linked against HeapAlloc, which is forwarded to RtlAllocateHeap in NTDLL. Then you ran BIND on your executable. In your EXE, you'd have an IMAGE_BOUND_IMPORT_DESCRIPTOR for KERNEL32.DLL, followed by an IMAGE_BOUND_FORWARDER_REF for NTDLL.DLL.

Immediately following that might be additional IMAGE_ BOUND_IMPORT_DESCRIPTORs for other DLLs you imported and bound against.

Delayload Data

Earlier I described how delayloading a DLL is a hybrid approach between an implicit import and explicitly importing APIs via LoadLibrary and GetProcAddress. Now let's take a look at the data structures and see how delayloading works.

Remember that delayloading is not an operating system feature. It's implemented entirely by additional code and data added by the linker and runtime library. As such, you won't find many references to delayloading in WINNT.H. However, you can see definite parallels between the delayload data and regular imports data.

The delayload data is pointed to by the IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT entry in the DataDirectory. This is an RVA to an array of ImgDelayDescr structures, defined in DelayImp.H from Visual C++. Figure 7 shows the contents. There's one ImgDelayDescr for each delayload imported DLL.

The key thing to glean from ImgDelayDescr is that it contains the addresses of an IAT and an INT for the DLL. These tables are identical in format to their regular imports equivalent, only they're written to and read by the runtime library code rather than the operating system. When you call an API from a delayloaded DLL for the first time, the runtime calls LoadLibrary (if necessary), and then GetProcAddress. The resulting address is stored in the delayload IAT so that future calls go directly to the API.
There is a bit of goofiness about the delayload data that needs explanation. In its original incarnation in Visual C++ 6.0, all ImgDelayDescr fields containing addresses used virtual addresses, rather than RVAs. That is, they contained actual addresses where the delayload data could be found. These fields are DWORDs, the size of a pointer on the x86.

Now fast-forward to IA-64 support. All of a sudden, 4 bytes isn't enough to hold a complete address. Ooops! At

this point, Microsoft did the correct thing and changed the fields containing addresses to RVAs. As shown in Figure 7, I've used the revised structure definitions and names.

There is still the issue of determining whether an ImgDelayDescr is using RVAs or virtual addresses. The structure has a field to hold flag values. When the "1" bit of the grAttrs field is on, the structure members should be treated as RVAs. This is the only option starting with Visual Studio® .NET and the 64-bit compiler. If that bit in grAttrs is off, the ImgDelayDescr fields are virtual addresses.

The Resources Section

Of all the sections within a PE, the resources are the most complicated to navigate. Here, I'll describe just the data structures that are used to get to the raw resource data such as icons, bitmaps, and dialogs. I won't go into the actual format of the resource data since it's beyond the scope of this article.

The resources are found in a section called .rsrc. The IMAGE_DIRECTORY_ENTRY_RESOURCE entry in the DataDirectory contains the RVA and size of the resources. For various reasons, the resources are organized in a manner similar to a file system—with directory and leaf nodes.

The resource pointer from the DataDirectory points to a structure of type IMAGE_RESOURCE_DIRECTORY. The IMAGE_RESOURCE_DIRECTORY structure contains unused Characteristic, TimeDateStamp, and version number fields. The only interesting fields in an IMAGE_RESOURCE_DIRECTORY are the NumberOfNamedEntries and the NumberOfIdEntries.

Following each IMAGE_RESOURCE_DIRECTORY structure is an array of IMAGE_RESOURCE_DIRECTORY_ENTRY structures. Adding the NumberOfNamedEntries and NumberOfIdEntries fields from the IMAGE_RESOURCE_DIRECTORY yields the count of IMAGE_RESOURCE_DIRECTORY_ENTRYs. (*If all these data structure names are painful for you to read, let me tell you, it's also awkward writing about them!)*

A directory entry points to either another resource directory or to the data for an individual resource. When the directory entry points to another resource directory, the high bit of the second DWORD in the structure is set and the remaining 31 bits are an offset to the resource directory. The offset is relative to the beginning of the resource section, not an RVA.

When a directory entry points to an actual resource instance, the high bit of the second DWORD is clear. The remaining 31 bits are the offset to the resource instance (for example, a dialog). Again, the offset is relative to the resource section, not an RVA.

Directory entries can be named or identified by an ID value. This is consistent with resources in an .RC file where you can specify a name or an ID for a resource instance. In the directory entry, when the high bit of the first DWORD is set, the remaining 31 bits are an offset to the string name of the resource. If the high bit is clear, the bottom 16 bits contain the ordinal identifier.

Enough theory! Let's look at an actual resource section and decipher what it means. Figure 8 shows abbreviated PEDUMP output for the resources in ADVAPI32.DLL.

Each line that starts with "ResDir" corresponds to an IMAGE_RESOURCE_DIRECTORY structure.

Following "ResDir" is the name of the resource directory, in parentheses. In this example, there are resource directories named 0, MOFDATA, MOFRESOURCENAME, STRING, C36, RCDATA, and 66. Following the name is the combined number of directory entries *(both named and by ID)*. In this example, the topmost directory has three immediate directory entries, while all the other directories contain a single entry.

In everyday use, the topmost directory is analogous to the root directory of a file system. Each directory entry below the "root" is always a directory in its own right. Each of these second-level directories corresponds to a resource type *(strings tables, dialogs, menus, and so on)*. Underneath each of the second-level "resource type" directories, you'll find third-level subdirectories.

There's a third-level subdirectory for each resource instance. For example, if there were five dialogs, there would be a second-level DIALOG directory with five directory entries beneath it. Each of the five directory entries would themselves be a directory. The name of the directory entry corresponds to the name or ID of the resource instance. Under each of these directory entries is a single item which contains the offset to the resource data. Simple, no?

If you learn more efficiently by reading code, be sure to check out the resource dumping code in PEDUMP *(see the February 2002 code download for this article)*. Besides displaying all the resource directories and their entries, it also dumps out several of the more common types of resource instances such as dialogs.

Base Relocations

In many locations in an executable, you'll find memory addresses. When an executable is linked, it's given a preferred load address. These memory addresses are only correct if the executable loads at the preferred load address specified by the ImageBase field in the IMAGE_FILE_HEADER structure.

If the loader needs to load the DLL at another address, all the addresses in the executable will be incorrect. This entails extra work for the loader. The May 2000 Under The Hood column *(mentioned earlier)* describes the performance hit when DLLs have the same preferred load addresses and how the REBASE tool can help.

The base relocations tell the loader every location in the executable that needs to be modified if the executable doesn't load at the preferred load address. Luckily for the loader, it doesn't need to know any details about how the address is being used. It just knows that there's a list of locations that need to be modified in some consistent way.

Let's look at an x86-based example to make this clear. Say you have the following instruction, which loads the value of a local variable *(at address 0x0040D434)* into the ECX register:

```
00401020: 8B 0D 34 D4 40 00 mov ecx,dword ptr [0x0040D434]
```

The instruction is at address 0x00401020 and is six bytes long. The first two bytes *(0x8B 0x0D)* make up the opcode of the instruction. The remaining four bytes hold a DWORD address *(0x0040D434)*. In this example, the instruction is from an executable with a preferred load address of 0x00400000. The global variable is therefore at an RVA of 0xD434.

If the executable does load at 0x00400000, the instruction can run exactly as is. But let's say that the executable somehow gets loaded at address of 0x00500000. If this happens, the last four bytes of the instruction need to be changed to 0x0050D434.

How can the loader make this change?
The loader compares the preferred and actual load addresses and calculates a delta. In this case, the delta value is 0x00100000. This delta can be added to the value of the DWORD-sized address to come up with the new address of the variable. In the previous example, there would be a base relocation for address 0x00401022, which is the location of the DWORD in the instruction.

In a nutshell, base relocations are just a list of locations in an executable where a delta value needs to be added to the existing contents of memory. The pages of an executable are brought into memory only as they're needed, and the format of the base relocations reflects this. The base relocations reside in a section called .reloc, but the correct way to find them is from the DataDirectory using the IMAGE_DIRECTORY_ENTRY_BASERELOC entry.

Base relocations are a series of very simple IMAGE_BASE_RELOCATION structures. The VirtualAddress field contains the RVA of the memory range to which the relocations belong. The SizeOfBlock field indicates how many bytes make up the relocation information for this base, including the size of the IMAGE_BASE_RELOCATION structure.

Immediately following the IMAGE_BASE_RELOCATION structure is a variable number of WORD values. The number of WORDs can be deduced from the SizeOfBlock field. Each WORD consists of two parts. The top 4 bits indicate the type of relocation, as given by the IMAGE_REL_BASED_xxx values in WINNT.H. The bottom 12 bits are an offset, relative to the VirtualAddress field, where the relocation should be applied.

In the previous example of base relocations, I simplified things a bit. There are actually multiple types of base relocations and methods for how they're applied. For x86 executables, all base relocations are of type IMAGE_REL_BASED_HIGHLOW.

You will often see a relocation of type IMAGE_REL_BASED_ABSOLUTE at the end of a group of relocations.

These relocations do nothing, and are there just to pad things so that the next IMAGE_BASE_RELOCATION is aligned on a 4-byte boundary.

For IA-64 executables, the relocations seem to always be of type IMAGE_REL_BASED_DIR64. As with x86 relocations, there will often be IMAGE_REL_BASED_ABSOLUTE relocations used for padding. Interestingly, although pages in IA-64 EXEs are 8KB, the base relocations are still done in 4KB chunks.

In Visual C++ 6.0, the linker omits relocations for EXEs when doing a release build. This is because EXEs are the first thing brought into an address space, and therefore are essentially guaranteed to load at the preferred load address. DLLs aren't so lucky, so base relocations should always be left in, unless you have a reason to omit them with the /FIXED switch. In Visual Studio .NET, the linker omits base relocations for debug and release mode EXE files.

The Debug Directory

When an executable is built with debug information, it's customary to include details about the format of the information and where it is. The operating system doesn't require this to run the executable, but it's useful for development tools. An EXE can have multiple forms of debug information; a data structure known as the debug directory indicates what's available.

The DebugDirectory is found via the IMAGE_DIRECTORY_ENTRY_DEBUG slot in the DataDirectory.

It consists of an array of IMAGE_DEBUG_DIRECTORY structures (see Figure 9), one for each type of debug information. The number of elements in the debug directory can be calculated using the Size field in the DataDirectory.

By far, the most prevalent form of debug information today is the PDB file. The PDB file is essentially an evolution of CodeView-style debug information. The presence of PDB information is indicated by a debug directory entry of type IMAGE_DEBUG_TYPE_CODEVIEW. If you examine the data pointed to by this entry, you'll find a short CodeView-style header. The majority of this debug data is just a path to the external PDB file. In Visual Studio 6.0, the debug header began with an NB10 signature. In Visual Studio .NET, the header begins with an RSDS.

In Visual Studio 6.0, COFF debug information can be generated with the /DEBUGTYPE:COFF linker switch. This capability is gone in Visual Studio .NET. Frame Pointer Omission (FPO) debug information comes into play with optimized x86 code, where the function may not have a regular stack frame. FPO data allows the debugger to locate local variables and parameters. The two types of OMAP debug information exist only for Microsoft programs. Microsoft has an internal tool that reorganizes the code in executable files to minimize paging. *(Yes, more than the Working Set Tuner can do.)* The OMAP information lets tools convert between the original addresses in the debug information and the new addresses after having been moved.

Incidentally, DBG files also contain a debug directory like I just described. DBG files were prevalent in the Windows NT 4.0 era, and they contained primarily COFF debug information. However, they've been phased out in favor of PDB files in Windows XP.

The .NET Header

Executables produced for the Microsoft .NET environment are first and foremost PE files. However, in most cases normal code and data in a .NET file are minimal. The primary purpose of a .NET executable is to get the .NET-specific information such as metadata and intermediate language (IL) into memory. In addition, a .NET executable links against MSCOREE.DLL.

This DLL is the starting point for a .NET process. When a .NET executable loads, its entry point is usually a tiny stub of code. That stub just jumps to an exported function in MSCOREE.DLL (_CorExeMain or _CorDllMain). From there, MSCOREE takes charge, and starts using the metadata and IL from the executable file. This setup is similar to the way apps in Visual Basic (prior to .NET) used MSVBVM60.DLL.

The starting point for .NET information is the IMAGE_COR20_HEADER structure, currently defined in CorHDR.H from the .NET Framework SDK and more recent versions of WINNT.H. The IMAGE_COR20_HEADER is pointed to by the IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR entry in the DataDirectory. Figure 10 shows the fields of an IMAGE_COR20_HEADER. The format of the metadata, method IL, and other things pointed to by the IMAGE_COR20_HEADER will be described in a subsequent article.

TLS Initialization

When using thread local variables declared with declspec(thread), the compiler puts them in a section named .tls. When the system sees a new thread starting, it allocates memory from the process heap to hold the thread local variables for the thread. This memory is initialized from the values in the .tls section. The system also puts a pointer to the allocated memory in the TLS array, pointed to by FS: [2Ch] *(on the x86 architecture)*.

The presence of thread local storage (TLS) data in an executable is indicated by a nonzero IMAGE_DIRECTORY_ENTRY_TLS entry in the DataDirectory. If nonzero, the entry points to an IMAGE_TLS_DIRECTORY structure, shown in Figure 11.

It's important to note that the addresses in the IMAGE_TLS_DIRECTORY structure are virtual addresses, not RVAs. Thus, they will get modified by base relocations if the executable doesn't load at its preferred load address. Also, the IMAGE_TLS_DIRECTORY itself is not in the .tls section; it resides in the .rdata section.

Program Exception Data

Some architectures (including the IA-64) don't use frame-based exception handling, like the x86 does; instead, they used table-based exception handling in which there is a table containing information about every function that might be affected by exception unwinding.

The data for each function includes the starting address, the ending address, and information about how and where the exception should be handled. When an exception occurs, the system searches through the tables to locate the appropriate entry and handles it. The exception table is an array of IMAGE_RUNTIME_FUNCTION_ENTRY structures.

The array is pointed to by the IMAGE_DIRECTORY_ENTRY_EXCEPTION entry in the DataDirectory.

The format of the IMAGE_RUNTIME_FUNCTION_ENTRY structure varies from architecture to architecture. For the IA-64, the layout looks like this:

```
DWORD BeginAddress;
DWORD EndAddress;
DWORD UnwindInfoAddress;
```

The format of the UnwindInfoAddress data isn't given in WINNT.H. However, the format can be found in Chapter 11 of the "IA-64 Software Conventions and Runtime Architecture Guide" from Intel.

The PEDUMP Program

My PEDUMP program *(available for download with Part 1 of this article)* is significantly improved from the 1994 version. It displays every data structure described in this article, including:

- IMAGE_NT_HEADERS
- Imports / Exports
- Resources
- Base relocations
- Debug directory
- Delayload imports
- Bound import descriptors
- IA-64 exception handling tables
- TLS initialization data
- .NET runtime header

In addition to dumping PE executables, PEDUMP can also dump COFF format OBJ files, COFF import libraries *(new and old formats)*, COFF symbol tables, and DBG files.

PEDUMP is a command-line program. Running it without any options on one of the file types just described leads to a default dump that contains the more useful data structures. There are several command-line options for additional output (see Figure 12).

The PEDUMP source code is interesting for a couple of reasons. It compiles and runs as either a 32 or 64-bit executable. So if you have an Itanium box handy, give it a whirl! In addition, PEDUMP can dump both 32 and 64-bit executables, regardless of how it was compiled. In other words, the 32-bit version can dump 32 and 64-bit files, and the 64-bit version can dump 32 and 64-bit files.

In thinking about making PEDUMP work on both 32 and 64-bit files, I wanted to avoid having two copies of every function, one for the 32-bit form of a structure and another for the 64-bit form. The solution was to use C++ templates.

In several files (EXEDUMP.CPP in particular), you'll find various template functions. In most cases, the template function has a template parameter that expands to either an IMAGE_NT_HEADERS32 or IMAGE_NT_HEADERS64. When invoking these functions, the code determines the 32 or 64-bitness of the executable file and calls the appropriate function with the appropriate parameter type, causing an appropriate template expansion.

With the PEDUMP sources, you'll find a Visual C++ 6.0 project file. Besides the traditional x86 debug and release configurations, there's also a 64-bit build configuration. To get this to work, you'll need to add the path to the 64-bit tools *(currently in the Platform SDK)* at the top of the Executable path under the Tools | Options | Directories tab. You'll also need to make sure that the proper paths to the 64-bit Include and Lib directories are set properly. My project file has correct settings for my machine, but you may need to change them to build on your machine.

In order to make PEDUMP as complete as possible, it was necessary to use the latest versions of the Windows header files. In the June 2001 Platform SDK which I developed against, these files are in the .\include\prerelease and .\Include\Win64\crt\ directories. In the August 2001 SDK, there's no need to use the prerelease directories, since WINNT.H has been updated. The essential point is that the code does build. You may just need to have a recent enough Platform SDK installed or modify the project directories if building the 64-bit version.

Wrap-up

The Portable Executable format is a well-structured and relatively simple executable format. It's particularly nice that PE files can be mapped directly into memory so that the data structures on disk are the same as those Windows uses at runtime. I've also been surprised at how well the PE format has held up with all the various changes that have been thrown at it in the past 10 years, including the transition to 64-bit Windows and .NET.
Although I've covered many aspects of PE files, there are still topics that I haven't gotten to. There are flags, attributes, and data structures that occur infrequently enough that I decided not to describe them here. However, I hope that this "big picture" introduction to PE files has made the Microsoft PE specifications easier for you to understand.

Appendix

Figure 2 IMAGE_DATA_DIRECTORY Values

Value	Description
IMAGE_DIRECTORY_ENTRY_EXPORT	Points to the exports (an IMAGE_EXPORT_DIRECTORY structure).
IMAGE_DIRECTORY_ENTRY_IMPORT	Points to the imports (an array of IMAGE_IMPORT_DESCRIPTOR structures).
IMAGE_DIRECTORY_ENTRY_RESOURCE	Points to the resources (an IMAGE_RESOURCE_DIRECTORY structure.
IMAGE_DIRECTORY_ENTRY_EXCEPTION	Points to the exception handler table (an array of IMAGE_RUNTIME_FUNCTION_ENTRY structures). CPU-specific and for table-based exception handling. Used on every CPU except the x86.
IMAGE_DIRECTORY_ENTRY_SECURITY	Points to a list of WIN_CERTIFICATE structures, defined in WinTrust.H. Not mapped into memory as part of the image. Therefore, the VirtualAddress field is a file offset, rather than an RVA.
IMAGE_DIRECTORY_ENTRY_BASERELOC	Points to the base relocation information.
IMAGE_DIRECTORY_ENTRY_DEBUG	Points to an array of IMAGE_DEBUG_DIRECTORY structures, each describing some debug information for the image. Early Borland linkers set the Size field of this IMAGE_DATA_DIRECTORY entry to the number of structures, rather than the size in bytes. To get the number of IMAGE_DEBUG_DIRECTORYs, divide the Size field by the size of an IMAGE_DEBUG_DIRECTORY.
IMAGE_DIRECTORY_ENTRY_ARCHITECTURE	Points to architecture-specific data, which is an array of IMAGE_ARCHITECTURE_HEADER structures. Not used for x86 or IA-64, but appears to have been used for DEC/Compaq Alpha.
IMAGE_DIRECTORY_ENTRY_GLOBALPTR	The VirtualAddress field is the RVA to be used as the global pointer (gp) on certain architectures. Not used on x86, but is used on IA-64. The Size field isn't used. See the November 2000 Under The Hood column for more information on the IA-64 gp.
IMAGE_DIRECTORY_ENTRY_TLS	Points to the Thread Local Storage initialization section.
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG	Points to an IMAGE_LOAD_CONFIG_DIRECTORY structure. The information in an IMAGE_LOAD_CONFIG_DIRECTORY is specific to Windows NT, Windows 2000, and Windows XP (for example, the GlobalFlag value). To put this structure in your executable, you need to define a global structure with the name __load_config_used, and of type IMAGE_LOAD_CONFIG_DIRECTORY. For non-x86 architectures, the symbol name needs to be _load_config_used (with a single underscore). If you do try to include an IMAGE_LOAD_CONFIG_DIRECTORY, it can be tricky to get the name right in your C++ code. The symbol name that the linker sees must be exactly: __load_config_used (with two underscores). The C++ compiler adds an underscore to global symbols. In addition, it decorates global symbols with type information. So, to get everything right, in your C++ code, you'd have something like this: extern "C" IMAGE_LOAD_CONFIG_DIRECTORY _load_config_used = {...}
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT	Points to an array of IMAGE_BOUND_IMPORT_DESCRIPTORs, one for each DLL that this image has bound against. The timestamps in the array entries allow the loader to quickly determine whether the binding is fresh. If stale, the loader ignores the binding information and resolves the imported APIs normally.
IMAGE_DIRECTORY_ENTRY_IAT	Points to the beginning of the first Import Address Table (IAT). The IATs for each imported DLL appear sequentially in memory. The Size field indicates the total size of all the IATs. The loader uses this address and size

	to temporarily mark the IATs as read-write during import resolution.
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT	Points to the delayload information, which is an array of CImgDelayDescr structures, defined in DELAYIMP.H from Visual C++. Delayloaded DLLs aren't loaded until the first call to an API in them occurs. It's important to note that Windows has no implicit knowledge of delay loading DLLs. The delayload feature is completely implemented by the linker and runtime library.
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR	This value has been renamed to IMAGE_DIRECTORY_ENTRY_COMHEADER in more recent updates to the system header files. It points to the top-level information for .NET information in the executable, including metadata. This information is in the form of an IMAGE_COR20_HEADER structure.

Figure 3 IMAGE_FILE_HEADER

Size	Field	Description
WORD	Machine	The target CPU for this executable. Common values are: IMAGE_FILE_MACHINE_I386 0x014c // Intel 386 IMAGE_FILE_MACHINE_IA64 0x0200 // Intel 64
WORD	NumberOfSections	Indicates how many sections are in the section table. The section table immediately follows the IMAGE_NT_HEADERS.
DWORD	TimeDateStamp	Indicates the time when the file was created. This value is the number of seconds since January 1, 1970, Greenwich Mean Time (GMT). This value is a more accurate indicator of when the file was created than is the file system date/time. An easy way to translate this value into a human-readable string is with the _ctime function (which is time-zone-sensitive!). Another useful function for working with this field is gmtime.
DWORD	PointerToSymbolTable	The file offset of the COFF symbol table, described in section 5.4 of the Microsoft specification. COFF symbol tables are relatively rare in PE files, as newer debug formats have taken over. Prior to Visual Studio .NET, a COFF symbol table could be created by specifying the linker switch /DEBUGTYPE:COFF. COFF symbol tables are almost always found in OBJ files. Set to 0 if no symbol table is present.
DWORD	NumberOfSymbols	Number of symbols in the COFF symbol table, if present. COFF symbols are a fixed size structure, and this field is needed to find the end of the COFF symbols. Immediately following the COFF symbols is a string table used to hold longer symbol names.
WORD	SizeOfOptionalHeader	The size of the optional data that follows the IMAGE_FILE_HEADER. In PE files, this data is the IMAGE_OPTIONAL_HEADER. This size is different depending on whether it's a 32 or 64-bit file. For 32-bit PE files, this field is usually 224. For 64-bit PE32+ files, it's usually 240. However, these sizes are just minimum values, and larger values could appear.
WORD	Characteristics	A set of bit flags indicating attributes of the file. Valid values of these flags are the IMAGE_FILE_xxx values defined in WINNT.H. Some of the more common values include those listed in Figure 4.

Figure 4 IMAGE_FILE_XXX

Value	Description
IMAGE_FILE_RELOCS_STRIPPED	Relocation information stripped from a file.
IMAGE_FILE_EXECUTABLE_IMAGE	The file is executable.
IMAGE_FILE_AGGRESIVE_WS_TRIM	Lets the OS aggressively trim the working set.
IMAGE_FILE_LARGE_ADDRESS_AWARE	The application can handle addresses greater than two gigabytes.
IMAGE_FILE_32BIT_MACHINE	This requires a 32-bit word machine.
IMAGE_FILE_DEBUG_STRIPPED	Debug information is stripped to a .DBG file.

IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP	If the image is on removable media, copy to and run from the swap file.
IMAGE_FILE_NET_RUN_FROM_SWAP	If the image is on a network, copy to and run from the swap file.
IMAGE_FILE_DLL	The file is a DLL.
IMAGE_FILE_UP_SYSTEM_ONLY	The file should only be run on single-processor machines.

Figure 5 IMAGE_OPTIONAL_HEADER

Size	Structure Member	Description
WORD	Magic	A signature WORD, identifying what type of header this is. The two most common values are IMAGE_NT_OPTIONAL_HDR32_MAGIC 0x10b and IMAGE_NT_OPTIONAL_HDR64_MAGIC 0x20b.
BYTE	MajorLinkerVersion	The major version of the linker used to build this executable. For PE files from the Microsoft linker, this version number corresponds to the Visual Studio version number (for example, version 6 for Visual Studio 6.0).
BYTE	MinorLinkerVersion	The minor version of the linker used to build this executable.
DWORD	SizeOfCode	The combined total size of all sections with the IMAGE_SCN_CNT_CODE attribute.
DWORD	SizeOfInitializedData	The combined size of all initialized data sections.
DWORD	SizeOfUninitializedData	The size of all sections with the uninitialized data attributes. This field will often be 0, since the linker can append uninitialized data to the end of regular data sections.
DWORD	AddressOfEntryPoint	The RVA of the first code byte in the file that will be executed. For DLLs, this entrypoint is called during process initialization and shutdown and during thread creations/destructions. In most executables, this address doesn't directly point to main, WinMain, or DllMain. Rather, it points to runtime library code that calls the aforementioned functions. This field can be set to 0 in DLLs, and none of the previous notifications will be received. The linker /NOENTRY switch sets this field to 0.
DWORD	BaseOfCode	The RVA of the first byte of code when loaded in memory.
DWORD	BaseOfData	Theoretically, the RVA of the first byte of data when loaded into memory. However, the values for this field are inconsistent with different versions of the Microsoft linker. This field is not present in 64-bit executables.
DWORD	ImageBase	The preferred load address of this file in memory. The loader attempts to load the PE file at this address if possible (that is, if nothing else currently occupies that memory, it's aligned properly and at a legal address, and so on). If the executable loads at this address, the loader can skip the step of applying base relocations (described in Part 2 of this article). For EXEs, the default ImageBase is 0x400000. For DLLs, it's 0x10000000. The ImageBase can be set at link time with the /BASE switch, or later with the REBASE utility.
DWORD	SectionAlignment	The alignment of sections when loaded into memory. The alignment must be greater or equal to the file alignment field (mentioned next). The default alignment is the page size of the target CPU. For user mode executables to run under Windows 9x or Windows Me, the minimum alignment size is a page (4KB). This field can be set with the linker /ALIGN switch.
DWORD	FileAlignment	The alignment of sections within the PE file. For x86 executables, this value is usually either 0x200 or 0x1000. The default has changed with different versions of the Microsoft linker. This value must be a power of 2, and if the SectionAlignment is less than the CPU's page size, this field must match the SectionAlignment. The linker switch /OPT:WIN98 sets the file alignment on x86 executables to 0x1000, while /OPT:NOWIN98 sets the alignment to 0x200.
WORD	MajorOperatingSystemVersion	The major version number of the required operating system. With the advent of so many versions of Windows, this field has effectively become irrelevant.

WORD	MinorOperatingSystemVersion	The minor version number of the required OS.
WORD	MajorImageVersion	The major version number of this file. Unused by the system and can be 0. It can be set with the linker /VERSION switch.
WORD	MinorImageVersion	The minor version number of this file.
WORD	MajorSubsystemVersion	The major version of the operating subsystem needed for this executable. At one time, it was used to indicate that the newer Windows 95 or Windows NT 4.0 user interface was required, as opposed to older versions of the Windows NT interface. Today, because of the proliferation of the various versions of Windows, this field is effectively unused by the system and is typically set to the value 4. Set with the linker /SUBSYSTEM switch.
WORD	MinorSubsystemVersion	The minor version of the operating subsystem needed for this executable.
DWORD	Win32VersionValue	Another field that never took off. Typically set to 0.
DWORD	SizeOfImage	SizeOfImage contains the RVA that would be assigned to the section following the last section if it existed. This is effectively the amount of memory that the system needs to reserve when loading this file into memory. This field must be a multiple of the section alignment.
DWORD	SizeOfHeaders	The combined size of the MS-DOS header, PE headers, and section table. All of these items will occur before any code or data sections in the PE file. The value of this field is rounded up to a multiple of the file alignment.
DWORD	CheckSum	The checksum of the image. The CheckSumMappedFile API in IMAGEHLP.DLL can calculate this value. Checksums are required for kernel-mode drivers and some system DLLs. Otherwise, this field can be 0. The checksum is placed in the file when the /RELEASE linker switch is used.
WORD	Subsystem	An enum value indicating what subsystem (user interface type) the executable expects. This field is only important for EXEs. Important values include: • IMAGE_SUBSYSTEM_NATIVE // Image doesn't require a subsystem • IMAGE_SUBSYSTEM_WINDOWS_GUI // Use the Windows GUI • IMAGE_SUBSYSTEM_WINDOWS_CUI // Run as a console mode application When run, the OS creates a console window for it, and provides stdin, stdout, and stderr file handles
WORD	DllCharacteristics	Flags indicating characteristics of this DLL. These correspond to the IMAGE_DLLCHARACTERISTICS_xxx fields #defines. Current values are: • IMAGE_DLLCHARACTERISTICS_NO_BIND // Do not bind this image • IMAGE_DLLCHARACTERISTICS_WDM_DRIVER // Driver uses WDM model • IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE // When the terminal server loads // an application that is not Terminal-Services-aware, it also loads a DLL that contains compatibility code
DWORD	SizeOfStackReserve	In EXE files, the maximum size the initial thread in the process can grow to. This is 1MB by default. Not all this memory is committed initially.
DWORD	SizeOfStackCommit	In EXE files, the amount of memory initially committed to the stack. By default, this field is 4KB.
DWORD	SizeOfHeapReserve	In EXE files, the initial reserved size of the default process heap. This is 1MB by default. In current versions of Windows, however, the heap can grow beyond this size without intervention by the user.
DWORD	SizeOfHeapCommit	In EXE files, the size of memory committed to the heap. By default, this is 4KB.
DWORD	LoaderFlags	This is obsolete.
DWORD	NumberOfRvaAndSizes	At the end of the IMAGE_NT_HEADERS structure is an array of IMAGE_DATA_DIRECTORY structures. This field contains the number of entries

		in the array. This field has been 16 since the earliest releases of Windows NT.
IMAGE_	DataDirectory[16]	An array of IMAGE_DATA_DIRECTORY structures. Each structure contains the RVA and size of some important part of the executable (for instance, imports, exports, resources).

Figure 6 The IMAGE_SECTION_HEADER

Size	Field	Description
BYTE	Name[8]	The ASCII name of the section. A section name is not guaranteed to be null-terminated. If you specify a section name longer than eight characters, the linker truncates it to eight characters in the executable. A mechanism exists for allowing longer section names in OBJ files. Section names often start with a period, but this is not a requirement. Section names with a $ in the name get special treatment from the linker. Sections with identical names prior to the $ character are merged. The characters following the $ provide an alphabetic ordering for how the merged sections appear in the final section. There's quite a bit more to the subject of sections with $ in the name and how they're combined, but the details are outside the scope of this article
DWORD	Misc.VirtualSize	Indicates the actual, used size of the section. This field may be larger or smaller than the SizeOfRawData field. If the VirtualSize is larger, the SizeOfRawData field is the size of the initialized data from the executable, and the remaining bytes up to the VirtualSize should be zero-padded. This field is set to 0 in OBJ files.
DWORD	VirtualAddress	In executables, indicates the RVA where the section begins in memory. Should be set to 0 in OBJs.
DWORD	SizeOfRawData	The size (in bytes) of data stored for the section in the executable or OBJ. For executables, this must be a multiple of the file alignment given in the PE header. If set to 0, the section is uninitialized data.
DWORD	PointerToRawData	The file offset where the data for the section begins. For executables, this value must be a multiple of the file alignment given in the PE header.
DWORD	PointerToRelocations	The file offset of relocations for this section. This is only used in OBJs and set to zero for executables. In OBJs, it points to an array of IMAGE_RELOCATION structures if non-zero.
DWORD	PointerToLinenumbers	The file offset for COFF-style line numbers for this section. Points to an array of IMAGE_LINENUMBER structures if non-zero. Only used when COFF line numbers are emitted.
WORD	NumberOfRelocations	The number of relocations pointed to by the PointerToRelocations field. Should be 0 in executables.
WORD	NumberOfLinenumbers	The number of line numbers pointed to by the NumberOfRelocations field. Only used when COFF line numbers are emitted.
DWORD	Characteristics	Flags OR'ed together, indicating the attributes of this section. Many of these flags can be set with the linker's /SECTION option. Common values include those listed in Figure 7.

Figure 7 Flags

Value	Description
IMAGE_SCN_CNT_CODE	The section contains code.
IMAGE_SCN_MEM_EXECUTE	The section is executable.
IMAGE_SCN_CNT_INITIALIZED_DATA	The section contains initialized data.
IMAGE_SCN_CNT_UNINITIALIZED_DATA	The section contains uninitialized data.
IMAGE_SCN_MEM_DISCARDABLE	The section can be discarded from the final executable. Used to hold information for the linker's use, including the .debug$ sections.

IMAGE_SCN_MEM_NOT_PAGED	The section is not pageable, so it should always be physically present in memory. Often used for kernel-mode drivers.
IMAGE_SCN_MEM_SHARED	The physical pages containing this section's data will be shared between all processes that have this executable loaded. Thus, every process will see the exact same values for data in this section. Useful for making global variables shared between all instances of a process. To make a section shared, use the /section:name,S linker switch.
IMAGE_SCN_MEM_READ	The section is readable. Almost always set.
IMAGE_SCN_MEM_WRITE	The section is writeable.
IMAGE_SCN_LNK_INFO	The section contains information for use by the linker. Only exists in OBJs.
IMAGE_SCN_LNK_REMOVE	The section contents will not become part of the image. This only appears in OBJ files.
IMAGE_SCN_LNK_COMDAT	The section contents is communal data (comdat). Communal data is data (or code) that can be defined in multiple OBJs. The linker will select one copy to include in the executable. Comdats are vital for support of C++ template functions and function-level linking. Comdat sections only appear in OBJ files.
IMAGE_SCN_ALIGN_XBYTES	The alignment of this section's data in the resultant executable. There are a variety of these values (_4BYTES, _8BYTES, _16BYTES, and so on). The default, if not specified, is 16 bytes. These flags will only be set in OBJ files.

Figure 8 Section Names

Name	Description
.text	The default code section.
.data	The default read/write data section. Global variables typically go here.
.rdata	The default read-only data section. String literals and C++/COM vtables are examples of items put into .rdata.
.idata	The imports table. It has become common practice (either explicitly, or via linker default behavior) to merge the .idata section into another section, typically .rdata. By default, the linker only merges the .idata section into another section when creating a release mode executable.
.edata	The exports table. When creating an executable that exports APIs or data, the linker creates an .EXP file. The .EXP file contains an .edata section that's added into the final executable. Like the .idata section, the .edata section is often found merged into the .text or .rdata sections.
.rsrc	The resources. This section is read-only. However, it should not be named anything other than .rsrc, and should not be merged into other sections.
.bss	Uninitialized data. Rarely found in executables created with recent linkers. Instead, the VirtualSize of the executable's .data section is expanded to make enough room for uninitialized data.
.crt	Data added for supporting the C++ runtime (CRT). A good example is the function pointers that are used to call the constructors and destructors of static C++ objects. See the January 2001 Under The Hood column for details on this.
.tls	Data for supporting thread local storage variables declared with __declspec(thread). This includes the initial value of the data, as well as additional variables needed by the runtime.
.reloc	The base relocations in an executable. Base relocations are generally only needed for DLLs and not EXEs. In release mode, the linker doesn't emit base relocations for EXE files. Relocations can be removed when linking with the /FIXED switch.
.sdata	"Short" read/write data that can be addressed relative to the global pointer. Used for the IA-64 and other architectures that use a global pointer register. Regular-sized global variables on the IA-64 will go in this section.
.srdata	"Short" read-only data that can be addressed relative to the global pointer. Used on the IA-64 and other architectures that use a global pointer register.
.pdata	The exception table. Contains an array of IMAGE_RUNTIME_FUNCTION_ENTRY structures, which are CPU-specific. Pointed to by the IMAGE_DIRECTORY_ENTRY_EXCEPTION slot in the DataDirectory. Used for architectures with table-based exception handling, such as the IA-64. The only architecture that doesn't use table-based

	exception handling is the x86.
.debug$S	Codeview format symbols in the OBJ file. This is a stream of variable-length CodeView format symbol records.
.debug$T	Codeview format type records in the OBJ file. This is a stream of variable-length CodeView format type records.
.debug$P	Found in the OBJ file when using precompiled headers.
.drectve	Contains linker directives and is only found in OBJs. Directives are ASCII strings that could be passed on the linker command line. For instance: -defaultlib:LIBC Directives are separated by a space character.
.didat	Delayload import data. Found in executables built in nonrelease mode. In release mode, the delayload data is merged into another section.

Figure 9 IMAGE_EXPORT_DIRECTORY Structure Members

Size	Member	Description
DWORD	Characteristics	Flags for the exports. Currently, none are defined.
DWORD	TimeDateStamp	The time/date that the exports were created. This field has the same definition as the IMAGE_NT_HEADERS.FileHeader. TimeDateStamp (number of seconds since 1/1/1970 GMT).
WORD	MajorVersion	The major version number of the exports. Not used, and set to 0.
WORD	MinorVersion	The minor version number of the exports. Not used, and set to 0.
DWORD	Name	A relative virtual address (RVA) to an ASCII string with the DLL name associated with these exports (for example, KERNEL32.DLL).
DWORD	Base	This field contains the starting ordinal value to be used for this executable's exports. Normally, this value is 1, but it's not required to be so. When looking up an export by ordinal, the value of this field is subtracted from the ordinal, with the result used as a zero-based index into the Export Address Table (EAT).
DWORD	NumberOfFunctions	The number of entries in the EAT. Note that some entries may be 0, indicating that no code/data is exported with that ordinal value.
DWORD	NumberOfNames	The number of entries in the Export Names Table (ENT). This value will always be less than or equal to the NumberOf-Functions field. It will be less when there are symbols exported by ordinal only. It can also be less if there are numeric gaps in the assigned ordinals. This field is also the size of the export ordinal table (below).
DWORD	AddressOfFunctions	The RVA of the EAT. The EAT is an array of RVAs. Each nonzero RVA in the array corresponds to an exported symbol.
DWORD	AddressOfNames	The RVA of the ENT. The ENT is an array of RVAs to ASCII strings. Each ASCII string corresponds to a symbol exported by name. This table is sorted so that the ASCII strings are in order. This allows the loader to do a binary search when looking for an exported symbol. The sorting of the names is binary (like the C++ RTL strcmp function provides), rather than a locale-specific alphabetic ordering.
DWORD	AddressOfNameOrdinals	The RVA of the export ordinal table. This table is an array of WORDs. This table maps an array index from the ENT into the corresponding export address table entry.

Figure 10 KERNEL32 Exports

```
exports table:
  Name:           KERNEL32.dll
  Characteristics: 00000000
  TimeDateStamp:   3B7DDFD8 -> Fri Aug 17 23:24:08 2001
  Version:         0.00
  Ordinal base:    00000001
```

```
# of functions:  000003A0
# of Names:      000003A0

Entry Pt  Ordn  Name
00012ADA     1  ActivateActCtx
000082C2     2  AddAtomA
•••remainder of exports omitted
```

Figure 11 **IMAGE_IMPORT_DESCRIPTOR Structure**

Size	Member	Description
DWORD	OriginalFirstThunk	This field is badly named. It contains the RVA of the Import Name Table (INT). This is an array of IMAGE_THUNK_DATA structures. This field is set to 0 to indicate the end of the array of IMAGE_IMPORT_DESCRIPTORs.
DWORD	TimeDateStamp	This is 0 if this executable is not bound against the imported DLL. When binding in the old style (see the section on Binding), this field contains the time/date stamp (number of seconds since 1/1/1970 GMT) when the binding occurred. When binding in the new style, this field is set to -1.
DWORD	ForwarderChain	This is the Index of the first forwarded API. Set to -1 if no forwarders. Only used for old-style binding, which could not handle forwarded APIs efficiently.
DWORD	Name	The RVA of the ASCII string with the name of the imported DLL.
DWORD	FirstThunk	Contains the RVA of the Import Address Table (IAT). This is array of IMAGE_THUNK_DATA structures.

Figure 12 **ImgDelayDescr Structure**

Size	Member	Description
DWORD	grAttrs	The attributes for this structure. Currently, the only flag defined is dlattrRva (1), indicating that the address fields in the structure should be treated as RVAs, rather than virtual addresses.
RVA	rvaDLLName	An RVA to a string with the name of the imported DLL. This string is passed to LoadLibrary.
RVA	rvaHmod	An RVA to an HMODULE-sized memory location. When the Delayloaded DLL is brought into memory, its HMODULE is stored at this location.
RVA	rvaIAT	An RVA to the Import Address Table for this DLL. This is the same format as a regular IAT.
RVA	rvaINT	An RVA to the Import Name Table for this DLL. This is the same format as a regular INT.
RVA	rvaBoundIAT	An RVA of the optional bound IAT. An RVA to a bound copy of an Import Address Table for this DLL. This is the same format as a regular IAT. Currently, this copy of the IAT is not actually bound, but this feature may be added in future versions of the BIND program.
RVA	rvaUnloadIAT	An RVA of the optional copy of the original IAT. An RVA to an unbound copy of an Import Address Table for this DLL. This is the same format as a regular IAT. Currently always set to 0.
DWORD	dwTimeStamp	The date/time stamp of the delayload imported DLL. Normally set to 0.

Figure 13 **Resources from ADVAPI32.DLL**

```
Resources (RVA: 6B000)
  • ResDir (0) Entries:03 (Named:01, ID:02) TimeDate:00000000
      ○ ResDir (MOFDATA) Entries:01 (Named:01, ID:00) TimeDate:00000000
          ▪ ResDir (MOFRESOURCENAME) Entries:01 (Named:00, ID:01) TimeDate:00000000
              • ID: 00000409  DataEntryOffs: 00000128
              • DataRVA: 6B6F0  DataSize: 190F5  CodePage: 0
```

- *ResDir (STRING) Entries:01 (Named:00, ID:01) TimeDate:00000000*
  - *ResDir (C36) Entries:01 (Named:00, ID:01) TimeDate:00000000*
    - *ID: 00000409 DataEntryOffs: 00000138*
    - *DataRVA: 6B1B0 DataSize: 0053C CodePage: 0*
- *ResDir (RCDATA) Entries:01 (Named:00, ID:01) TimeDate:00000000*
  - *ResDir (66) Entries:01 (Named:00, ID:01) TimeDate:00000000*
    - *ID: 00000409 DataEntryOffs: 00000148*
    - *DataRVA: 85908 DataSize: 0005C CodePage: 0*

Figure 14 Fields of IMAGE_DEBUG_DIRECTORY

Size	Member	Description
DWORD	Characteristics	Unused and set to 0.
DWORD	TimeDateStamp	The time/date stamp of this debug information (number of seconds since 1/1/1970, GMT).
WORD	MajorVersion	The major version of this debug information. Unused.
WORD	MinorVersion	The minor version of this debug information. Unused.
DWORD	Type	The type of the debug information. The following types are the most commonly encountered: IMAGE_DEBUG_TYPE_COFF IMAGE_DEBUG_TYPE_CODEVIEW // Including PDB files IMAGE_DEBUG_TYPE_FPO // Frame pointer omission IMAGE_DEBUG_TYPE_MISC // IMAGE_DEBUG_MISC IMAGE_DEBUG_TYPE_OMAP_TO_SRC IMAGE_DEBUG_TYPE_OMAP_FROM_SRC IMAGE_DEBUG_TYPE_BORLAND // Borland format
DWORD	SizeOfData	The size of the debug data in this file. Doesn't count the size of external debug files such as .PDBs.
DWORD	AddressOfRawData	The RVA of the debug data, when mapped into memory. Set to 0 if the debug data isn't mapped in.
DWORD	PointerToRawData	The file offset of the debug data (not an RVA).

Figure 15 IMAGE_COR20_HEADER Structure

Type	Member	Description
DWORD	cb	Size of the header in bytes.
WORD	MajorRuntimeVersion	The minimum version of the runtime required to run this program. For the first release of .NET, this value is 2.
WORD	MinorRuntimeVersion	The minor portion of the version. Currently 0.
IMAGE_DATA_DIRECTORY	MetaData	The RVA to the metadata tables.

DWORD	Flags	Flag values containing attributes for this image. These values are currently defined as:
		COMIMAGE_FLAGS_ILONLY // Image contains only IL code that
		// is not required to run on a specific CPU.
		COMIMAGE_FLAGS_32BITREQUIRED // Only runs in 32-bit processes.
		COMIMAGE_FLAGS_IL_LIBRARY
		STRONGNAMESIGNED // Image is signed with hash data
		COMIMAGE_FLAGS_TRACKDEBUGDATA // Causes the JIT/runtime to
		// keep debug information

		// around for methods.
DWORD	EntryPointToken	Token for the MethodDef of the entry point for the image. The .NET runtime calls this method to begin managed execution in the file.
IMAGE_DATA_DIRECTORY	Resources	The RVA and size of the .NET resources.
IMAGE_DATA_DIRECTORY	StrongNameSignature	The RVA of the strong name hash data.
IMAGE_DATA_DIRECTORY	CodeManagerTable	The RVA of the code manager table. A code manager contains the code required to obtain the state of a running program (such as tracing the stack and track GC references).
IMAGE_DATA_DIRECTORY	VTableFixups	The RVA of an array of function pointers that need fixups. This is for support of unmanaged C++ vtables.
IMAGE_DATA_DIRECTORY	ExportAddressTableJumps	The RVA to an array of RVAs where export JMP thunks are written. These thunks allow managed methods to be exported so that unmanaged code can call them.
IMAGE_DATA_DIRECTORY	ManagedNativeHeader	For internal use of the .NET runtime in memory. Set to 0 in the executable.

Figure 16 IMAGE_TLS_DIRECTORY Structure

Size	Member	Description
DWORD	StartAddressOfRawData	The beginning address of a range of memory used to initialize a new thread's TLS data in memory.
DWORD	EndAddressOfRawData	The ending address of the range of memory used to initialize a new thread's TLS data in memory.
DWORD	AddressOfIndex	When the executable is brought into memory and a .tls section is present, the loader allocates a TLS handle via TlsAlloc. It stores the handle at the address given by this field. The runtime library uses this index to locate the thread local data.
DWORD	AddressOfCallBacks	Address of an array of PIMAGE_TLS_CALLBACK function pointers. When a thread is created or destroyed, each function in the list is called. The end of the list is indicated by a pointer-sized variable set to 0. In normal Visual C++ executables, this list is empty.
DWORD	SizeOfZeroFill	The size in bytes of the initialization data, beyond the initialized data delimited by the StartAddressOfRawData and EndAddressOfRawData fields. All per-thread data after this range is initialized to 0.
DWORD	Characteristics	Reserved. Currently set to 0.

Figure 17 Command-line Options

/A	Include everything in dump
/B	Show base relocations
/H	Include hex dump of sections
/I	Include Import Address Table thunk addresses
/L	Include line number information
/P	Include PDATA (runtime functions)
/R	Include detailed resources (stringtables and dialogs)
/S	Show symbol table

Figure 18 - RunInit.cpp

```
// RunInit.cpp
```

```
//=============================================================================
// Matt Pietrek, September 1999 Microsoft Systems Journal
// Pseudocode for LdrpRunInitializeRoutines in NTDLL.DLL (Windows NT 4.0, SP3)
// bImplicitLoad parameter is nonzero when LdrpRunInitializeRoutines is
// called for the first time in a process (that is, when the implicitly
// linked modules are initialized.
// On subsequent invocations (caused by calls to LoadLibrary), bImplicitLoad
// is 0.
//=============================================================================

#include <ntexapi.h>    // For HardError defines near the end

// Global symbols (name is accurate, and comes from NTDLL.DBG)
//  _NtdllBaseTag
//  _ShowSnaps
//  _SaveSp
//  _CurSp
//  _LdrpInLdrInit
//  _LdrpFatalHardErrorCount
//  _LdrpImageHasTls

NTSTATUS
LdrpRunInitializeRoutines( DWORD bImplicitLoad )
{
   // Get the number of modules that *might* need to be initialized.  Some
   // of the modules may already have been initialized.

   unsigned nRoutinesToRun = _LdrpClearLoadInProgress();

   if ( nRoutinesToRun )
   {
      // If there are any init routines, allocate memory for an array
      // containing information about each module
      pInitNodeArray = _RtlAllocateHeap(GetProcessHeap(),
                                 _NtdllBaseTag + 0x60000,
                                nRoutinesToRun * 4 );

      if ( 0 == pInitNodeArray )    // Make sure allocation worked
         return STATUS_NO_MEMORY;
   }
   else
      pInitNodeArray = 0;

   // The Process Environment Block (Peb), keeps a pointer to the linked
   // list of modules that have just been loaded.  Get a pointer to this info
   //
   pCurrNode = *(pCurrentPeb->ModuleLoaderInfoHead);
   ModuleLoaderInfoHead = pCurrentPeb->ModuleLoaderInfoHead;

   if ( _ShowSnaps )
   {
      _DbgPrint( "LDR: Real INIT LIST\n" );
   }
```

```
nModulesInitedSoFar = 0;

if ( pCurrNode != ModuleLoaderInfoHead )
{
   // Iterate through the linked list
   //
   while ( pCurrNode != ModuleLoaderInfoHead )
   {
      ModuleLoaderInfo  pModuleLoaderInfo;

      // Apparently the next node pointer is 0x10 bytes inside
      // the ModuleLoaderInfo structure
      //
      pModuleLoaderInfo = &NextNode - 0x10;

      // This doesn't seem to do anything...
      localVar3C = pModuleLoaderInfo;

      // Determine if the module has already been initialized.  If so,
      // skip over it.
      //
      // X_LOADER_SAW_MODULE = 0x40
      if ( !(pModuleLoaderInfo->Flags35 & X_LOADER_SAW_MODULE) )
      {
         // This module hasn't previously been initialized.  Check
         // to see if it has an EntryPoint to call.
         //
         if ( pModuleLoaderInfo->EntryPoint )
         {
            // This previously uninitialized module has an entry
            // point.  Add it to the array of modules that will
            // be called to initialize later in this routine.
            //
            pInitNodeArray[nModulesInitedSoFar] =pModuleLoaderInfo;

            // If ShowSnaps is nonzero, spit out the module path
            // and the entry point address for the module. For example:
            // C:\WINNT\system32\KERNEL32.dll init routine 77f01000
            if ( _ShowSnaps )
            {
               _DbgPrint(  "%wZ init routine %x\n",
                        &pModuleLoaderInfo->24,
                        pModuleLoaderInfo->EntryPoint );
            }

            nModulesInitedSoFar++;
         }
      }

      // Set the X_LOADER_SAW_MODULE flag for this module.  Note that
      // the module hasn't actually been initialized.  Rather, it will
      // be by the time this routine returns.
      pModuleLoaderInfo->Flags35 &= X_LOADER_SAW_MODULE;
```

```
        // Advance to the next node in the module list
        pCurrNode = pCurrNode->pNext
    }
}
else
{
    pModuleLoaderInfo = localVar3C;     // May not be initialized???
}

if ( 0 == pInitNodeArray )
    return STATUS_SUCCESS;

// At this point, pInitNodeArray contains an array of pointers to
// modules that haven't yet seen the DLL_PROCESS_ATTACH notification.
// It's now time to start calling the initialization routines.
//
try     // Wrap all this in a try block, in case the init routine faults
{
    nModulesInitedSoFar = 0;  // Start at array element 0

    // Begin iterating through the module array
    //
    while ( nModulesInitedSoFar < nRoutinesToRun )
    {
        // Get a pointer to the module's info out of the array
        pModuleLoaderInfo = pInitNodeArray[ nModulesInitedSoFar ];

        // This doesn't seem to do anything...
        localVar3C = pModuleLoaderInfo;

        nModulesInitedSoFar++;

        // Store init routine address in a local variable
        pfnInitRoutine = pModuleLoaderInfo->EntryPoint;

        fBreakOnDllLoad = 0;    // Default is to not break on load

        // If this process is a debuggee, check to see if the loader
        // should break into a debugger before calling the initialization.
        //
        // DebuggerPresent (offset 2 in PEB) is what IsDebuggerPresent()
        // returns. IsDebuggerPresent is a Windows NT-only API.
        if ( pCurrentPeb->DebuggerPresent || pCurrentPeb->1 )
        {
            LONG retCode;

            // Query the "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\
            // Windows NT\CurrentVersion\Image File Execution Options"
            // registry key.  If a a subkey entry with the name of
            // the executable exists, check for the BreakOnDllLoad value.
                retCode = _LdrQueryImageFileExecutionOptions(
                pModuleLoaderInfo->pwszDllName, "BreakOnDllLoad",
                REG_DWORD, &fBreakOnDllLoad,
```

```
            sizeof(DWORD), 0 );

    // If reg value not found (usually the case), then don't
    // break on this DLL init
    if ( retCode <= STATUS_SUCCESS )
        fBreakOnDllLoad = 0;
}

if ( fBreakOnDllLoad )
{
    if ( _ShowSnaps )
    {
        // Inform the debug output stream of the module name
        // and the init routine address before actually breaking
        // into the debugger

        _DbgPrint(  "LDR: %wZ loaded.",
                &pModuleLoaderInfo->pModuleLoaderInfo );

        _DbgPrint(  "- About to call init routine at %lx\n",
                pfnInitRoutine )
    }

    // Break into the debugger
    _DbgBreakPoint();   // An INT 3, followed by a RET
}
else if ( _ShowSnaps && pfnInitRoutine )
{
    // Inform the debug output stream of the module name
    // and the init routine address before calling it
    _DbgPrint(  "LDR: %wZ loaded.",
            pModuleLoaderInfo->pModuleLoaderInfo );

    _DbgPrint("- Calling init routine at %lx\n", pfnInitRoutine);
}

if ( pfnInitRoutine )
{
    // Set flag indicating that the DLL_PROCESS_ATTACH notification
    // has been sent to this DLL
    // (Shouldn't this come *after* the actual call?)
    // X_LOADER_CALLED_PROCESS_ATTACH = 0x8
    pModuleLoaderInfo->Flags36 |= X_LOADER_CALLED_PROCESS_ATTACH;

    // If there's Thread Local Storage (TLS) for this module,
    // call the TLS init functions.  *** NOTE *** This only
    // occurs during the first time this code is called (when
    // implicitly loaded DLLs are initialized).  Dynamically
    // loaded DLLs shouldn't use TLS declared vars, as per the
    // SDK documentation
    if ( pModuleLoaderInfo->bHasTLS && bImplicitLoad )
    {
        _LdrpCallTlsInitializers(   pModuleLoaderInfo->hModDLL,
                                    DLL_PROCESS_ATTACH );
```

```
}

hModDLL = pModuleLoaderInfo->hModDLL

MOV     ESI,ESP // Save off the ESP register into ESI

MOV     EDI,DWORD PTR [pfnInitRoutine]
        // Load EDI with module's entry point

// In C++ code, the following ASM would look like:
// initRetValue =
// pfnInitRoutine(hInstDLL,DLL_PROCESS_ATTACH,bImplicitLoad);

PUSH    DWORD PTR [bImplicitLoad]

PUSH    DLL_PROCESS_ATTACH

PUSH    DWORD PTR [hModDLL]

CALL    EDI     // Call the init routine.  Excellent point
              // to set a breakpoint.  Stepping into this
              // call will take you to the DLL's entry point

MOV     BYTE PTR [initRetValue],AL  // Save the return value
                            // from the entry point

MOV     DWORD PTR [_SaveSp],ESI // Save stack values after the
MOV     DWORD PTR [_CurSp],ESP  // entry point code returns

MOV     ESP,ESI     // Restore ESP to value before the call

// Check ESP (stack pointer) after the call.  If it's not
// the same as the ESP value before the call, the DLL's
// init routine didn't clean up the stack properly.  For
// example, it's entry routine may have been defined
// improperly.  Although this rarely happens, if it does,
// let the user know and ask if they want to continue.
if ( _CurSP != _SavSP )
{
   hardErrorParam = pModuleLoaderInfo->FullDllPath;

   hardErrorRetCode = _NtRaiseHardError(
         STATUS_BAD_DLL_ENTRYPOINT | 0x10000000,
         1,  // Number of parameters
         1,  // UnicodeStringParametersMask,
         &hardErrorParam,
         OptionYesNo,    // Let user decide
         &hardErrorResponse );

   if ( _LdrpInLdrInit )
      _LdrpFatalHardErrorCount++;

   if (    (hardErrorRetCode >= STATUS_SUCCESS)
      &&  (ResponseYes == hardErrorResponse) )
```

```
            {
               return STATUS_DLL_INIT_FAILED;
            }
         }

         // If the DLL's entry point returned 0 (failure), tell the user
         if ( 0 == initRetValue )
         {
            DWORD hardErrorParam2;
            DWORD hardErrorResponse2;

            hardErrorParam2 = pModuleLoaderInfo->FullDllPath;

            _NtRaiseHardError(  STATUS_DLL_INIT_FAILED,
                        1,  // Number of parameters
                        1,  // UnicodeStringParametersMask
                        &hardErrorParam2,
                        OptionOk,   // OK is only response
                        &hardErrorResponse2 );

            if ( _LdrpInLdrInit )
               _LdrpFatalHardErrorCount++;

            return STATUS_DLL_INIT_FAILED;
         }
      }
   }

   // If the EXE itself has TLS declared vars, call the init routines.
   // See the comment for the previous call to _LdrpCallTlsInitializers
   // for more details.
   //
   if ( _LdrpImageHasTls && bImplicitLoad )
   {
      _LdrpCallTlsInitializers(   pCurrentPeb->ProcessImageBase,
                        DLL_PROCESS_ATTACH );
   }
}
__finally
{
   // Before exiting the routine, make sure that the memory allocated
   // at the beginning is freed
   _RtlFreeHeap( GetProcessHeap(), 0, pInitNodeArray );
}

return STATUS_SUCCESS;
}
```

Figure 19 - ShowSnaps

```
// ShowSnaps Output from CALC.EXE

//=================================================================
 ## Matt's comments denoted by ##'s
```

```
Loaded 'C:\WINNT\system32\CALC.EXE', no matching symbolic information found.
Loaded symbols for 'C:\WINNT\system32\ntdll.dll'
LDR: PID: 0x3a started - '"C:\WINNT\system32\CALC.EXE"'
LDR: NEW PROCESS
    Image Path: C:\WINNT\system32\CALC.EXE (CALC.EXE)
    Current Directory: C:\WINNT\system32
    Search Path: C:\WINNT\system32;.;C:\WINNT\System32;C:\WINNT\system;...
LDR: SHELL32.dll used by CALC.EXE
Loaded 'C:\WINNT\system32\SHELL32.DLL', no matching symbolic information found.
LDR: ntdll.dll used by SHELL32.dll
LDR: Snapping imports for SHELL32.dll from ntdll.dll
LDR: KERNEL32.dll used by SHELL32.dll
Loaded symbols for 'C:\WINNT\system32\KERNEL32.DLL'
LDR: ntdll.dll used by KERNEL32.dll
LDR: Snapping imports for KERNEL32.dll from ntdll.dll
LDR: Snapping imports for SHELL32.dll from KERNEL32.dll
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlEnterCriticalSection
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlDeleteCriticalSection

//.... additional output omitted

LDR: GDI32.dll used by SHELL32.dll
Loaded symbols for 'C:\WINNT\system32\GDI32.DLL'
LDR: ntdll.dll used by GDI32.dll
LDR: Snapping imports for GDI32.dll from ntdll.dll
LDR: KERNEL32.dll used by GDI32.dll
LDR: Snapping imports for GDI32.dll from KERNEL32.dll
LDR: USER32.dll used by GDI32.dll
Loaded symbols for 'C:\WINNT\system32\USER32.DLL'
LDR: ntdll.dll used by USER32.dll
LDR: Snapping imports for USER32.dll from ntdll.dll
LDR: KERNEL32.dll used by USER32.dll
LDR: Snapping imports for USER32.dll from KERNEL32.dll
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlSizeHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlReAllocateHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlFreeHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlAllocateHeap

//.... additional output omitted

## Note loader looking for and verifying "bound" DLL imports in COMCTL32
Loaded 'C:\WINNT\system32\COMCTL32.DLL', no matching symbolic information found.
LDR: COMCTL32.dll bound to ntdll.dll
LDR: COMCTL32.dll has correct binding to ntdll.dll
LDR: COMCTL32.dll bound to GDI32.dll
LDR: COMCTL32.dll has correct binding to GDI32.dll
LDR: COMCTL32.dll bound to KERNEL32.dll
LDR: COMCTL32.dll has correct binding to KERNEL32.dll
```

```
LDR: COMCTL32.dll bound to ntdll.dll via forwarder(s) from KERNEL32.dll
LDR: COMCTL32.dll has correct binding to ntdll.dll
LDR: COMCTL32.dll bound to USER32.dll
LDR: COMCTL32.dll has correct binding to USER32.dll
LDR: COMCTL32.dll bound to ADVAPI32.dll
LDR: COMCTL32.dll has correct binding to ADVAPI32.dll

//.... additional output omitted

LDR: Refcount   COMCTL32.dll (1)
LDR: Refcount   GDI32.dll (3)
LDR: Refcount   KERNEL32.dll (6)
LDR: Refcount   USER32.dll (4)
LDR: Refcount   ADVAPI32.dll (5)
LDR: Refcount   KERNEL32.dll (7)
LDR: Refcount   GDI32.dll (4)
LDR: Refcount   USER32.dll (5)

## List of implicit link DLLs to be init'ed.
LDR: Real INIT LIST
    C:\WINNT\system32\KERNEL32.dll init routine 77f01000
    C:\WINNT\system32\RPCRT4.dll init routine 77e1b6d5
    C:\WINNT\system32\ADVAPI32.dll init routine 77dc1000
    C:\WINNT\system32\USER32.dll init routine 77e78037
    C:\WINNT\system32\COMCTL32.dll init routine 71031a18
    C:\WINNT\system32\SHELL32.dll init routine 77c41094

## Beginning of actual calls to implicitly linked DLL init routines
LDR: KERNEL32.dll loaded. - Calling init routine at 77f01000
LDR: RPCRT4.dll loaded. - Calling init routine at 77e1b6d5
LDR: ADVAPI32.dll loaded. - Calling init routine at 77dc1000
LDR: USER32.dll loaded. - Calling init routine at 77e78037

## USER32 does AppInit_DLLs thing, so static inits temporarily interrupted
## In this case, "globaldll.dll" is LoadLibrary'ed from USER32 init code
LDR: LdrLoadDll, loading c:\temp\globaldll.dll from C:\WINNT\system32;.;
LDR: Loading (DYNAMIC) c:\temp\globaldll.dll
Loaded 'C:\TEMP\GlobalDLL.dll', no matching symbolic information found.
LDR: KERNEL32.dll used by globaldll.dll

//.... additional output omitted

LDR: Real INIT LIST
    c:\temp\globaldll.dll init routine 10001310
LDR: globaldll.dll loaded. - Calling init routine at 10001310

## Now back to calling the inits of the statically linked DLLs
LDR: COMCTL32.dll loaded. - Calling init routine at 71031a18
LDR: LdrGetDllHandle, searching for USER32.dll from
LDR: LdrGetProcedureAddress by NAME - GetSystemMetrics
LDR: LdrGetProcedureAddress by NAME - MonitorFromWindow
LDR: SHELL32.dll loaded. - Calling init routine at 77c41094
//.... additional output omitted
```

Figure 20 - TLSInit.cpp

```
// TLSInit.cpp
//=============================================================

void _LdrpCallTlsInitializers( HMODULE hModule, DWORD fdwReason )
{
   PIMAGE_TLS_DIRECTORY pTlsDir;
   DWORD size

   // Look up the TLS directory in the IMAGE_OPTIONAL_HEADER.DataDirectory
   pTlsDir = _RtlImageDirectoryEntryToData(hModule,
                                1,
                                IMAGE_DIRECTORY_ENTRY_TLS,
                                &size );

   __try   // Protect all this code with a try/catch block
   {
      if ( pTlsDir->AddressOfCallbacks )
      {
         if ( _ShowSnaps )   // diagnostic output
         {
            _DbgPrint(  "LDR: Tls Callbacks Found. "
                     "Imagebase %lx Tls %lx CallBacks %lx\n",
                     hModule, TlsDir, pTlsDir->AddressOfCallbacks );
         }

         // Get pointer to beginning of array of TLS callback addresses
         PVOID * pCallbacks = pTlsDir->AddressOfCallbacks;

         while ( *pCallbacks )   // Iterate through each array entry
         {
            PIMAGE_TLS_CALLBACK pTlsCallback = *pCallbacks;
            pCallbacks++;

            if ( _ShowSnaps )   // More diagnostic output
            {
               _DbgPrint(  "LDR: Calling Tls Callback "
                        "Imagebase %lx Function %lx\n",
                        hModule, pTlsCallback );
            }

            // Make the actual call
            pTlsCallback( hModule, fdwReason, 0 );
         }
      }
   }
   __except( EXCEPTION_EXECUTE_HANDLER )
   {
   }
}
```

Figure 21 Loader APIs

API	NTDLL APIs
LoadResource	LdrAccessResource LdrAlternateResourcesEnabled
DisableThreadLibraryCalls	LdrDisableThreadCalloutsForDll LdrEnumResources LdrFindAppCompatVariableInfo LdrFindEntryForAddress
EnumResourceTypesW	LdrFindResourceDirectory_U
FindResourceExA	LdrFindResource_U LdrFlushAlternateResourceModules LdrGetAlternateResourceModuleHandle
GetModuleHandleForUnicodeString	LdrGetDllHandle
GetProcAddress	LdrGetProcedureAddress LdrInitializeThunk
LoadLibraryEx (LOAD_LIBRARY_AS_DATAFILE)	LdrLoadAlternateResourceModule
LoadLibrary	LdrLoadDll LdrProcessRelocationBlock LdrQueryApplicationCompatibilityGoo LdrQueryImageFileExecutionOptions LdrQueryProcessModuleInformation LdrRelocateImage
ExitProcess	LdrShutdownProcess
ExitThread	LdrShutdownThread LdrUnloadAlternateResourceModule
FreeLibrary	LdrUnloadDll LdrVerifyImageMatchesChecksum LdrVerifyMappedImageMatchesChecksum

Figure 22 Private Loader APIs

```
LdrpAccessResourceData
LdrpAllocateDataTableEntry
LdrpAllocateTls
LdrpCallInitRoutine
LdrpCallTlsInitializers
LdrpCheckForKnownDll
LdrpCheckForLoadedDll
LdrpCheckForLoadedDllHandle
LdrpClearLoadInProgress
LdrpCompareResourceNames_U
LdrpCreateDllSection
LdrpDefineDllTag
LdrpDllTagProcedures
LdrpDphDetectSnapRoutines
LdrpDphInitializeTargetDll
LdrpDphSnapImports
LdrpFetchAddressOfEntryPoint
LdrpForkProcess
LdrpFreeTls
LdrpGetProcedureAddress
LdrpInitializationFailure
LdrpInitialize
LdrpInitializeProcess
LdrpInitializeThread
LdrpInitializeTls
LdrpInsertMemoryTableEntry
LdrpLoadDll
LdrpLoadImportModule
LdrpMapDll
LdrpNameToOrdinal
LdrpRelocateStartContext
LdrpResolveDllName
LdrpRunInitializeRoutines
LdrpSearchResourceSection_U
LdrpSetAlternateResourceModuleHandle
LdrpSetProtection
LdrpSnapIAT
LdrpSnapThunk
LdrpTagAllocateHeap0...LdrpTagAllocateHeap63
LdrpTagAllocateHeap
LdrpUpdateLoadCount
LdrpValidateImageForMp
LdrpWalkImportDescriptor
```

Figure 23 **Internal Loader Routines**

```
#1 - LoadLibraryExW 3rd parameter is 0:
LdrLoadDll (0x77f889a9)
   LdrpLoadDll 0x77f887e0
      LdrpCheckForLoadedDll (0x77f87122)
      LdrpMapDll (0x77f8bc77)
         LdrpCheckForKnownDll (0x77f8c62b)
         LdrpResolveDllName (0x77f8c3df)
         LdrpCreateDllSection (0x77f8c355)
         LdrpAllocateDataTableEntry (0x77f8be69)
         LdrpFetchAddressOfEntryPoint (0x77f8bf23)
         LdrpInsertMemoryTableEntry (0x77f8bebb)
      LdrpWalkImportDescriptor (0x77f8be15)
         LdrpLoadImportModule (0x77f8bfd1)
```

```
            *LdrpCheckForLoadedDll
         LdrpSnapIAT (0x77f8c047)
            LdrpSnapThunk (0x77f87bd1)
               LdrpNameToOrdinal (0x77f87cf0)
               **LdrpLoadDll
               LdrpGetProcedureAddress (0x77f87a20)
                  LdrpCheckforLoadedDllHandle (0x77f870cc)
                  **LdrpSnapThunk
      LdrpUpdateLoadCount (0x77f88afa)
         *LdrpCheckForLoadedDll
         **LdrpUpdateLoadCount
      LdrpRunInitializeRoutines (0x77f8bcb8)
      LdrpClearLoadInProgress (0x77f88c12)

#2 - LoadLibraryExW 3rd parameter is 0 and DLL has been bound:
LdrLoadDll (0x77f889a9)
   LdrpLoadDll 0x77f887e0
      LdrpCheckForLoadedDll (0x77f87122)
      LdrpMapDll (0x77f8bc77)
         LdrpCheckForKnownDll (0x77f8c62b)
         LdrpResolveDllName (0x77f8c3df)
         LdrpCreateDllSection (0x77f8c355)
         LdrpAllocateDataTableEntry (0x77f8be69)
         LdrpFetchAddressOfEntryPoint (0x77f8bf23)
         LdrpInsertMemoryTableEntry (0x77f8bebb)
      LdrpWalkImportDescriptor (0x77f8be15)
         LdrpLoadImportModule (0x77f8bfd1)
            *LdrpCheckForLoadedDll
      LdrpUpdateLoadCount (0x77f88afa)
         *LdrpCheckForLoadedDll
         **LdrpUpdateLoadCount
      LdrpRunInitializeRoutines (0x77f8bcb8)
      LdrpClearLoadInProgress (0x77f88c12)

#3 - LoadLibraryExW 3rd Parameter is DONT_RESOLVE_DLL_REFERENCES:
LdrLoadDll (0x77f889a9)
   LdrpLoadDll 0x77f887e0
      LdrpCheckForLoadedDll (0x77f87122)
      LdrpMapDll (0x77f8bc77)
         LdrpCheckForKnownDll (0x77f8c62b)
         LdrpResolveDllName (0x77f8c3df)
         LdrpCreateDllSection (0x77f8c355)
         LdrpAllocateDataTableEntry (0x77f8be69)
         LdrpFetchAddressOfEntryPoint (0x77f8bf23)
         LdrpInsertMemoryTableEntry (0x77f8bebb)

#4 - LoadLibraryExW 3rd Parameter is LOAD_WITH_ALTERED_SEARCH_PATH:
LdrLoadDll (0x77f889a9)
   LdrpLoadDll 0x77f887e0
      LdrpCheckForLoadedDll (0x77f87122)
      LdrpMapDll (0x77f8bc77)
         LdrpCheckForKnownDll (0x77f8c62b)
         LdrpResolveDllName (0x77f8c3df)
         LdrpCreateDllSection (0x77f8c355)
         LdrpAllocateDataTableEntry (0x77f8be69)
         LdrpFetchAddressOfEntryPoint (0x77f8bf23)
         LdrpInsertMemoryTableEntry (0x77f8bebb)
      LdrpWalkImportDescriptor (0x77f8be15)
         LdrpLoadImportModule (0x77f8bfd1)
```

```
            *LdrpCheckForLoadedDll
      LdrpUpdateLoadCount (0x77f88afa)
         *LdrpCheckForLoadedDll
         **LdrpUpdateLoadCount
      LdrpRunInitializeRoutines (0x77f8bcb8)
      LdrpClearLoadInProgress (0x77f88c12)

#5 - LoadLibraryExW 3rd Parameter is LOAD_LIBRARY_AS_DATAFILE:
LdrpCheckForLoadedDll (0x77f87122)

All addresses based upon Windows 2000 Professional (Build 2195: Service Pack 1)
* previously documented
** recursive call
```

Figure 24 APIs Forwarded to NTDLL

API	Destination
DeleteCriticalSection	Forwarded to NTDLL.RtlDeleteCriticalSection
EnterCriticalSection	Forwarded to NTDLL.RtlEnterCriticalSection
HeapAlloc	Forwarded to NTDLL.RtlAllocateHeap
HeapFree	Forwarded to NTDLL.RtlFreeHeap
HeapReAlloc	Forwarded to NTDLL.RtlReAllocateHeap
HeapSize	Forwarded to NTDLL.RtlSizeHeap
LeaveCriticalSection	Forwarded to NTDLL.RtlLeaveCriticalSection
RtlFillMemory	Forwarded to NTDLL.RtlFillMemory
RtlMoveMemory	Forwarded to NTDLL.RtlMoveMemory
RtlUnwind	Forwarded to NTDLL.RtlUnwind
RtlZeroMemory	Forwarded to NTDLL.RtlZeroMemory
SetCriticalSectionSpinCount	Forwarded to NTDLL.RtlSetCriticalSection- SpinCount
TryEnterCriticalSection	Forwarded to NTDLL.RtlTryEnterCriticalSection
VerSetConditionMask	Forwarded to NTDLL.VerSetConditionMask

Figure 25 Binding via LdrpSnapIAT and LdrpSnapThunk for Forwarded.DLL

```
          IAT        IAT
  Address     File       Memory  API Name (* - Forwarded APIs)
  1200B000  0000C314 => 77E851E1 HeapCreate
* 1200B004  0000C322 => 77FCA535 HeapFree (RtlFreeHeap)
  1200B008  0000C0AA => 77E8F07F GetCommandLineA
  1200B00C  0000C0BC => 77E85C77 GetVersion
  1200B010  0000C0CA => 77EA83DE DbgBreakPoint
  1200B014  0000C0D8 => 77E8F043 GetStdHandle
  1200B018  0000C0E8 => 77E8334F WriteFile
  1200B01C  0000C0F4 => 77E82EF1 InterlockedDecrement
  1200B020  0000C10C => 77E9E0C8 OutputDebugStringA
  1200B024  0000C122 => 77E87031 GetProcAddress
  1200B028  0000C134 => 77E87273 LoadLibraryA
  1200B02C  0000C144 => 77E82EE0 InterlockedIncrement
  1200B030  0000C15C => 77E88885 GetModuleFileNameA
  1200B034  0000C172 => 77E8F32D ExitProcess
  1200B038  0000C180 => 77EB45FF TerminateProcess
  1200B03C  0000C194 => 77E8304F GetCurrentProcess
  1200B040  0000C1A8 => 77E83510 GetCurrentThreadId
  1200B044  0000C1BE => 77E836DD TlsSetValue
  1200B048  0000C1CC => 77E8C512 TlsAlloc
  1200B04C  0000C1D8 => 77E8F254 TlsFree
  1200B050  0000C1E2 => 77E83008 SetLastError
  1200B054  0000C1F2 => 77E83025 TlsGetValue
  1200B058  0000C200 => 77E8301B GetLastError
  1200B05C  0000C210 => 77E831E7 SetHandleCount
  1200B060  0000C222 => 77E84C93 GetFileType
  1200B064  0000C230 => 77E8F10A GetStartupInfoA
* 1200B068  0000C242 => 77F837C4 DeleteCriticalSection (RtlDeleteCriticalSection)
  1200B06C  0000C25A => 77E83A61 IsBadWritePtr
  1200B070  0000C26A => 77E84F4F IsBadReadPtr
  1200B074  0000C27A => 77E8BC6C HeapValidate
  1200B078  0000C28A => 77E82116 FreeEnvironmentStringsA
  1200B07C  0000C2A4 => 77E8F085 FreeEnvironmentStringsW
  1200B080  0000C2BE => 77E8593F WideCharToMultiByte
  1200B084  0000C2D4 => 77E9C60D GetEnvironmentStrings
  1200B088  0000C2EC => 77E8324E GetEnvironmentStringsW
  1200B08C  0000C306 => 77E8523C HeapDestroy
  1200B090  0000C41E => 77E841B6 LCMapStringW
  1200B094  0000C40E => 77E95278 LCMapStringA
  1200B098  0000C32E => 77E85194 VirtualFree
  1200B09C  0000C33C => 77E83833 InitializeCriticalSection
* 1200B0A0  0000C358 => 77F81B42 EnterCriticalSection (RtlEnterCriticalSection)
```

```
* 1200B0A4  0000C370 => 77F81B73 LeaveCriticalSection (RtlLeaveCriticalSection)
* 1200B0A8  0000C388 => 77FCA055 HeapAlloc (RtlAllocateHeap)
* 1200B0AC  0000C394 => 77F85B48 HeapReAlloc (RtlReAllocateHeap)
  1200B0B0  0000C3A2 => 77E850EC VirtualAlloc
  1200B0B4  0000C3B2 => 77E8EFB8 GetCPInfo
  1200B0B8  0000C3BE => 77E83852 GetACP
  1200B0BC  0000C3C8 => 77E8724D GetOEMCP
  1200B0C0  0000C3D4 => 77E84035 MultiByteToWideChar
  1200B0C4  0000C3EA => 77E9740C GetStringTypeA
  1200B0C8  0000C3FC => 77E867D4 GetStringTypeW
  1200B0CC  0000C42E => 77E853E8 SetFilePointer
* 1200B0D0  0000C440 => 77F8E13A RtlUnwind (RtlUnwind)
  1200B0D4  0000C44C => 77E8F3BC SetStdHandle
  1200B0D8  0000C45C => 77E863C1 FlushFileBuffers
  1200B0DC  0000C470 => 77E83053 CloseHandle
  1200B0E0  00000000    00000000
  1200B0E4  0000C090 => 77E1F098 MessageBoxW
  1200B0E8  00000000    00000000

********************************************************************************
LdrSnap Display (Note the displays of forwarded APIs):

LDR: LdrLoadDll, loading Forwarder.DLL from
    E:\PROJECTS\TEMP\Test\debug;.;D:\WINNT\System32;D:\WINNT\system;D:\WINNT;...
LDR: Loading (DYNAMIC) E:\PROJECTS\TEMP\Test\debug\Forwarder.DLL
LDR: KERNEL32.dll used by Forwarder.DLL
LDR: Snapping imports for Forwarder.DLL from KERNEL32.dll
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlEnterCriticalSection
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlDeleteCriticalSection
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlFreeHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlLeaveCriticalSection
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlAllocateHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlReAllocateHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlUnwind
LDR: Real INIT LIST
    E:\PROJECTS\TEMP\Test\debug\Forwarder.DLL init routine 110010e9
LDR: Forwarder.DLL loaded. - Calling init routine at 110010e9
```

Figure 26 Pre-binding using SDK Bind

```
          IAT
         File &
 Address    Memory  API Name (* - Forwarded APIs)
 10004000  77E851E1 HeapCreate
 10004004  77E85C77 GetVersion
 10004008  77E8F32D ExitProcess
```

```
  1000400C  77EB45FF TerminateProcess
  10004010  77E8304F GetCurrentProcess
  10004014  77E83510 GetCurrentThreadId
  10004018  77E836DD TlsSetValue
  1000401C  77E8C512 TlsAlloc
  10004020  77E8F254 TlsFree
  10004024  77E83025 TlsGetValue
  10004028  77E831E7 SetHandleCount
  1000402C  77E8F043 GetStdHandle
  10004030  77E84C93 GetFileType
  10004034  77E8F10A GetStartupInfoA
* 10004038  77F837C4 DeleteCriticalSection (RtlDeleteCriticalSection)
  1000403C  77E88885 GetModuleFileNameA
  10004040  77E82116 FreeEnvironmentStringsA
  10004044  77E8F085 FreeEnvironmentStringsW
  10004048  77E8593F WideCharToMultibyte
  1000404C  77E9C60D GetEnvironmentStrings
  10004050  77E8324E GetEnvironmentStringsW
  10004054  77E8523C HeapDestroy
  10004058  77E8F07F GetCommandLineA
  1000405C  77E85194 VirtualFree
* 10004060  77FCA535 HeapFree (RtlFreeHeap)
  10004064  77E8334F WriteFile
  10004068  77E83833 InitializeCriticalSection
* 1000406C  77F81B42 EnterCriticalSection (RtlEnterCriticalSection)
* 10004070  77F81B73 LeaveCriticalSection (RtlLeaveCriticalSection)
* 10004074  77FCA055 HeapAlloc (RtlAllocateHeap)
  10004078  77E8EFB8 GetCPInfo
  1000407C  77E83852 GetACP
  10004080  77E8724D GetOEMCP
  10004084  77E850EC VirtualAlloc
* 10004088  77F85B48 HeapReAlloc (RtlReAllocateHeap)
  1000408C  77E87031 GetProcAddress
  10004090  77E87273 LoadLibraryA
  10004094  77E84035 MultiByteToWideChar
  10004098  77E95278 LCMapStringA
  1000409C  77E841B6 LCMapStringW
  100040A0  77E9740C GetStringTypeA
  100040A4  77E867D4 GetStringTypeW
* 100040A8  77F8E13A RtlUnwind (RtlUnwind)

********************************************************************************
LdrSnap Display (Note there are no displays of forwarded APIs):

LDR: LdrLoadDll, loading TestDll.DLL from
     E:\PROJECTS\TEMP\Test\debug;.;D:\WINNT\System32;D:\WINNT\system;D:\WINNT;...
LDR: Loading (DYNAMIC) E:\PROJECTS\TEMP\Test\debug\TestDll.DLL
LDR: TestDll.DLL bound to KERNEL32.dll
LDR: TestDll.DLL has correct binding to KERNEL32.dll
LDR: TestDll.DLL bound to NTDLL.DLL via forwarder(s) from KERNEL32.dll
LDR: TestDll.DLL has correct binding to NTDLL.DLL
LDR: Real INIT LIST
     E:\PROJECTS\TEMP\Test\debug\TestDll.DLL init routine 10001109
LDR: TestDll.DLL loaded. - Calling init routine at 10001109
```

Extras

Under the Hood

Code for this article: (Image18, Image19, Image20)

Matt Pietrek does advanced research for the NuMega Labs of Compuware Corporation, and is the author of several books. His Web site at http://www.wheaty.net has a FAQ page and information on previous columns and articles.

In recent columns for *MSJ* (June 1999), I've discussed COM type libraries and database access layers such as ActiveX® Data Objects (ADO) and OLE DB. Longtime readers of my *MSJ* writings (both of them) probably think I've gone soft. To redeem myself, this month I'll tour part of the Windows NT® loader code where the operating system and your code come together. I'll also demonstrate a nifty trick for getting loader status information from the loader, and a related trick you can use in the Developer Studio® debugger.

Consider what you know about EXEs, DLLs, and how they're loaded and initialized. You probably know that when a C++ DLL is loaded, its DllMain function is called. Think about what happens when your EXE implicitly links to some set of DLLs (for example, KERNEL32.DLL and USER32.DLL). In what order will those DLLs be initialized? Is it possible for one of your DLLs to be initialized before another DLL that you depend on? The Platform SDK has this to say under the "Dynamic-Link Library Entry-Point Function" section.

Your function should perform only simple initialization tasks, such as setting up thread local storage (TLS), creating synchronization objects, and opening files. It must not call the LoadLibrary function, because this may create dependency loops in the DLL load order. This can result in a DLL being used before the system has executed its initialization code. Similarly, you must not call the FreeLibrary function in the entry-point function, because this can result in a DLL being used after the system has executed its termination code.

Calling Win32® functions other than TLS, synchronization, and file functions may also result in problems that are difficult to diagnose. For example, calling User, Shell, and COM functions can cause access violation errors, because some functions in their DLLs call LoadLibrary to load other system components.

Something I've learned firsthand is that the above documentation is still way too vague. For example, reading a registry key is a natural thing you'd want to do inside your DllMain function. It certainly qualifies as initialization. Unfortunately, in the right circumstances ADVAPI32.DLL isn't initialized before your DllMain code, and the registry APIs will just fail.

Given the stern warning about using LoadLibrary in the documentation, it's especially interesting that the Windows NT USER32.DLL explicitly ignores the preceding advice. You may be aware of a Windows NT only registry key called AppInit_Dlls that loads a list of DLLs into each process. It turns out that the actual loading of these DLLs occurs as part of USER32's initialization. USER32 looks at this registry key and calls LoadLibrary for these DLLs in its DllMain code. A little thought here reveals that the AppInit_Dlls trick doesn't work if your app doesn't use USER32.DLL. But I digress.

My point in bringing this up is that DLL loading and initialization is still a gray area. In most cases, a simplified view of how the OS loader works is sufficient. In those oddball 5 percent of cases, however, you can go nuts unless you have a more detailed working model of how the OS loader behaves.

Load 'er Up!

hat most programmers think of as module loading is actually two distinct steps. Step one is to map the EXE or DLL into memory. As this occurs, the loader looks at the Import Address Table (IAT) of the module and determines whether the module depends on additional DLLs. If the DLLs aren't already loaded in that process, the loader maps them in as well. This procedure recurses until all of the dependent modules have been mapped into memory. A great way to see all the implicitly dependent DLLs for a given executable is the DEPENDS program from the Platform SDK.

Step two of module loading is to initialize all of the DLLs. Stop and ponder this. While the OS loader is mapping the EXE and/or DLLs into memory in step one, it's not calling the initialization routines. The initialization routines are called after all the modules have been mapped into memory. Key point: the order in which DLLs are mapped into memory is not necessarily the same as the order in which the DLLs are initialized. I've seen people look at the DLL mapping notifications as they appear in the Developer Studio debugger and mistakenly assume that the DLLs were initialized in that same order.

In Windows NT, the routine that invokes the entry point of EXEs and DLLs is called LdrpRunInitializeRoutines, and it's worth taking a look at here. In my own work, I've stepped through the assembler code for LdrpRunInitializeRoutines many times. However, looking at a ream of assembler code isn't the best way to understand it. Therefore, I rewrote LdrpRunInitializeRoutines from Windows NT 4.0 SP3 in C++-like pseudocode, with the results shown in **Figure 18**. To be completely accurate, in NTDLL.DBG the routine name is __stdcall mangled to _LdrpRunInitializeRoutines@4. Also, in my pseudocode, unless a variable or structure name is prefixed with an underscore, it was a name I made up.

LdrpRunInitializeRoutines is the final stop in the Windows NT loader code before an EXE's or DLL's specified entry point is called. *(In the following discussion, I'll use "entry point" and "initialization routine" interchangeably.)* This loader code executes in the process context that loaded the DLL-that is, it's not part of some special loader process. LdrpRunInitializeRoutines is called at least once during process startup to handle implicitly loaded DLLs. LdrpRunInitializeRoutines is also called every time one or more DLLs is dynamically loaded, usually because of a call to LoadLibrary.

Each time LdrpRunInitializeRoutines executes, it seeks out and calls the entry point of all DLLs that have been mapped into memory, but not yet initialized. In examining the pseudocode, take note of all the extra code that provides trace output, even in the nonchecked builds of Windows NT. I'm referring to all the code that uses the _ShowSnaps variable and the _DbgPrint function. I'll come back to these players later.

At a high level, the function breaks up into four distinct sections. The first portion of the code calls _LdrpClearLoadInProgress. This NTDLL function returns the number of DLLs that have just been mapped into memory. For example, if you called LoadLibrary on FOO.DLL and FOO had implicit links to BAR.DLL and BAZ.DLL, _LdrpClearLoadInProgress would return 3 since three DLLs were mapped into memory.

After the number of DLLs to be concerned with is known, LdrpRunInitializeRoutines calls _RtlAllocateHeap (also known as HeapAlloc) to get memory for an array of pointers. In the pseudocode I've called this array pInitNodeArray. Each pointer in pInitNodeArray will eventually point to a structure containing information about the newly loaded (but not yet initialized) DLL.

In the second part of LdrpRunInitializeRoutines, the code digs into internal process data structures to obtain a linked list containing each of the newly loaded DLLs. As the code iterates through the linked list, it checks to see if the loader has somehow seen this DLL before *(not likely)*. It also checks to ensure that the DLL has an entry point. If both tests are passed, the code appends the module information pointer to the pInitNodeArray. The pseudocode refers to the module information as pModuleLoaderInfo. Note that it's entirely possible for a DLL to not have an entry point-for example, a resource-only DLL. Thus, the number of entries in pInitNodeArray may be fewer than the value returned earlier by _LdrpClearLoadInProgress.

The third *(and largest)* section of LdrpRunInitializeRoutines is where things really start to happen. The code's mission here is to enumerate through each element in pInitNodeArray and call the entry point. Because of the very real possibility that a DLL's initialization code may fault, the entire third section of code is surrounded by a __try block. This is why a dynamically loaded DLL can fault in its DllMain without bringing the whole process down.

Iterating through an array and calling an entry point for each node should be a small task. However, some relatively obscure features of Windows NT add to the complexity. For starters, consider whether the process is being debugged by a Win32 debugger such as MSDEV.EXE. Windows NT has an option that allows you to suspend a process and send control to the debugger before a DLL is initialized. This feature is on a per-DLL basis, and is enabled by adding a string value *(BreakOnDllLoad)* to a registry key with the name of the DLL *(for instance, FOO.DLL)*. See the pseudocode comment above the call to _LdrQueryImageFileExecutionOptions in **Figure 18** for more information.

Another bit of extra code that may execute before a DLL's entry point invocation is the TLS initialization. When you declare TLS variables using __declspec*(thread)*, the linker includes data that causes this condition to be triggered. Right before the DLL's entry point is called, LdrpRunInitializeRoutines checks to see if a TLS initialization is necessary and, if so, calls _LdrpCallTlsInitializers. More on this later.

The moment of truth finally comes when LdrpRunInitializeRoutines calls the DLL's entry point. I deliberately left this part of the pseudocode in assembly language. You'll see why later. The crucial instruction is CALL EDI. Here, EDI points to the DLL's entry point, which is specified in the DLL's PE header. When CALL EDI returns, the DLL in question has completed its initialization. For DLLs written in C++, this means that the DllMain code has executed its DLL_PROCESS_ATTACH code. Also, note the third parameter to the entry point, normally referred to as pvReserved. In truth, this parameter is nonzero for DLLs that the EXE implicitly links to directly or through another DLL. The third parameter is zero for all other DLLs (that is, DLLs loaded as a result of a LoadLibrary call).

After the DLL entry point is invoked, LdrpRunInitializeRoutines does a sanity check to make sure the DLL entry point code was defined properly. The loader code looks at the stack pointer (ESP) value from before and after the entry point call. If they're different, something's wrong with the DLL's initialization function. Since most programmers never define the real DLL entry point function, this scenario rarely happens. However, when it does, you're informed of the problem via an onerous dialog (see **Figure 2**). I had to use a debugger and modify a register value at just the right spot to produce this dialog.

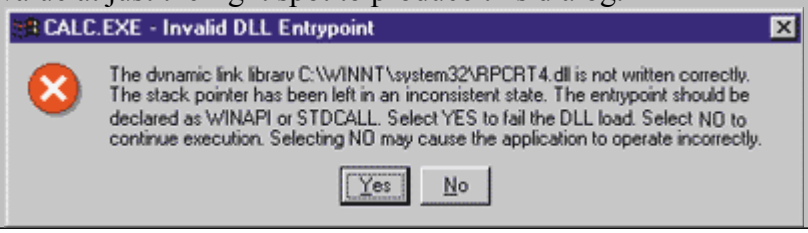

Figure 2 An Invalid DLL Entry Point

Following the stack check, LdrpRunInitializeRoutines checks the return code from the entry point routine. For C++ DLLs, this is the value returned from DllMain. If the DLL returned 0, it usually means something is wrong and that the DLL doesn't want to remain loaded. When this happens, you get the dreaded "DLL Initialization Failed" dialog.

The final portion of the third section of LdrpRunInitializeRoutines occurs after all the DLLs have been initialized. If the process EXE itself has TLS data, and if the implicitly linked DLLs are being initialized, the code calls _LdrpCallTlsInitializers.

The fourth *(and final)* section of LdrpRunInitializeRoutines is the cleanup code. Remember earlier, when _RtlAllocateHeap created the pInitNodeArray? This memory needs to be freed, which occurs inside a __finally block. Even if one of the DLLs faults in its initialization code, the __try/__finally code ensures that _RtlFreeHeap is called to free pInitNodeArray

This ends our tour of LdrpRunInitializeRoutines, so let's now look at some side topics that the code presents.

Debugging Initialization Routines

Every once in a while I come across a problem where a DLL is faulting in its initialization code. Unfortunately, the fault could be from any one of several DLLs, and the operating system doesn't tell me which DLL is the culprit. In these circumstances you can get sneaky and use a debugger breakpoint to narrow down the problem.

Most debuggers blithely skip past the initialization of statically linked DLLs. They're focused on getting you to the first instruction or first line in your EXE. However, knowing what LdrpRunInitializeRoutines looks like, you can set a breakpoint on the CALL EDI instruction where execution goes to the DLL entry point. Once the breakpoint is set, each time a DLL is about to get its DLL_PROCESS_ATTACH notification you'll stop in NTDLL at the CALL instruction. **Figure 3** shows what this looks like in the Visual C++® 6.0 IDE (MSDEV.EXE).

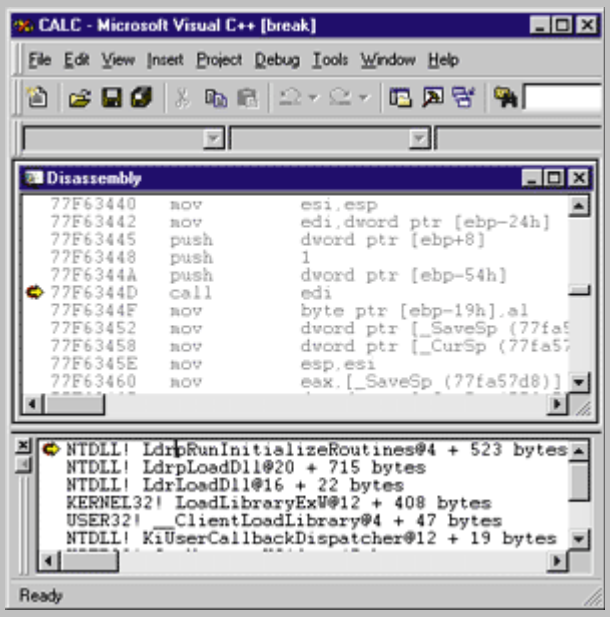

Figure 3 Setting a Breakpoint on CALL EDI

If you choose to step into the CALL, you'll end up at the first instruction of the entry point of the DLL. It's important to understand that this code is almost never code you write. Rather, it's usually code in the runtime library that does its setup work and then calls your initialization code. For example, in a DLL written in Visual C++, the entry point is _DllMainCRTStartup, which is in CRTDLL.C. Without symbol tables or source code, what you'll see in the MSDEV assembly window will look something like **Figure 4**.

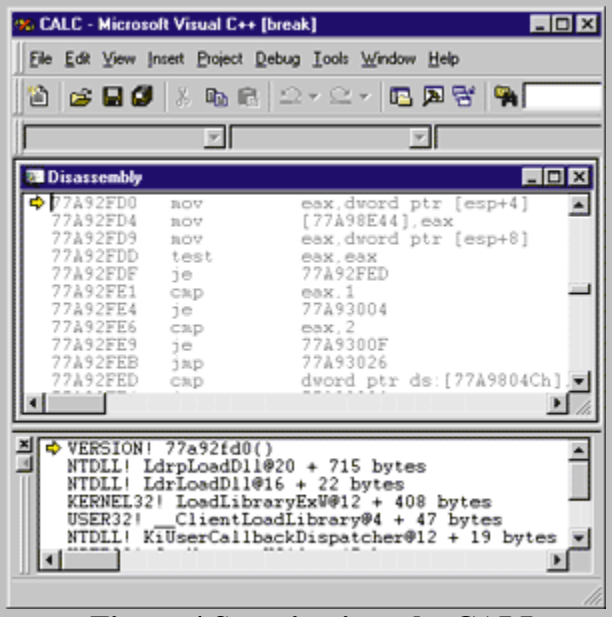

Figure 4 Stepping into the CALL

Usually my debugging process follows a predictable pattern. Step one is to figure out which DLL is faulting. Do this by setting the aforementioned breakpoint, and make one instruction step into each DLL as it initializes. Using the debugger, figure out which DLL you're in, and write it down. One way to do this is to use the memory window to look on the stack (ESP) and obtain the HMODULE of the DLL you've entered.

After you know which DLL you've entered, let the process continue *(Go)*. In short order, the breakpoint should be hit again for the next DLL. Repeat this as often as necessary until you identify the problem DLL. You'll recognize the problem DLL because it will be called to initialize, but the process terminates before the initialization code returns.

Step two is to drill into the faulting DLL. If the offending DLL is one that you have source for, try setting a breakpoint on your DllMain code, then let the process run to see if your breakpoint is hit. If you don't have source, just run the process. Your breakpoint on the CALL EDI instruction should still be in place from before. Keep running until you get to the one where the initialization faults. Step into this entry point, and keep stepping until you can ascertain the problem. This may require stepping through a lot of assembly code! I never said this was easy, but sometimes it's the only way to hunt the problem down.

Finding the CALL EDI instruction can be tricky *(at least with the current Microsoft® debuggers)*. You can see why I deliberately left this part of the pseudocode in assembler. For starters, you'll definitely need to have the correct NTDLL.DBG in your SYSTEM32 directory, alongside NTDLL.DLL. The debugger should automatically load the symbol table when you begin stepping through your program.

Using the assembly window in Visual C++, you can (in theory) goto an address using a symbolic name. Here, you'd want to go to _LdrpRunInitializeRoutines@4 and then scroll down until you see the CALL EDI

instruction. Unfortunately, the Visual C++ debugger doesn't recognize NTDLL symbol names unless you're already stopped in NTDLL.DLL.

If you happen to know the address of _LdrpRunInitializeRoutines@4 *(for instance, 0x77F63242 in Windows NT 4.0 SP 3 for Intel)*, you can type that in and the assembly window will happily display it. Heck, the IDE will even show you that it's the start of a function called _LdrpRunInitializeRoutines@4.
If you're not a debugger guru, the failure to recognize the symbol name is extremely confusing. If you are a debugger nut like me, it's extremely annoying because you know what's causing the problem.

WinDBG from the Platform SDK is a little better about recognizing symbol names. Once you've started the target process, you can set a breakpoint on _LdrpRunInitializeRoutines@4 using its name. Unfortunately, the first time you execute the process, execution blows past _LdrpRunInitializeRoutines@4 before you get a chance to set your breakpoint. To remedy this, start WinDBG, make one instruction step, set the breakpoint, stop debugging, and remain in the debugger. You can then restart the debuggee and the breakpoint will be hit on every invocation of _LdrpRunInitializeRoutines@4. This same trick works in the Visual C++ debugger.

What's This ShowSnaps Thing?

One of the first things that jumped out at me when I looked at the LdrpRunInitializeRoutines code was the _ShowSnaps global variable. Now's a good time to briefly divert to the subject of the GlobalFlag and GFlags.EXE.

The Windows NT registry contains a DWORD value that influences certain behaviors of system code. Most of these modifications are heap- and debugging-related. The GlobalFlag value of the registry key

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager
```

is a set of bitfields. Knowledge Base article Q147314 describes most of the bitfields, so I won't go into them here. In addition to this systemwide GlobalFlag value, individual executables can have their own distinct GlobalFlag value. The process-specific GlobalFlag value is found under

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Image File Execution Options\imagename
```

where imagename is the name of an executable *(for instance, WinWord.exe)*. All of these documentation-challenged bitfields and highly nested resource keys scream out for a program to simplify it all. Wouldn't you know it, Microsoft has just such a program.

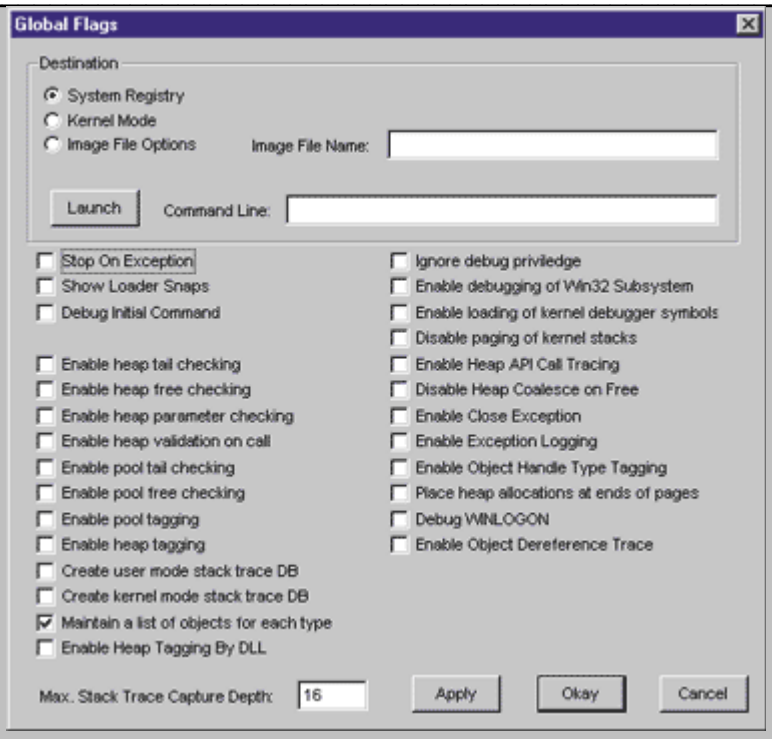

Figure 5 GFlags.EXE

Figure 5 shows GFlags.EXE, which comes in the Windows NT 4.0 Resource Kit. Near the top-left of the GFlags window are three radio buttons. Selecting either of the top two *(System Registry or Kernel Mode)* lets you make changes to the global Session Manager value of GlobalFlags. If you select the third radio button *(Image File Options)*, the set of available option checkboxes shrinks dramatically. This is because some of the GlobalFlag options only affect kernel mode code and don't make sense on a per-process basis. It's important to note that most of the kernel mode-only options assume you're using a system-level debugger such as i386kd. Without such a debugger to poke around or receive the output, there's not much use in enabling these options.

To tie this back to the subject of _ShowSnaps, enabling the Show Loader Snaps option in GFlags causes the _ShowSnaps variable to be set to a nonzero value in NTDLL.DLL. In the registry, this bit value is 0x00000002, which is #defined as FLG_SHOW_LDR_SNAPS. Luckily, this bitflag is one of the GlobalFlag values that can be enabled on a per-process basis. The output can be quite voluminous if you enable it systemwide.

Examining ShowSnaps Output

Let's take a look at what sort of output enabling Show Loader Snaps produces. It turns out that other parts of the Windows NT loader that I haven't discussed also check this variable and emit additional output. **Figure 19** shows some abbreviated output from running CALC.EXE. To get the text, I first used GFlags to turn on Show Loader Snaps for CALC.EXE. Next, I ran CALC.EXE under the control of MSDEV.EXE and captured the output from the debug pane.

In **Figure 19**, note that all the output originating in NTDLL is preceded by an LDR: prefix. Other lines in the output *(for instance, "Loaded symbols for XXX")* were inserted by the MSDEV process. In looking at the LDR: output, there's a wealth of information. For example, as the process starts, the complete path to the EXE is given, along with the current directory and search path.

As NTDLL loads each DLL and fixes up imported functions, you'll see lines like this:

- LDR: ntdll.dll used by SHELL32.dll
- LDR: Snapping imports for SHELL32.dll from ntdll.dll

The first line decrees that SHELL32.DLL links to APIs in NTDLL. The second line shows that the imported NTDLL APIs are being "snapped." When an executable module imports functions from another DLL, an array of function pointers resides in the importing module. This array of function pointers is known as the IAT. One of the loader's jobs is to locate the addresses of the imported functions and punch them into the IAT. Hence, the term "snapping" in the LDR: output.

Another interesting set of lines in the output shows bound DLLs being handled:

- LDR: COMCTL32.dll bound to KERNEL32.dll
- LDR: COMCTL32.dll has correct binding to KERNEL32.dll

In previous columns, I've talked about the binding process done by BIND.EXE or the BindImageEx API in IMAGEHLP.DLL. Binding an executable to a DLL is the act of looking up the address of the imported APIs and writing them to the importing executable. This speeds up the loading process since the imported addresses don't have to be looked up at load time.

The first line in the above output indicates that the COMCTL32 has bound against APIs in KERNEL32.DLL. The second line indicates that the bound addresses are correct. The loader does this by comparing timestamps. If the timestamps don't match, the binding is invalid. In this case, the loader has to look up the imported addresses just as if the executable hadn't been bound in the first place.

TLS Initialization

I'll finish up this column by showing pseudocode for one other routine. In LdrpRunInitializeRoutines, right before the module's entry point is called, NTDLL checks to see if the module needs TLS initialization. If so, it calls LdrpCallTlsInitializers. **Figure 20** shows my pseudocode for this routine.

The code in LdrpCallTlsInitializers is simple enough. Located in the PE header is an offset (RVA) to an IMAGE_TLS_DIRECTORY structure (defined in WINNT.H). The code calls RtlImageDirectoryEntryToData to obtain a pointer to this structure. Within the IMAGE_TLS_DIRECTORY structure is a pointer to an array of callback function addresses to be invoked. These functions are prototyped as IMAGE_TLS_CALLBACK functions, which are defined in WINNT.H. Not coincidentally, the TLS initialization callbacks happen to look just like a DllMain function. For what it's worth, when using __declspec(thread) variables, Visual C++ emits data that causes this routine to be invoked. However, no actual callbacks are currently defined by the runtime library, so the array of function pointers is a single NULL entry.

Conclusion

This wraps up my coverage of Windows NT module initialization. Obviously, I have skipped or skimmed over a lot of related material. For example, what is the algorithm for determining the order in which the modules will be initialized? The algorithm Windows NT uses has changed at least once, and it would be nice to have a Microsoft technical note that at least gives some guidelines. Likewise, I haven't covered the mirror image topic: module unloading. However, I hope this glimpse into the inner workings of the Windows NT loader has provided you with material for further exploration.

Peering Inside the PE: A Tour of the Win32 Portable Executable File Format

Matt Pietrek - March 1994

Matt Pietrek is the author of Windows Internals (Addison-Wesley, 1993). He works at Nu-Mega Technologies Inc., and can be reached via CompuServe: 71774,362

The format of an operating system's executable file is in many ways a mirror of the operating system. Although studying an executable file format isn't usually high on most programmers' list of things to do, a great deal of knowledge can be gleaned this way. In this article, I'll give a tour of the Portable Executable (PE) file format that Microsoft has designed for use by all their Win32®-based systems: Windows NT®, Win32s™, and Windows® 95.

The PE format plays a key role in all of Microsoft's operating systems for the foreseeable future, including Windows 2000. If you use Win32s or Windows NT, you're already using PE files. Even if you program only for Windows 3.1 using Visual C++®, you're still using PE files (the 32-bit MS-DOS® extended components of Visual C++ use this format). In short, PEs are already pervasive and will become unavoidable in the near future. Now is the time to find out what this new type of executable file brings to the operating system party.

I'm not going to make you stare at endless hex dumps and chew over the significance of individual bits for pages on end. Instead, I'll present the concepts embedded in the PE file format and relate them to things you encounter everyday. For example, the notion of thread local variables, as in

```
declspec(thread) int i;
```

drove me crazy until I saw how it was implemented with elegant simplicity in the executable file. Since many of you are coming from a background in 16-bit Windows, I'll correlate the constructs of the Win32 PE file format back to their 16-bit NE file format equivalents.

In addition to a different executable format, Microsoft also introduced a new object module format produced by their compilers and assemblers. This new OBJ file format has many things in common with the PE executable format. I've searched in vain to find any documentation on the new OBJ file format. So I deciphered it on my own, and will describe parts of it here in addition to the PE format.

It's common knowledge that Windows NT has a VAX® VMS® and UNIX® heritage. Many of the Windows NT creators designed and coded for those platforms before coming to Microsoft. When it came time to design Windows NT, it was only natural that they tried to minimize their bootstrap time by using previously written and tested tools. The executable and object module format that these tools produced and worked with is called COFF (an acronym for Common Object File Format). The relative age of COFF can be seen by things such as fields specified in octal format. The COFF format by itself was a good starting point, but needed to be extended to meet all the needs of a modern operating system like Windows NT or Windows 95. The result of this updating is the Portable Executable format. It's called *"portable"* because all the implementations of Windows NT on various platforms (x86, MIPS®, Alpha, and so on) use the same executable format. Sure, there are differences in things like the binary encodings of CPU instructions.

The important thing is that the operating system loader and programming tools don't have to be completely rewritten for each new CPU that arrives on the scene.

The strength of Microsoft's commitment to get Windows NT up and running quickly is evidenced by the fact that they abandoned existing 32-bit tools and file formats. Virtual device drivers written for 16-bit Windows were using a different 32-bit file layout—the LE format—long before Windows NT appeared on the scene. More important than that is the shift of OBJ formats. Prior to the Windows NT C compiler, all Microsoft compilers used the Intel OMF (Object Module Format) specification. As mentioned earlier, the Microsoft compilers for Win32 produce COFF-format OBJ files. Some Microsoft competitors such as Borland and Symantec have chosen to forgo the COFF format OBJs and stick with the Intel OMF format. The upshot of this is that companies producing OBJs or LIBs for use with multiple compilers will need to go back to distributing separate versions of their products for different compilers *(if they weren't already)*.

The PE format is documented (in the loosest sense of the word) in the WINNT.H header file. About midway through WINNT.H is a section titled "Image Format." This section starts out with small tidbits from the old familiar MS-DOS MZ format and NE format headers before moving into the newer PE information. WINNT.H provides definitions of the raw data structures used by PE files, but contains only a few useful comments to make sense of what the structures and flags mean. Whoever wrote the header file for the PE format (the name Michael J. O'Leary keeps popping up) is certainly a believer in long, descriptive names, along with deeply nested structures and macros. When coding with WINNT.H, it's not uncommon to have expressions like this:

```
pNTHeader->
OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_DEBUG].VirtualAddress;
```

To help make logical sense of the information in WINNT.H, read the Portable Executable and Common Object File Format Specification, available on MSDN Library quarterly CD-ROM releases up to and including October 2001.

Turning momentarily to the subject of COFF-format OBJs, the WINNT.H header file includes structure definitions and typedefs for COFF OBJ and LIB files. Unfortunately, I've been unable to find any documentation on this similar to that for the executable file mentioned above. Since PE files and COFF OBJ files are so similar, I decided that it was time to bring these files out into the light and document them as well.

Beyond just reading about what PE files are composed of, you'll also want to dump some PE files to see these concepts for yourself. If you use Microsoft® tools for Win32-based development, the DUMPBIN program will dissect and output PE files and COFF OBJ/LIB files in readable form. Of all the PE file dumpers, DUMPBIN is easily the most comprehensive. It even has a nifty option to disassemble the code sections in the file it's taking apart. Borland users can use TDUMP to view PE executable files, but TDUMP doesn't understand the COFF OBJ files. This isn't a big deal since the Borland compiler doesn't produce COFF-format OBJs in the first place.

I've written a PE and COFF OBJ file dumping program, PEDUMP *(see Table 1)*, that I think provides more understandable output than DUMPBIN. Although it doesn't have a disassembler or work with LIB files, it is otherwise functionally equivalent to DUMPBIN, and adds a few new features to make it worth considering. The source code for PEDUMP is available on any MSJ bulletin board, so I won't list it here in its entirety. Instead, I'll show sample output from PEDUMP to illustrate the concepts as I describe them.

Table 1. PEDUMP.C

```
//--------------------
```

```
// PROGRAM: PEDUMP
// FILE:    PEDUMP.C
// AUTHOR:  Matt Pietrek - 1993
//--------------------
#include <windows.h>
#include <stdio.h>
#include "objdump.h"
#include "exedump.h"
#include "extrnvar.h"

// Global variables set here, and used in EXEDUMP.C and OBJDUMP.C
BOOL fShowRelocations = FALSE;
BOOL fShowRawSectionData = FALSE;
BOOL fShowSymbolTable = FALSE;
BOOL fShowLineNumbers = FALSE;

char HelpText[] =
"PEDUMP - Win32/COFF .EXE/.OBJ file dumper - 1993 Matt Pietrek\n\n"
"Syntax: PEDUMP [switches] filename\n\n"
"  /A    include everything in dump\n"
"  /H    include hex dump of sections\n"
"  /L    include line number information\n"
"  /R    show base relocations\n"
"  /S    show symbol table\n";

// Open up a file, memory map it, and call the appropriate dumping routine
void DumpFile(LPSTR filename)
{
   HANDLE hFile;
   HANDLE hFileMapping;
   LPVOID lpFileBase;
   PIMAGE_DOS_HEADER dosHeader;

   hFile = CreateFile(filename, GENERIC_READ, FILE_SHARE_READ, NULL,
                      OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);

   if ( hFile = = INVALID_HANDLE_VALUE )
   {   printf("Couldn't open file with CreateFile()\n");
      return; }

   hFileMapping = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
   if ( hFileMapping = = 0 )
   {   CloseHandle(hFile);
      printf("Couldn't open file mapping with CreateFileMapping()\n");
      return; }

   lpFileBase = MapViewOfFile(hFileMapping, FILE_MAP_READ, 0, 0, 0);
   if ( lpFileBase = = 0 )
   {
      CloseHandle(hFileMapping);
      CloseHandle(hFile);
      printf("Couldn't map view of file with MapViewOfFile()\n");
      return;
   }

   printf("Dump of file %s\n\n", filename);

   dosHeader = (PIMAGE_DOS_HEADER)lpFileBase;
   if ( dosHeader->e_magic = = IMAGE_DOS_SIGNATURE )
```

```
        { DumpExeFile( dosHeader ); }
    else if ( (dosHeader->e_magic = = 0x014C)    // Does it look like a i386
            && (dosHeader->e_sp = = 0) )        // COFF OBJ file???
    {
        // The two tests above aren't what they look like.  They're
        // really checking for IMAGE_FILE_HEADER.Machine = = i386 (0x14C)
        // and IMAGE_FILE_HEADER.SizeOfOptionalHeader = = 0;
        DumpObjFile( (PIMAGE_FILE_HEADER)lpFileBase );
    }
    else
        printf("unrecognized file format\n");
    UnmapViewOfFile(lpFileBase);
    CloseHandle(hFileMapping);
    CloseHandle(hFile);
}

// process all the command line arguments and return a pointer to
// the filename argument.
PSTR ProcessCommandLine(int argc, char *argv[])
{
    int i;

    for ( i=1; i < argc; i++ )
    {
        strupr(argv[i]);

        // Is it a switch character?
        if ( (argv[i][0] = = '-') || (argv[i][0] = = '/') )
        {
            if ( argv[i][1] = = 'A' )
            {   fShowRelocations = TRUE;
                fShowRawSectionData = TRUE;
                fShowSymbolTable = TRUE;
                fShowLineNumbers = TRUE; }
            else if ( argv[i][1] = = 'H' )
                fShowRawSectionData = TRUE;
            else if ( argv[i][1] = = 'L' )
                fShowLineNumbers = TRUE;
            else if ( argv[i][1] = = 'R' )
                fShowRelocations = TRUE;
            else if ( argv[i][1] = = 'S' )
                fShowSymbolTable = TRUE;
        }
        else    // Not a switch character.  Must be the filename
        {   return argv[i]; }
    }
}

int main(int argc, char *argv[])
{
    PSTR filename;

    if ( argc = = 1 )
    {   printf(    HelpText );
        return 1; }

    filename = ProcessCommandLine(argc, argv);
    if ( filename )
        DumpFile( filename );
```

```
    return 0;
}
```

Win32 and PE Basic Concepts

Let's go over a few fundamental ideas that permeate the design of a PE file (see Figure 1). I'll use the term "module" to mean the code, data, and resources of an executable file or DLL that have been loaded into memory. Besides code and data that your program uses directly, a module is also composed of the supporting data structures used by Windows to determine where the code and data is located in memory. In 16-bit Windows, the supporting data structures are in the module database (the segment referred to by an HMODULE). In Win32, these data structures are in the PE header, which I'll explain shortly.

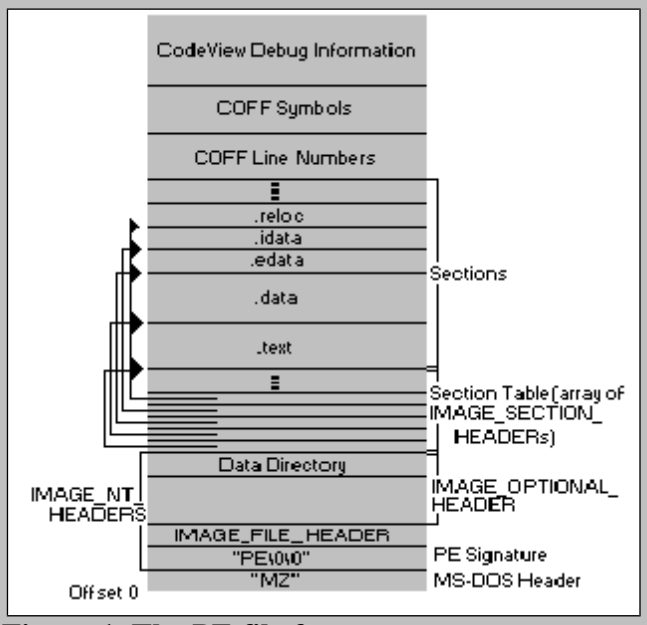

Figure 1. The PE file format

The first important thing to know about PE files is that the executable file on disk is very similar to what the module will look like after Windows has loaded it. The Windows loader doesn't need to work extremely hard to create a process from the disk file. The loader uses the memory-mapped file mechanism to map the appropriate pieces of the file into the virtual address space. To use a construction analogy, a PE file is like a prefabricated home. It's essentially brought into place in one piece, followed by a small amount of work to wire it up to the rest of the world (that is, to connect it to its DLLs and so on). This same ease of loading applies to PE-format DLLs as well. Once the module has been loaded, Windows can effectively treat it like any other memory-mapped file.

This is in marked contrast to the situation in 16-bit Windows. The 16-bit NE file loader reads in portions of the file and creates completely different data structures to represent the module in memory. When a code or data segment needs to be loaded, the loader has to allocate a new segment from the global heap, find where the raw data is stored in the executable file, seek to that location, read in the raw data, and apply any applicable fixups. In addition, each 16-bit module is responsible for remembering all the selectors it's currently using, whether the segment has been discarded, and so on.

For Win32, all the memory used by the module for code, data, resources, import tables, export tables, and other required module data structures is in one contiguous block of memory. All you need to know in this situation is where the loader mapped the file into memory. You can easily find all the various pieces of the module by

following pointers that are stored as part of the image.

Another idea you should be acquainted with is the Relative Virtual Address (RVA). Many fields in PE files are specified in terms of RVAs. An RVA is simply the offset of some item, relative to where the file is memory-mapped. For example, let's say the loader maps a PE file into memory starting at address 0x10000 in the virtual address space. If a certain table in the image starts at address 0x10464, then the table's RVA is 0x464.

```
(Virtual address 0x10464)-(base address 0x10000) = RVA 0x00464
```

To convert an RVA into a usable pointer, simply add the RVA to the base address of the module. The base address is the starting address of a memory-mapped EXE or DLL and is an important concept in Win32. For the sake of convenience, Windows NT and Windows 95 uses the base address of a module as the module's instance handle (HINSTANCE). In Win32, calling the base address of a module an HINSTANCE is somewhat confusing, because the term "instance handle" comes from 16-bit Windows. Each copy of an application in 16-bit Windows gets its own separate data segment (and an associated global handle) that distinguishes it from other copies of the application, hence the term instance handle. In Win32, applications don't need to be distinguished from one another because they don't share the same address space. Still, the term HINSTANCE persists to keep continuity between 16-bit Windows and Win32. What's important for Win32 is that you can call GetModuleHandle for any DLL that your process uses to get a pointer for accessing the module's components.

The final concept that you need to know about PE files is sections. A section in a PE file is roughly equivalent to a segment or the resources in an NE file. Sections contain either code or data. Unlike segments, sections are blocks of contiguous memory with no size constraints. Some sections contain code or data that your program declared and uses directly, while other data sections are created for you by the linker and librarian, and contain information vital to the operating system. In some descriptions of the PE format, sections are also referred to as objects. The term object has so many overloaded meanings that I'll stick to calling the code and data areas sections.

The PE Header

Like all other executable file formats, the PE file has a collection of fields at a known (or easy to find) location that define what the rest of the file looks like. This header contains information such as the locations and sizes of the code and data areas, what operating system the file is intended for, the initial stack size, and other vital pieces of information that I'll discuss shortly. As with other executable formats from Microsoft, this main header isn't at the very beginning of the file. The first few hundred bytes of the typical PE file are taken up by the MS-DOS stub.

This stub is a tiny program that prints out something to the effect of "This program cannot be run in MS-DOS mode." So if you run a Win32-based program in an environment that doesn't support Win32, you'll get this informative error message. When the Win32 loader memory maps a PE file, the first byte of the mapped file corresponds to the first byte of the MS-DOS stub. That's right. With every Win32-based program you start up, you get an MS-DOS-based program loaded for free!

As in other Microsoft executable formats, you find the real header by looking up its starting offset, which is stored in the MS-DOS stub header. The WINNT.H file includes a structure definition for the MS-DOS stub header that makes it very easy to look up where the PE header starts. The e_lfanew field is a relative offset (or RVA, if you prefer) to the actual PE header. To get a pointer to the PE header in memory, just add that field's value to the image base:

```
// Ignoring typecasts and pointer conversion issues for clarity...
pNTHeader = dosHeader + dosHeader->e_lfanew;
```

Once you have a pointer to the main PE header, the fun can begin. The main PE header is a structure of type IMAGE_NT_HEADERS, which is defined in WINNT.H. This structure is composed of a DWORD and two substructures and is laid out as follows:

```
DWORD Signature;
IMAGE_FILE_HEADER FileHeader;
IMAGE_OPTIONAL_HEADER OptionalHeader;
```

The Signature field viewed as ASCII text is "PE\0\0". If after using the e_lfanew field in the MS-DOS header, you find an NE signature here rather than a PE, you're working with a 16-bit Windows NE file. Likewise, an LE in the signature field would indicate a Windows 3.x virtual device driver (VxD). An LX here would be the mark of a file for OS/2 2.0.

Following the PE signature DWORD in the PE header is a structure of type IMAGE_FILE_HEADER. The fields of this structure contain only the most basic information about the file. The structure appears to be unmodified from its original COFF implementations. Besides being part of the PE header, it also appears at the very beginning of the COFF OBJs produced by the Microsoft Win32 compilers. The fields of the IMAGE_FILE_HEADER are shown in Table 2.

Table 2. IMAGE_FILE_HEADER Fields

<table>
<tr><td>WORD Machine</td><td>The CPU that this file is intended for. The following CPU IDs are defined:

<table><tr><td>0x14d</td><td>Intel i860</td></tr><tr><td>0x14c</td><td>Intel I386 (same ID used for 486 and 586)</td></tr><tr><td>0x162</td><td>MIPS R3000</td></tr><tr><td>0x166</td><td>MIPS R4000</td></tr><tr><td>0x183</td><td>DEC Alpha AXP</td></tr></table></td></tr>
<tr><td>WORD NumberOfSections</td><td>The number of sections in the file.</td></tr>
<tr><td>DWORD TimeDateStamp</td><td>The time that the linker (or compiler for an OBJ file) produced this file. This field holds the number of seconds since December 31st, 1969, at 4:00 P.M.</td></tr>
<tr><td>DWORD PointerToSymbolTable</td><td>The file offset of the COFF symbol table. This field is only used in OBJ files and PE files with COFF debug information. PE files support multiple debug formats, so debuggers should refer to the IMAGE_DIRECTORY_ENTRY_DEBUG entry in the data directory (defined later).</td></tr>
<tr><td>DWORD NumberOfSymbols</td><td>The number of symbols in the COFF symbol table. See above.</td></tr>
<tr><td>WORD SizeOfOptionalHeader</td><td>The size of an optional header that can follow this structure. In OBJs, the field is 0. In executables, it is the size of the IMAGE_OPTIONAL_HEADER structure that follows this structure.</td></tr>
<tr><td>WORD Characteristics</td><td>Flags with information about the file. Some important fields:

<table><tr><td>0x0001</td><td>There are no relocations in this file</td></tr><tr><td>0x0002</td><td>File is an executable image (not a OBJ or LIB)</td></tr><tr><td>0x2000</td><td>File is a dynamic-link library, not a program</td></tr></table></td></tr>
</table>

Other fields are defined in WINNT.H

The third component of the PE header is a structure of type IMAGE_OPTIONAL_HEADER. For PE files, this portion certainly isn't optional. The COFF format allows individual implementations to define a structure of additional information beyond the standard IMAGE_FILE_HEADER. The fields in the IMAGE_OPTIONAL_HEADER are what the PE designers felt was critical information beyond the basic information in the IMAGE_FILE_HEADER.

All of the fields of the IMAGE_OPTIONAL_HEADER aren't necessarily important to know about (see Figure 4). The more important ones to be aware of are the ImageBase and the Subsystem fields. You can skim or skip the description of the fields.

Table 3. IMAGE_OPTIONAL_HEADER Fields

WORD Magic	Appears to be a signature WORD of some sort. Always appears to be set to 0x010B.
BYTE MajorLinkerVersion BYTE MinorLinkerVersion	The version of the linker that produced this file. The numbers should be displayed as decimal values, rather than as hex. A typical linker version is 2.23.
DWORD SizeOfCode	The combined and rounded-up size of all the code sections. Usually, most files only have one code section, so this field matches the size of the .text section.
DWORD SizeOfInitializedData	This is supposedly the total size of all the sections that are composed of initialized data (not including code segments.) However, it doesn't seem to be consistent with what appears in the file.
DWORD SizeOfUninitializedData	The size of the sections that the loader commits space for in the virtual address space, but that don't take up any space in the disk file. These sections don't need to have specific values at program startup, hence the term uninitialized data. Uninitialized data usually goes into a section called .bss.
DWORD AddressOfEntryPoint	The address where the loader will begin execution. This is an RVA, and usually can usually be found in the .text section.
DWORD BaseOfCode	The RVA where the file's code sections begin. The code sections typically come before the data sections and after the PE header in memory. This RVA is usually 0x1000 in Microsoft Linker-produced EXEs. Borland's TLINK32 looks like it adds the image base to the RVA of the first code section and stores the result in this field.
DWORD BaseOfData	The RVA where the file's data sections begin. The data sections typically come last in memory, after the PE header and the code sections.
DWORD ImageBase	When the linker creates an executable, it assumes that the file will be memory-mapped to a specific location in memory. That address is stored in this field, assuming a load address allows linker optimizations to take place. If the file really is memory-mapped to that address by the loader, the code doesn't need any patching before it can be run. In executables produced for Windows NT, the default image base is 0x10000. For DLLs, the default is 0x400000. In Windows 95, the address 0x10000 can't be used to load 32-bit EXEs because it lies within a linear address region shared by all processes. Because of this, Microsoft has changed the default base address for Win32 executables to 0x400000. Older programs that were linked assuming a base address of 0x10000 will take longer to load under Windows 95 because the loader needs to apply the base relocations.
DWORD SectionAlignment	When mapped into memory, each section is guaranteed to start at a virtual address that's a multiple of this value. For paging purposes, the default section alignment is 0x1000.
DWORD FileAlignment	In the PE file, the raw data that comprises each section is guaranteed to start at a multiple of this value. The default value is 0x200 bytes, probably to ensure that sections always start at the beginning of a disk sector (which are also 0x200 bytes in length). This field is equivalent to the segment/resource alignment size in

<table>
<tr><td></td><td>NE files. Unlike NE files, PE files typically don't have hundreds of sections, so the space wasted by aligning the file sections is almost always very small.</td></tr>
<tr><td>WORD MajorOperatingSystemVersion
WORD MinorOperatingSystemVersion</td><td>The minimum version of the operating system required to use this executable. This field is somewhat ambiguous since the subsystem fields (a few fields later) appear to serve a similar purpose. This field defaults to 1.0 in all Win32 EXEs to date.</td></tr>
<tr><td>WORD MajorImageVersion
WORD MinorImageVersion</td><td>A user-definable field. This allows you to have different versions of an EXE or DLL. You set these fields via the linker /VERSION switch. For example, "LINK /VERSION:2.0 myobj.obj".</td></tr>
<tr><td>WORD MajorSubsystemVersion
WORD MinorSubsystemVersion</td><td>Contains the minimum subsystem version required to run the executable. A typical value for this field is 3.10 (meaning Windows NT 3.1).</td></tr>
<tr><td>DWORD Reserved1</td><td>Seems to always be 0.</td></tr>
<tr><td>DWORD SizeOfImage</td><td>This appears to be the total size of the portions of the image that the loader has to worry about. It is the size of the region starting at the image base up to the end of the last section. The end of the last section is rounded up to the nearest multiple of the section alignment.</td></tr>
<tr><td>DWORD SizeOfHeaders</td><td>The size of the PE header and the section (object) table. The raw data for the sections starts immediately after all the header components.</td></tr>
<tr><td>DWORD CheckSum</td><td>Supposedly a CRC checksum of the file. As in other Microsoft executable formats, this field is ignored and set to 0. The one exception to this rule is for trusted services and these EXEs must have a valid checksum.</td></tr>
<tr><td>WORD Subsystem</td><td>The type of subsystem that this executable uses for its user interface. WINNT.H defines the following values:

<table>
<tr><td>NATIVE</td><td>1</td><td>Doesn't require a subsystem (such as a device driver)</td></tr>
<tr><td>WINDOWS_GUI</td><td>2</td><td>Runs in the Windows GUI subsystem</td></tr>
<tr><td>WINDOWS_CUI</td><td>3</td><td>Runs in the Windows character subsystem (a console app)</td></tr>
<tr><td>OS2_CUI</td><td>5</td><td>Runs in the OS/2 character subsystem (OS/2 1.x apps only)</td></tr>
<tr><td>POSIX_CUI</td><td>7</td><td>Runs in the Posix character subsystem</td></tr>
</table></td></tr>
<tr><td>WORD DllCharacteristics</td><td>A set of flags indicating under which circumstances a DLL's initialization function (such as DllMain) will be called. This value appears to always be set to 0, yet the operating system still calls the DLL initialization function for all four events.

The following values are defined:

<table>
<tr><td>1</td><td>Call when DLL is first loaded into a process's address space</td></tr>
<tr><td>2</td><td>Call when a thread terminates</td></tr>
<tr><td>4</td><td>Call when a thread starts up</td></tr>
<tr><td>8</td><td>Call when DLL exits</td></tr>
</table></td></tr>
<tr><td>DWORD SizeOfStackReserve</td><td>The amount of virtual memory to reserve for the initial thread's stack. Not all of this memory is committed, however (see the next field). This field defaults to 0x100000 (1MB). If you specify 0 as the stack size to CreateThread, the resulting thread will also have a stack of this same size.</td></tr>
<tr><td>DWORD SizeOfStackCommit</td><td>The amount of memory initially committed for the initial thread's stack. This field defaults to 0x1000 bytes (1 page) for the Microsoft Linker while TLINK32 makes it two pages.</td></tr>
<tr><td>DWORD SizeOfHeapReserve</td><td>The amount of virtual memory to reserve for the initial process</td></tr>
</table>

	heap. This heap's handle can be obtained by calling GetProcessHeap. Not all of this memory is committed (see the next field).
DWORD SizeOfHeapCommit	The amount of memory initially committed in the process heap. The default is one page.
DWORD LoaderFlags	From WINNT.H, these appear to be fields related to debugging support. I've never seen an executable with either of these bits enabled, nor is it clear how to get the linker to set them. The following values are defined: 1. Invoke a breakpoint instruction before starting the process 2. Invoke a debugger on the process after it's been loaded
DWORD NumberOfRvaAndSizes	The number of entries in the DataDirectory array (below). This value is always set to 16 by the current tools.
IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]	An array of IMAGE_DATA_DIRECTORY structures. The initial array elements contain the starting RVA and sizes of important portions of the executable file. Some elements at the end of the array are currently unused. The first element of the array is always the address and size of the exported function table (if present). The second array entry is the address and size of the imported function table, and so on. For a complete list of defined array entries, see the IMAGE_DIRECTORY_ENTRY_XXX #defines in WINNT.H. This array allows the loader to quickly find a particular section of the image (for example, the imported function table), without needing to iterate through each of the images sections, comparing names as it goes along. Most array entries describe an entire section's data. However, the IMAGE_DIRECTORY_ENTRY_DEBUG element only encompasses a small portion of the bytes in the .rdata section.

The Section Table

Between the PE header and the raw data for the image's sections lies the section table. The section table is essentially a phone book containing information about each section in the image. The sections in the image are sorted by their starting address (RVAs), rather than alphabetically.

Now I can better clarify what a section is. In an NE file, your program's code and data are stored in distinct "segments" in the file. Part of the NE header is an array of structures, one for each segment your program uses. Each structure in the array contains information about one segment. The information stored includes the segment's type (code or data), its size, and its location elsewhere in the file.

In a PE file, the section table is analogous to the segment table in the NE file. Unlike an NE file segment table, though, a PE section table doesn't store a selector value for each code or data chunk. Instead, each section table entry stores an address where the file's raw data has been mapped into memory. While sections are analogous to 32-bit segments, they really aren't individual segments. They're just really memory ranges in a process's virtual address space.

Another area where PE files differ from NE files is how they manage the supporting data that your program doesn't use, but the operating system does; for example, the list of DLLs that the executable uses or the location of the fixup table. In an NE file, resources aren't considered segments. Even though they have selectors assigned to

them, information about resources is not stored in the NE header's segment table. Instead, resources are relegated to a separate table towards the end of the NE header. Information about imported and exported functions also doesn't warrant its own segment; it's crammed into the NE header.

The story with PE files is different. Anything that might be considered vital code or data is stored in a full-fledged section. Thus, information about imported functions is stored in its own section, as is the table of functions that the module exports. The same goes for the relocation data. Any code or data that might be needed by either the program or the operating system gets its own section.

Before I discuss specific sections, I need to describe the data that the operating system manages the sections with. Immediately following the PE header in memory is an array of IMAGE_SECTION_HEADERs. The number of elements in this array is given in the PE header (the IMAGE_NT_HEADER.FileHeader.NumberOfSections field). I used PEDUMP to output the section table and all of the section's fields and attributes. Figure 5 shows the PEDUMP output of a section table for a typical EXE file, and Figure 6 shows the section table in an OBJ file.

Table 4. A Typical Section Table from an EXE File

```
01 .text     VirtSize: 00005AFA  VirtAddr:  00001000
   raw data offs:   00000400  raw data size: 00005C00
   relocation offs: 00000000  relocations:   00000000
   line # offs:     00009220  line #'s:      0000020C
   characteristics: 60000020
     CODE  MEM_EXECUTE  MEM_READ

 02 .bss      VirtSize: 00001438  VirtAddr:  00007000
   raw data offs:   00000000  raw data size: 00001600
   relocation offs: 00000000  relocations:   00000000
   line # offs:     00000000  line #'s:      00000000
   characteristics: C0000080
     UNINITIALIZED_DATA  MEM_READ  MEM_WRITE

 03 .rdata    VirtSize: 0000015C  VirtAddr:  00009000
   raw data offs:   00006000  raw data size: 00000200
   relocation offs: 00000000  relocations:   00000000
   line # offs:     00000000  line #'s:      00000000
   characteristics: 40000040
     INITIALIZED_DATA  MEM_READ

 04 .data     VirtSize: 0000239C  VirtAddr:  0000A000
   raw data offs:   00006200  raw data size: 00002400
   relocation offs: 00000000  relocations:   00000000
   line # offs:     00000000  line #'s:      00000000
   characteristics: C0000040
     INITIALIZED_DATA  MEM_READ  MEM_WRITE

 05 .idata    VirtSize: 0000033E  VirtAddr:  0000D000
   raw data offs:   00008600  raw data size: 00000400
   relocation offs: 00000000  relocations:   00000000
   line # offs:     00000000  line #'s:      00000000
   characteristics: C0000040
     INITIALIZED_DATA  MEM_READ  MEM_WRITE

 06 .reloc    VirtSize: 000006CE  VirtAddr:  0000E000
```

```
  raw data offs:   00008A00  raw data size: 00000800
  relocation offs: 00000000  relocations:   00000000
  line # offs:     00000000  line #'s:      00000000
  characteristics: 42000040
    INITIALIZED_DATA  MEM_DISCARDABLE  MEM_READ
```

Table 5. A Typical Section Table from an OBJ File

```
01 .drectve  PhysAddr: 00000000  VirtAddr:  00000000
  raw data offs:   000000DC  raw data size: 00000026
  relocation offs: 00000000  relocations:   00000000
  line # offs:     00000000  line #'s:      00000000
  characteristics: 00100A00
    LNK_INFO  LNK_REMOVE

 02 .debug$S  PhysAddr: 00000026  VirtAddr:  00000000
  raw data offs:   00000102  raw data size: 000016D0
  relocation offs: 000017D2  relocations:   00000032
  line # offs:     00000000  line #'s:      00000000
  characteristics: 42100048
    INITIALIZED_DATA  MEM_DISCARDABLE  MEM_READ

 03 .data     PhysAddr: 000016F6  VirtAddr:  00000000
  raw data offs:   000019C6  raw data size: 00000D87
  relocation offs: 0000274D  relocations:   00000045
  line # offs:     00000000  line #'s:      00000000
  characteristics: C0400040
    INITIALIZED_DATA  MEM_READ  MEM_WRITE

 04 .text     PhysAddr: 0000247D  VirtAddr:  00000000
  raw data offs:   000029FF  raw data size: 000010DA
  relocation offs: 00003AD9  relocations:   000000E9
  line # offs:     000043F3  line #'s:      000000D9
  characteristics: 60500020
    CODE  MEM_EXECUTE  MEM_READ

 05 .debug$T  PhysAddr: 00003557  VirtAddr:  00000000
  raw data offs:   00004909  raw data size: 00000030
  relocation offs: 00000000  relocations:   00000000
  line # offs:     00000000  line #'s:      00000000
  characteristics: 42100048
    INITIALIZED_DATA  MEM_DISCARDABLE  MEM_READ
```

Each IMAGE_SECTION_HEADER has the format described in Figure 7. It's interesting to note what's missing from the information stored for each section. First off, notice that there's no indication of any PRELOAD attributes. The NE file format allows you to specify with the PRELOAD attribute which segments should be loaded at module load time. The OS/2® 2.0 LX format has something similar, allowing you to specify up to eight pages to preload. The PE format has nothing like this. Microsoft must be confident in the performance of Win32 demand-paged loading.

Table 6. IMAGE_SECTION_HEADER Formats

BYTE Name[IMAGE_SIZEOF_SHORT_NAME]	This is an 8-byte ANSI name (not UNICODE) that names the section. Most section names start with a . (such as ".text"), but this is not a requirement, as some PE documentation would have you believe. You can name your own sections with either the segment directive in assembly language, or with "#pragma data_seg" and "#pragma code_seg" in the Microsoft C/C++ compiler. It's important to note that if the section name takes up the full 8 bytes, there's no NULL terminator byte. If you're a printf devotee, you can use %.8s to avoid copying the name string to another buffer where you can NULL-terminate it. *union {* *DWORD PhysicalAddress* *DWORD VirtualSize* *} Misc;* This field has different meanings, in EXEs or OBJs. In an EXE, it holds the actual size of the code or data. This is the size before rounding up to the nearest file alignment multiple. The SizeOfRawData field (seems a bit of a misnomer) later on in the structure holds the rounded up value. The Borland linker reverses the meaning of these two fields and appears to be correct. For OBJ files, this field indicates the physical address of the section. The first section starts at address 0. To find the physical address in an OBJ file of the next section, add the SizeOfRawData value to the physical address of the current section.
DWORD VirtualAddress	In EXEs, this field holds the RVA to where the loader should map the section. To calculate the real starting address of a given section in memory, add the base address of the image to the section's VirtualAddress stored in this field. With Microsoft tools, the first section defaults to an RVA of 0x1000. In OBJs, this field is meaningless and is set to 0.
DWORD SizeOfRawData	In EXEs, this field contains the size of the section after it's been rounded up to the file alignment size. For example, assume a file alignment size of 0x200. If the VirtualSize field from above says that the section is 0x35A bytes in length, this field will say that the section is 0x400 bytes long. In OBJs, this field contains the exact size of the section emitted by the compiler or assembler. In other words, for OBJs, it's equivalent to the VirtualSize field in EXEs.
DWORD PointerToRawData	This is the file-based offset of where the raw data emitted by the compiler or assembler can be found. If your program memory maps a PE or COFF file itself (rather than letting the operating system load it), this field is more important than the VirtualAddress field. You'll have a completely linear file mapping in this situation, so you'll find the data for the sections at this offset, rather than at the RVA specified in the VirtualAddress field.
DWORD PointerToRelocations	In OBJs, this is the file-based offset to the relocation information for this section. The relocation information for each OBJ section immediately follows the raw data for that section. In EXEs, this field (and the subsequent field) are

	meaningless, and set to 0. When the linker creates the EXE, it resolves most of the fixups, leaving only base address relocations and imported functions to be resolved at load time. The information about base relocations and imported functions is kept in their own sections, so there's no need for an EXE to have per-section relocation data following the raw section data.
DWORD PointerToLinenumbers	This is the file-based offset of the line number table. A line number table correlates source file line numbers to the addresses of the code generated for a given line. In modern debug formats like the CodeView format, line number information is stored as part of the debug information. In the COFF debug format, however, the line number information is stored separately from the symbolic name/type information. Usually, only code sections (such as .text) have line numbers. In EXE files, the line numbers are collected towards the end of the file, after the raw data for the sections. In OBJ files, the line number table for a section comes after the raw section data and the relocation table for that section.
WORD NumberOfRelocations	The number of relocations in the relocation table for this section (the PointerToRelocations field from above). This field seems relevant only for OBJ files.
WORD NumberOfLinenumbers	The number of line numbers in the line number table for this section (the PointerToLinenumbers field from above).
DWORD Characteristics	What most programmers call flags, the COFF/PE format calls characteristics. This field is a set of flags that indicate the section's attributes (such as code/data, readable, or writeable,). For a complete list of all possible section attributes, see the IMAGE_SCN_XXX_XXX #defines in WINNT.H. Some of the more important flags are shown below: 0x00000020 This section contains code. Usually set in conjunction with the executable flag (0x80000000). 0x00000040 This section contains initialized data. Almost all sections except executable and the .bss section have this flag set. 0x00000080 This section contains uninitialized data (for example, the .bss section). 0x00000200 This section contains comments or some other type of information. A typical use of this section is the .drectve section emitted by the compiler, which contains commands for the linker. 0x00000800 This section's contents shouldn't be put in the final EXE file. These sections are used by the compiler/assembler to pass information to the linker. 0x02000000 This section can be discarded, since it's not needed by the process once it's been loaded. The most common discardable section is the base relocations (.reloc). 0x10000000 This section is shareable. When used with a DLL,

	the data in this section will be shared among all processes using the DLL. The default is for data sections to be nonshared, meaning that each process using a DLL gets its own copy of this section's data. In more technical terms, a shared section tells the memory manager to set the page mappings for this section such that all processes using the DLL refer to the same physical page in memory. To make a section shareable, use the SHARED attribute at link time. For example: `LINK /SECTION:MYDATA,RWS ...` tells the linker that the section called MYDATA should be readable, writeable, and shared. 0x20000000 This section is executable. This flag is usually set whenever the "contains code" flag (0x00000020) is set. 0x40000000 This section is readable. This flag is almost always set for sections in EXE files. 0x80000000 The section is writeable. If this flag isn't set in an EXE's section, the loader should mark the memory mapped pages as read-only or execute-only. Typical sections with this attribute are .data and .bss. Interestingly, the .idata section also has this attribute set.

Also missing from the PE format is the notion of page tables. The OS/2 equivalent of an IMAGE_SECTION_HEADER in the LX format doesn't point directly to where the code or data for a section can be found in the file. Instead, it refers to a page lookup table that specifies attributes and the locations of specific ranges of pages within a section. The PE format dispenses with all that, and guarantees that a section's data will be stored contiguously within the file. Of the two formats, the LX method may allow more flexibility, but the PE style is significantly simpler and easier to work with. Having written file dumpers for both formats, I can vouch for this!

Another welcome change in the PE format is that the locations of items are stored as simple DWORD offsets. In the NE format, the location of almost everything is stored as a sector value. To find the real offset, you need to first look up the alignment unit size in the NE header and convert it to a sector size (typically 16 or 512 bytes). You then need to multiply the sector size by the specified sector offset to get an actual file offset. If by chance something isn't stored as a sector offset in an NE file, it is probably stored as an offset relative to the NE header. Since the NE header isn't at the beginning of the file, you need to drag around the file offset of the NE header in your code.

All in all, the PE format is much easier to work with than the NE, LX, or LE formats *(assuming you can use memory-mapped files)*.

Common Sections

Having seen what sections are in general and where they're located, let's look at the common sections that you'll find in EXE and OBJ files. The list is by no means complete, but includes the sections you encounter every day (even if you're not aware of it).

The .text section is where all general-purpose code emitted by the compiler or assembler ends up. Since PE files

run in 32-bit mode and aren't restricted to 16-bit segments, there's no reason to break the code from separate source files into separate sections. Instead, the linker concatenates all the .text sections from the various OBJs into one big .text section in the EXE. If you use Borland C++ the compiler emits its code to a segment named CODE. PE files produced with Borland C++ have a section named CODE rather than one called .text. I'll explain this in a minute.

It was somewhat interesting to me to find out that there was additional code in the .text section beyond what I created with the compiler or used from the run-time libraries. In a PE file, when you call a function in another module (for example, GetMessage in USER32.DLL), the CALL instruction emitted by the compiler doesn't transfer control directly to the function in the DLL (see Figure 8). Instead, the call instruction transfers control to a

```
JMP DWORD PTR [XXXXXXXX]
```

instruction that's also in the .text section. The JMP instruction indirects through a DWORD variable in the .idata section. This .idata section DWORD contains the real address of the operating system function entry point. After thinking about this for a while, I came to understand why DLL calls are implemented this way. By funneling all calls to a given DLL function through one location, the loader doesn't need to patch every instruction that calls a DLL.

All the PE loader has to do is put the correct address of the target function into the DWORD in the .idata section. No call instructions need to be patched. This is in marked contrast to NE files, where each segment contains a list of fixups that need to be applied to the segment. If the segment calls a given DLL function 20 times, the loader must write the address of that function 20 times into the segment. The downside to the PE method is that you can't initialize a variable with the true address of a DLL function.

For example, you would think that something like

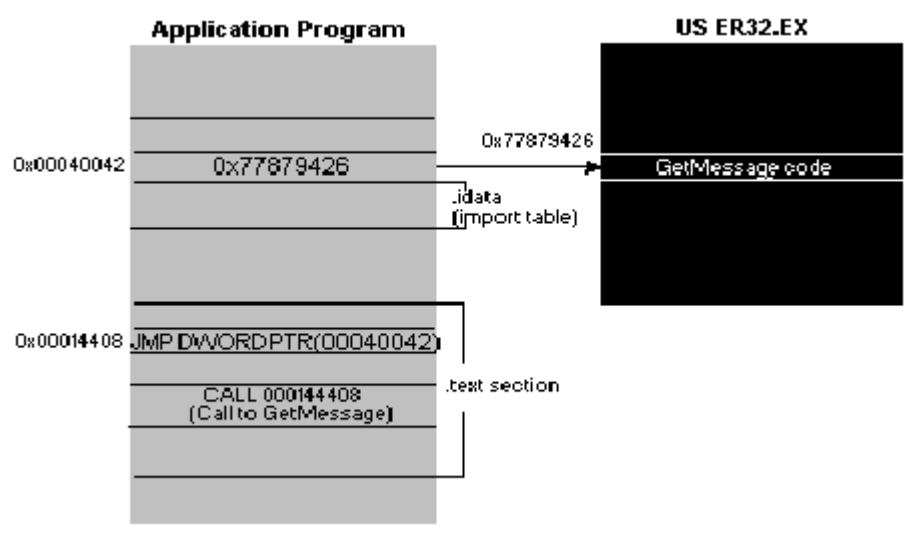

Figure 2. Calling a function in another module

```
FARPROC pfnGetMessage = GetMessage;
```

would put the address of GetMessage into the variable pfnGetMessage. In 16-bit Windows, this works, while in Win32 it doesn't. In Win32, the variable pfnGetMessage will end up holding the address of the JMP DWORD PTR [XXXXXXXX] thunk that I mentioned earlier. If you wanted to call through the function pointer, things would work as you'd expect. However, if you want to read the bytes at the beginning of GetMessage, you're out of

luck (unless you do additional work to follow the .idata "pointer" yourself). I'll come back to this topic later, in the discussion of the import table.

Although Borland could have had the compiler emit segments with a name of .text, it chose a default segment name of CODE. To determine a section name in the PE file, the Borland linker (TLINK32.EXE) takes the segment name from the OBJ file and truncates it to 8 characters (if necessary).

While the difference in the section names is a small matter, there is a more important difference in how Borland PE files link to other modules. As I mentioned in the .text description, all calls to OBJs go through a JMP DWORD PTR [XXXXXXXX] thunk. Under the Microsoft system, this thunk comes to the EXE from the .text section of an import library. Because the library manager (LIB32) creates the import library (and the thunk) when you link the external DLL, the linker doesn't have to "know" how to generate these thunks itself. The import library is really just some more code and data to link into the PE file.

The Borland system of dealing with imported functions is simply an extension of the way things were done for 16-bit NE files. The import libraries that the Borland linker uses are really just a list of function names along with the name of the DLL they're in. TLINK32 is therefore responsible for determining which fixups are to external DLLs, and generating an appropriate JMP DWORD PTR [XXXXXXXX] thunk for it. TLINK32 stores the thunks that it creates in a section named .icode.

Just as .text is the default section for code, the .data section is where your initialized data goes. This data consists of global and static variables that are initialized at compile time. It also includes string literals. The linker combines all the .data sections from the OBJ and LIB files into one .data section in the EXE. Local variables are located on a thread's stack, and take no room in the .data or .bss sections.

The .bss section is where any uninitialized static and global variables are stored. The linker combines all the .bss sections in the OBJ and LIB files into one .bss section in the EXE. In the section table, the RawDataOffset field for the .bss section is set to 0, indicating that this section doesn't take up any space in the file. TLINK doesn't emit this section. Instead it extends the virtual size of the DATA section.

.CRT is another initialized data section utilized by the Microsoft C/C++ run-time libraries (hence the name). Why this data couldn't go into the standard .data section is beyond me.

The .rsrc section contains all the resources for the module. In the early days of Windows NT, the RES file output of the 16-bit RC.EXE wasn't in a format that the Microsoft PE linker could understand. The CVTRES program converted these RES files into a COFF-format OBJ, placing the resource data into a .rsrc section within the OBJ. The linker could then treat the resource OBJ as just another OBJ to link in, allowing the linker to not "know" anything special about resources. More recent linkers from Microsoft appear to be able to process RES files directly.

The .idata section contains information about functions (and data) that the module imports from other DLLs. This section is equivalent to an NE file's module reference table. A key difference is that each function that a PE file imports is specifically listed in this section. To find the equivalent information in an NE file, you'd have to go digging through the relocations at the end of the raw data for each of the segments.

The .edata section is a list of the functions and data that the PE file exports for other modules. Its NE file equivalent is the combination of the entry table, the resident names table, and the nonresident names table. Unlike in 16-bit Windows, there's seldom a reason to export anything from an EXE file, so you usually only see .edata sections in DLLs. When using Microsoft tools, the data in the .edata section comes to the PE file via the EXP file. Put another way, the linker doesn't generate this information on its own. Instead, it relies on the library manager

(LIB32) to scan the OBJ files and create the EXP file that the linker adds to its list of modules to link. Yes, that's right! Those pesky EXP files are really just OBJ files with a different extension.

The .reloc section holds a table of base relocations. A base relocation is an adjustment to an instruction or initialized variable value that's needed if the loader couldn't load the file where the linker assumed it would. If the loader is able to load the image at the linker's preferred base address, the loader completely ignores the relocation information in this section. If you want to take a chance and hope that the loader can always load the image at the assumed base address, you can tell the linker to strip this information with the /FIXED option. While this may save space in the executable file, it may cause the executable not to work on other Win32-based implementations. For example, say you built an EXE for Windows NT and based the EXE at 0x10000. If you told the linker to strip the relocations, the EXE wouldn't run under Windows 95, where the address 0x10000 is already in use.

It's important to note that the JMP and CALL instructions that the compiler generates use offsets relative to the instruction, rather than actual offsets in the 32-bit flat segment. If the image needs to be loaded somewhere other than where the linker assumed for a base address, these instructions don't need to change, since they use relative addressing. As a result, there are not as many relocations as you might think. Relocations are usually only needed for instructions that use a 32-bit offset to some data. For example, let's say you had the following global variable declarations:

```
int i;
int *ptr = &i;
```

If the linker assumed an image base of 0x10000, the address of the variable i will end up containing something like 0x12004. At the memory used to hold the pointer "ptr", the linker will have written out 0x12004, since that's the address of the variable i. If the loader for whatever reason decided to load the file at a base address of 0x70000, the address of i would be 0x72004. The .reloc section is a list of places in the image where the difference between the linker assumed load address and the actual load address needs to be factored in.

When you use the compiler directive _ _declspec(thread), the data that you define doesn't go into either the .data or .bss sections. It ends up in the .tls section, which refers to "thread local storage," and is related to the TlsAlloc family of Win32 functions. When dealing with a .tls section, the memory manager sets up the page tables so that whenever a process switches threads, a new set of physical memory pages is mapped to the .tls section's address space. This permits per-thread global variables. In most cases, it is much easier to use this mechanism than to allocate memory on a per-thread basis and store its pointer in a TlsAlloc'ed slot.

There's one unfortunate note that must be added about the .tls section and _ _declspec(thread) variables. In Windows NT and Windows 95, this thread local storage mechanism won't work in a DLL if the DLL is loaded dynamically by LoadLibrary. In an EXE or an implicitly loaded DLL, everything works fine. If you can't implicitly link to the DLL, but need per-thread data, you'll have to fall back to using TlsAlloc and TlsGetValue with dynamically allocated memory.

Although the .rdata section usually falls between the .data and .bss sections, your program generally doesn't see or use the data in this section. The .rdata section is used for at least two things. First, in Microsoft linker-produced EXEs, the .rdata section holds the debug directory, which is only present in EXE files. *(In TLINK32 EXEs, the debug directory is in a section named .debug.)*

The debug directory is an array of IMAGE_DEBUG_DIRECTORY structures. These structures hold information about the type, size, and location of the various types of debug information stored in the file. Three main types of debug information appear: CodeView®, COFF, and FPO. Figure 9 shows the PEDUMP output for a typical debug

directory.

Table 7. A Typical Debug Directory

Type	Size	Address	FilePtr	Charactr	TimeData	Version	
COFF	000065C5	00000000	00009200	00000000	2CF8CF3D		0.00
???	00000114	00000000	0000F7C8	00000000	2CF8CF3D		0.00
FPO	000004B0	00000000	0000F8DC	00000000	2CF8CF3D		0.00
CODEVIEW	0000B0B4	00000000	0000FD8C	00000000	2CF8CF3D		0.00

The debug directory isn't necessarily found at the beginning of the .rdata section. To find the start of the debug directory table, use the RVA in the seventh entry *(IMAGE_DIRECTORY_ENTRY_DEBUG)* of the data directory. The data directory is at the end of the PE header portion of the file. To determine the number of entries in the Microsoft linker-generated debug directory, divide the size of the debug directory (found in the size field of the data directory entry) by the size of an IMAGE_DEBUG_DIRECTORY structure. TLINK32 emits a simple count, usually 1. The PEDUMP sample program demonstrates this.

The other useful portion of an .rdata section is the description string. If you specified a DESCRIPTION entry in your program's DEF file, the specified description string appears in the .rdata section. In the NE format, the description string is always the first entry of the nonresident names table. The description string is intended to hold a useful text string describing the file. Unfortunately, I haven't found an easy way to find it. I've seen PE files that had the description string before the debug directory, and other files that had it after the debug directory. I'm not aware of any consistent method of finding the description string *(or even if it's present at all).*

These .debug$S and .debug$T sections only appear in OBJs. They store the CodeView symbol and type information. The section names are derived from the segment names used for this purpose by previous 16-bit compilers ($$SYMBOLS and $$TYPES). The sole purpose of the .debug$T section is to hold the pathname to the PDB file that contains the CodeView information for all the OBJs in the project. The linker reads in the PDB and uses it to create portions of the CodeView information that it places at the end of the finished PE file.

The .drective section only appears in OBJ files. It contains text representations of commands for the linker. For example, in any OBJ I compile with the Microsoft compiler, the following strings appear in the .drectve section:

```
-defaultlib:LIBC -defaultlib:OLDNAMES
```

When you use _ _declspec(export) in your code, the compiler simply emits the command-line equivalent into the .drectve section *(for instance, "-export:MyFunction").*

In playing around with PEDUMP, I've encountered other sections from time to time. For instance, in the Windows 95 KERNEL32.DLL, there are LOCKCODE and LOCKDATA sections. Presumably these are sections that will get special paging treatment so that they're never paged out of memory.

There are two lessons to be learned from this. First, don't feel constrained to use only the standard sections provided by the compiler or assembler. If you need a separate section for some reason, don't hesitate to create your own. In the C/C++ compiler, use the #pragma code_seg and #pragma data_seg.

In assembly language, just create a 32-bit segment (which becomes a section) with a name different from the standard sections. If using TLINK32, you must use a different class or turn off code segment packing. The other

thing to remember is that section names that are out of the ordinary can often give a deeper insight into the purpose and implementation of a particular PE file.

PE File Imports

Earlier, I described how function calls to outside DLLs don't call the DLL directly. Instead, the CALL instruction goes to a JMP DWORD PTR [XXXXXXXX] instruction somewhere in the executable's .text section (or .icode section if you're using Borland C++).

The address that the JMP instruction looks up and transfers control to is the real target address. The PE file's .idata section contains the information necessary for the loader to determine the addresses of the target functions and patch them into the executable image.

The .idata section *(or import table, as I prefer to call it)* begins with an array of IMAGE_IMPORT_DESCRIPTORs. There is one IMAGE_IMPORT_DESCRIPTOR for each DLL that the PE file implicitly links to. There's no field indicating the number of structures in this array. Instead, the last element of the array is indicated by an IMAGE_IMPORT_DESCRIPTOR that has fields filled with NULLs.

The format of an IMAGE_IMPORT_DESCRIPTOR is shown in Figure 10.

Table 8. IMAGE_IMPORT_DESCRIPTOR Format

DWORD Characteristics	At one time, this may have been a set of flags. However, Microsoft changed its meaning and never bothered to update WINNT.H. This field is really an offset (an RVA) to an array of pointers. Each of these pointers points to an IMAGE_IMPORT_BY_NAME structure.
DWORD TimeDateStamp	The time/date stamp indicating when the file was built.
DWORD ForwarderChain	This field relates to forwarding. Forwarding involves one DLL sending on references to one of its functions to another DLL. For example, in Windows NT, NTDLL.DLL appears to forward some of its exported functions to KERNEL32.DLL. An application may think it's calling a function in NTDLL.DLL, but it actually ends up calling into KERNEL32.DLL. This field contains an index into FirstThunk array (described momentarily). The function indexed by this field will be forwarded to another DLL. Unfortunately, the format of how a function is forwarded isn't documented, and examples of forwarded functions are hard to find.
DWORD Name	This is an RVA to a NULL-terminated ASCII string containing the imported DLL's name. Common examples are "KERNEL32.DLL" and "USER32.DLL".
PIMAGE_THUNK_DATA FirstThunk	This field is an offset (an RVA) to an IMAGE_THUNK_DATA union. In almost every case, the union is interpreted as a pointer to an IMAGE_IMPORT_BY_NAME structure. If the field isn't one of these pointers, then it's supposedly treated as an export ordinal value for the DLL that's being imported. It's not clear from the documentation if you really can import a function by ordinal rather than by name.

The important parts of an IMAGE_IMPORT_DESCRIPTOR are the imported DLL name and the two arrays of IMAGE_IMPORT_BY_NAME pointers. In the EXE file, the two arrays (pointed to by the Characteristics and FirstThunk fields) run parallel to each other, and are terminated by a NULL pointer entry at the end of each array. The pointers in both arrays point to an IMAGE_IMPORT_BY_NAME structure. Figure 11 shows the situation graphically. Figure 12 shows the PEDUMP output for an imports table.

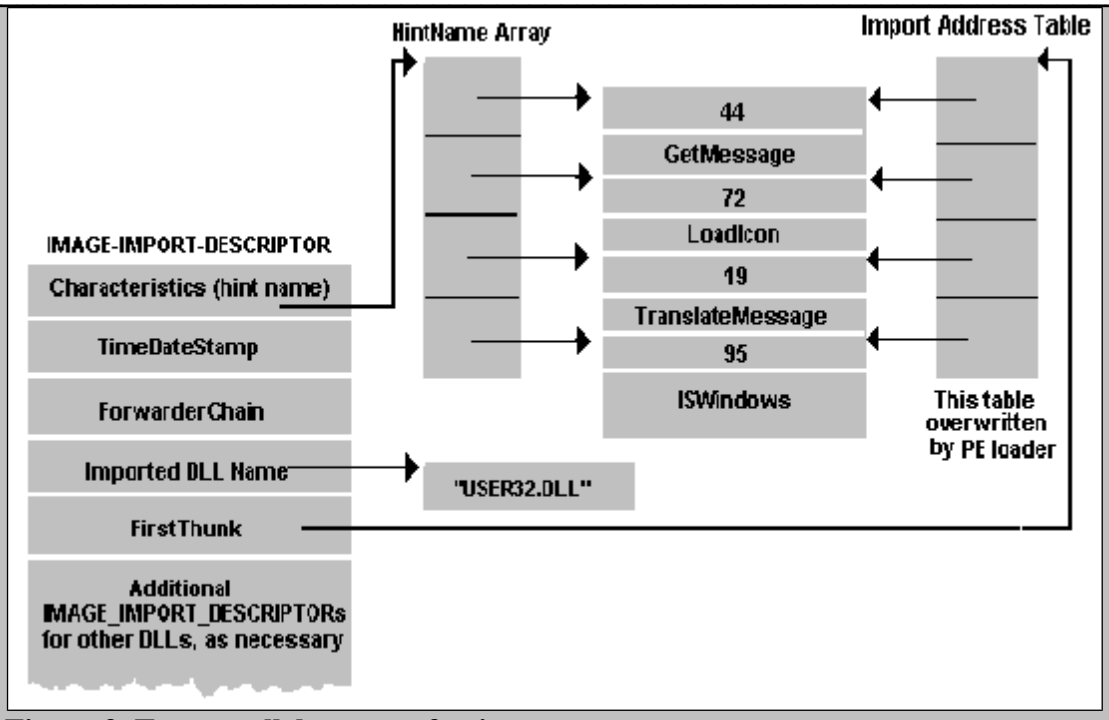

Figure 3. Two parallel arrays of pointers

Table 9. Imports Table from an EXE File

```
GDI32.dll
 Hint/Name Table: 00013064
 TimeDateStamp:   2C51B75B
 ForwarderChain:  FFFFFFFF
 First thunk RVA: 00013214
 Ordn  Name
   48  CreatePen
   57  CreateSolidBrush
   62  DeleteObject
  160  GetDeviceCaps
   //  Rest of table omitted...

 KERNEL32.dll
 Hint/Name Table: 0001309C
 TimeDateStamp:   2C4865A0
 ForwarderChain:  00000014
 First thunk RVA: 0001324C
 Ordn  Name
   83  ExitProcess
  137  GetCommandLineA
  179  GetEnvironmentStrings
  202  GetModuleHandleA
   //  Rest of table omitted...

 SHELL32.dll
 Hint/Name Table: 00013138
```

```
TimeDateStamp:   2C41A383
ForwarderChain:  FFFFFFFF
First thunk RVA: 000132E8
Ordn  Name
  46  ShellAboutA

USER32.dll
Hint/Name Table: 00013140
TimeDateStamp:   2C474EDF
ForwarderChain:  FFFFFFFF
First thunk RVA: 000132F0
Ordn  Name
  10  BeginPaint
  35  CharUpperA
  39  CheckDlgButton
  40  CheckMenuItem

  //  Rest of table omitted...
```

There is one IMAGE_IMPORT_BY_NAME structure for each function that the PE file imports. An IMAGE_IMPORT_BY_NAME structure is very simple, and looks like this:

```
WORD    Hint;
BYTE    Name[?];
```

The first field is the best guess as to what the export ordinal for the imported function is. Unlike with NE files, this value doesn't have to be correct. Instead, the loader uses it as a suggested starting value for its binary search for the exported function. Next is an ASCIIZ string with the name of the imported function.

Why are there two parallel arrays of pointers to the IMAGE_IMPORT_BY_NAME structures? The first array (the one pointed at by the Characteristics field) is left alone, and never modified. It's sometimes called the hint-name table. The second array (pointed at by the FirstThunk field) is overwritten by the PE loader.

The loader iterates through each pointer in the array and finds the address of the function that each IMAGE_IMPORT_BY_NAME structure refers to. The loader then overwrites the pointer to IMAGE_IMPORT_BY_NAME with the found function's address. The [XXXXXXXX] portion of the JMP DWORD PTR [XXXXXXXX] thunk refers to one of the entries in the FirstThunk array. Since the array of pointers that's overwritten by the loader eventually holds the addresses of all the imported functions, it's called the Import Address Table.

For you Borland users, there's a slight twist to the above description. A PE file produced by TLINK32 is missing one of the arrays. In such an executable, the Characteristics field in the IMAGE_IMPORT_DESCRIPTOR (aka the hint-name array) is 0.

Therefore, only the array that's pointed at by the FirstThunk field (the Import Address Table) is guaranteed to exist in all PE files. The story would end here, except that I ran into an interesting problem when writing PEDUMP. In the never ending search for optimizations, Microsoft “optimized” the thunk array in the system DLLs for Windows NT *(KERNEL32.DLL and so on)*.

In this optimization, the pointers in the array don't point to an IMAGE_IMPORT_BY_NAME structure—rather,

they already contain the address of the imported function. In other words, the loader doesn't need to look up function addresses and overwrite the thunk array with the imported function's addresses.

This causes a problem for PE dumping programs that are expecting the array to contain pointers to IMAGE_IMPORT_BY_NAME structures. You might be thinking, *"But Matt, why don't you just use the hint-name table array?"* That would be an ideal solution, except that the hint-name table array doesn't exist in Borland files. The PEDUMP program handles all these situations, but the code is understandably messy.

Since the import address table is in a writeable section, it's relatively easy to intercept calls that an EXE or DLL makes to another DLL. Simply patch the appropriate import address table entry to point at the desired interception function. There's no need to modify any code in either the caller or callee images. What could be easier?

It's interesting to note that in Microsoft-produced PE files, the import table is not something wholly synthesized by the linker. All the pieces necessary to call a function in another DLL reside in an import library.

When you link a DLL, the library manager (LIB32.EXE or LIB.EXE) scans the OBJ files being linked and creates an import library. This import library is completely different from the import libraries used by 16-bit NE file linkers. The import library that the 32-bit LIB produces has a .text section and several .idata$ sections. The .text section in the import library contains the JMP DWORD PTR [XXXXXXXX] thunk, which has a name stored for it in the OBJ's symbol table. The name of the symbol is identical to the name of the function being exported by the DLL (for example, _Dispatch_Message@4).

One of the .idata$ sections in the import library contains the DWORD that the thunk dereferences through. Another of the .idata$ sections has a space for the hint ordinal followed by the imported function's name. These two fields make up an IMAGE_IMPORT_BY_NAME structure.

When you later link a PE file that uses the import library, the import library's sections are added to the list of sections from your OBJs that the linker needs to process. Since the thunk in the import library has the same name as the function being imported, the linker assumes the thunk is really the imported function, and fixes up calls to the imported function to point at the thunk. The thunk in the import library is essentially "seen" as the imported function.

Besides providing the code portion of an imported function thunk, the import library provides the pieces of the PE file's .idata section (or import table). These pieces come from the various .idata$ sections that the library manager put into the import library. In short, the linker doesn't really know the differences between imported functions and functions that appear in a different OBJ file. The linker just follows its preset rules for building and combining sections, and everything falls into place naturally.

PE File Exports

The opposite of importing a function is exporting a function for use by EXEs or other DLLs. A PE file stores information about its exported functions in the .edata section. Generally, Microsoft linker-generated PE EXE files don't export anything, so they don't have an .edata section. Borland's TLINK32 always exports at least one symbol from an EXE. Most DLLs do export functions and have an .edata section. The primary components of an .edata section (aka the export table) are tables of function names, entry point addresses, and export ordinal values. In an NE file, the equivalents of an export table are the entry table, the resident names table, and the nonresident names table. These tables are stored as part of the NE header, rather than in distinct segments or resources.

At the start of an .edata section is an IMAGE_EXPORT_DIRECTORY structure (see Table 10). This structure is

immediately followed by data pointed to by fields in the structure.

Table 10. IMAGE_EXPORT_DIRECTORY Format

DWORD Characteristics	This field appears to be unused and is always set to 0.
DWORD TimeDateStamp	The time/date stamp indicating when this file was created.
WORD MajorVersion	These fields appear to be unused and are set to 0.
WORD MinorVersion	
DWORD Name	The RVA of an ASCIIZ string with the name of this DLL.
DWORD Base	The starting ordinal number for exported functions. For example, if the file exports functions with ordinal values of 10, 11, and 12, this field contains 10. To obtain the exported ordinal for a function, you need to add this value to the appropriate element of the AddressOfNameOrdinals array.
DWORD NumberOfFunctions	The number of elements in the AddressOfFunctions array. This value is also the number of functions exported by this module. Theoretically, this value could be different than the NumberOfNames field (next), but actually they're always the same.
DWORD NumberOfNames	The number of elements in the AddressOfNames array. This value seems always to be identical to the NumberOfFunctions field, and so is the number of exported functions.
PDWORD *AddressOfFunctions	This field is an RVA and points to an array of function addresses. The function addresses are the entry points (RVAs) for each exported function in this module.
PDWORD *AddressOfNames	This field is an RVA and points to an array of string pointers. The strings are the names of the exported functions in this module.
PWORD *AddressOfNameOrdinals	This field is an RVA and points to an array of WORDs. The WORDs are the export ordinals of all the exported functions in this module. However, don't forget to add in the starting ordinal number specified in the Base field.

The layout of the export table is somewhat odd (see Figure 4 and Table 10). As I mentioned earlier, the requirements for exporting a function are a name, an address, and an export ordinal. You'd think that the designers of the PE format would have put all three of these items into a structure, and then have an array of these structures. Instead, each component of an exported entry is an element in an array. There are three of these arrays (AddressOfFunctions, AddressOfNames, AddressOfNameOrdinals), and they are all parallel to one another. To find all the information about the fourth function, you need to look up the fourth element in each array.

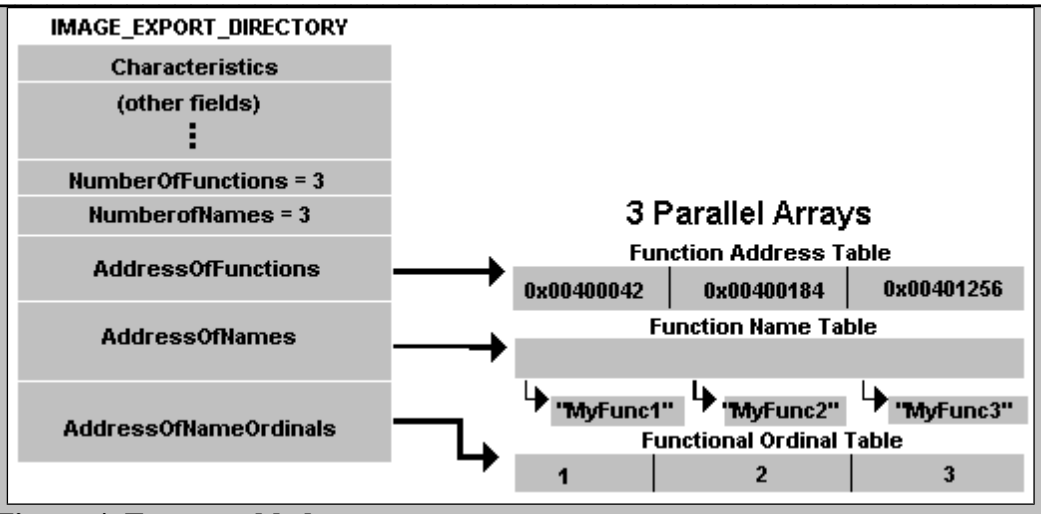

Figure 4. Export table layout

Table 11. Typical Exports Table from an EXE File

```
Name:           KERNEL32.dll
 Characteristics: 00000000
 TimeDateStamp:   2C4857D3
 Version:         0.00
 Ordinal base:    00000001
 # of functions:  0000021F
 # of Names:      0000021F

 Entry Pt  Ordn  Name
 00005090     1  AddAtomA
 00005100     2  AddAtomW
 00025540     3  AddConsoleAliasA
 00025500     4  AddConsoleAliasW
 00026AC0     5  AllocConsole
 00001000     6  BackupRead
 00001E90     7  BackupSeek
 00002100     8  BackupWrite
 0002520C     9  BaseAttachCompleteThunk
 00024C50    10  BasepDebugDump
 // Rest of table omitted...
```

Incidentally, if you dump out the exports from the Windows NT system DLLs (for example, KERNEL32.DLL and USER32.DLL), you'll note that in many cases there are two functions that only differ by one character at the end of the name, for instance CreateWindowExA and CreateWindowExW.

This is how UNICODE support is implemented transparently. The functions that end with A are the ASCII (or ANSI) compatible functions, while those ending in W are the UNICODE version of the function. In your code, you don't explicitly specify which function to call. Instead, the appropriate function is selected in WINDOWS.H, via preprocessor #ifdefs.

This excerpt from the Windows NT WINDOWS.H shows an example of how this works:

```
#ifdef UNICODE
    #define DefWindowProc  DefWindowProcW
#else
    #define DefWindowProc  DefWindowProcA
#endif // !UNICODE
```

PE File Resources

Finding resources in a PE file is quite a bit more complicated than in an NE file. The formats of the individual resources (for example, a menu) haven't changed significantly but you need to traverse a strange hierarchy to find them.

Navigating the resource directory hierarchy is like navigating a hard disk. There's a master directory (the root directory), which has subdirectories. The subdirectories have subdirectories of their own that may point to the raw resource data for things like dialog templates.

In the PE format, both the root directory of the resource directory hierarchy and all of its subdirectories are structures of type IMAGE_RESOURCE_DIRECTORY (see Table 12).

Table 12. IMAGE_RESOURCE_DIRECTORY Format

DWORD Characteristics	Theoretically this field could hold flags for the resource, but appears to always be 0.
DWORD TimeDateStamp	The time/date stamp describing the creation time of the resource.
WORD MajorVersion	Theoretically these fields would hold a version number for the resource. These field appear to always be set to 0.
WORD MinorVersion	The number of array elements that use names and that follow this structure. The number of array elements that use names and that follow this structure.
WORD NumberOfNamedEntries	
WORD NumberOfIdEntries	The number of array elements that use integer IDs, and which follow this structure.
IMAGE_RESOURCE_DIRECTORY_ENTRY DirectoryEntries[]	This field isn't really part of the IMAGE_RESOURCE_DIRECTORY structure. Rather, it's an array of IMAGE_RESOURCE_DIRECTORY_ENTRY structures that immediately follow the IMAGE_RESOURCE_DIRECTORY structure. The number of elements in the array is the sum of the NumberOfNamedEntries and NumberOfIdEntries fields. The directory entry elements that have name identifiers (rather than integer IDs) come first in the array.

A directory entry can either point at a subdirectory (that is, to another IMAGE_RESOURCE_DIRECTORY), or it can point to the raw data for a resource. Generally, there are at least three directory levels before you get to the actual raw resource data. The top-level directory (of which there's only one) is always found at the beginning of

the resource section (.rsrc). The subdirectories of the top-level directory correspond to the various types of resources found in the file.

For example, if a PE file includes dialogs, string tables, and menus, there will be three subdirectories: a dialog directory, a string table directory, and a menu directory. Each of these type subdirectories will in turn have ID subdirectories. There will be one ID subdirectory for each instance of a given resource type. In the above example, if there are three dialog boxes, the dialog directory will have three ID subdirectories.

Each ID subdirectory will have either a string name *(such as "MyDialog")* or the integer ID used to identify the resource in the RC file. Figure 5 shows a resource directory hierarchy example in visual form. Table 13 shows the PEDUMP output for the resources in the Windows NT CLOCK.EXE.

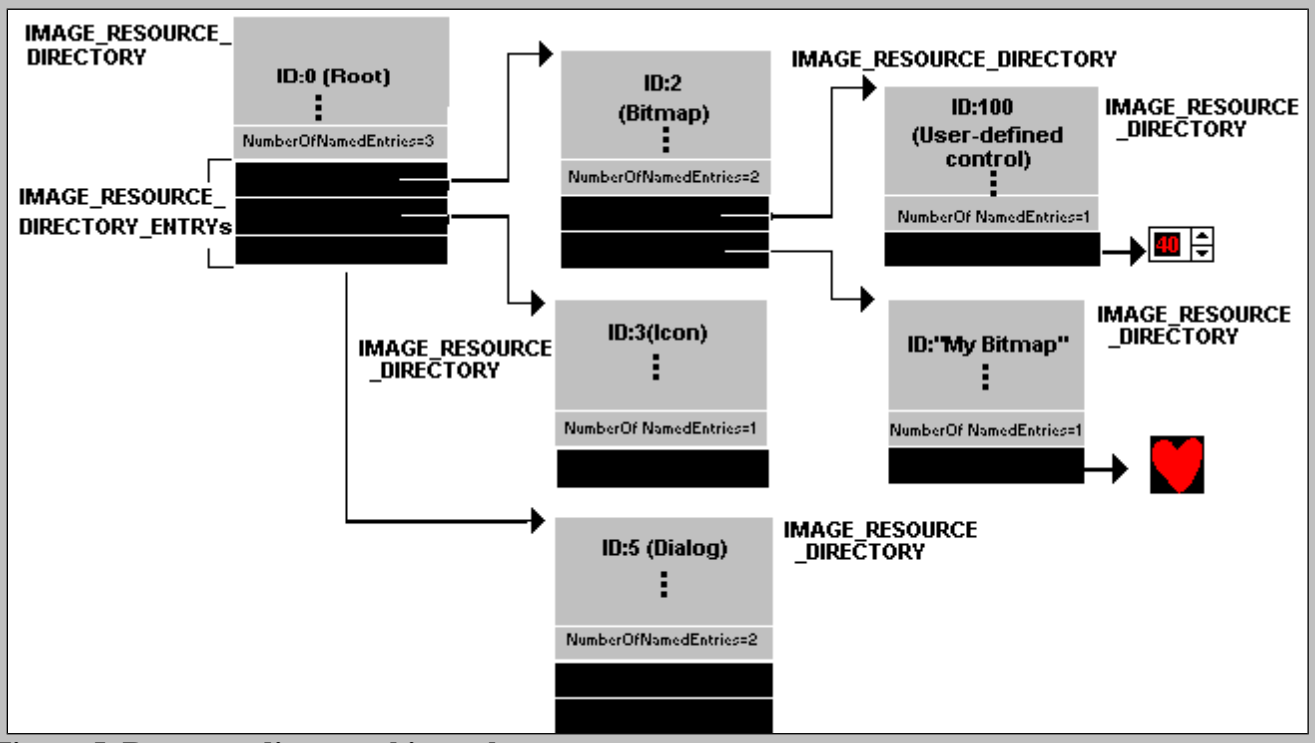

Figure 5. Resource directory hierarchy

Table 13. Resources Hierarchy for CLOCK.EXE

- ResDir (0) Named:00 ID:06 TimeDate:2C3601DB Vers:0.00 Char:0
  - ResDir (ICON) Named:00 ID:02 TimeDate:2C3601DB Vers:0.00 Char:0
    - ResDir (1) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
      - ID: 00000409 Offset: 00000200
    - ResDir (2) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
      - ID: 00000409 Offset: 00000210
  - ResDir (MENU) Named:02 ID:00 TimeDate:2C3601DB Vers:0.00 Char:0
    - ResDir (CLOCK) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
      - ID: 00000409 Offset: 00000220
    - ResDir (GENERICMENU) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
      - ID: 00000409 Offset: 00000230
  - ResDir (DIALOG) Named:01 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0

- ResDir (ABOUTBOX) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
  - ID: 00000409 Offset: 00000240
- ResDir (64) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
  - ID: 00000409 Offset: 00000250
- ResDir (STRING) Named:00 ID:03 TimeDate:2C3601DB Vers:0.00 Char:0
- ResDir (1) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
  - ID: 00000409 Offset: 00000260
- ResDir (2) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
  - ID: 00000409 Offset: 00000270
- ResDir (3) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
  - ID: 00000409 Offset: 00000280
- ResDir (GROUP_ICON) Named:01 ID:00 TimeDate:2C3601DB Vers:0.00 Char:0
- ResDir (CCKK) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
  - ID: 00000409 Offset: 00000290
- ResDir (VERSION) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
- ResDir (1) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
  - ID: 00000409 Offset: 000002A0

As mentioned earlier, each directory entry is a structure of type IMAGE_RESOURCE_DIRECTORY_ENTRY *(boy, these names are getting long!)*. Each IMAGE_RESOURCE_DIRECTORY_ENTRY has the format shown in Table 13.

Table 14. IMAGE_RESOURCE_DIRECTORY_ENTRY Format

DWORD Name	This field contains either an integer ID or a pointer to a structure that contains a string name. If the high bit (0x80000000) is zero, this field is interpreted as an integer ID. If the high bit is nonzero, the lower 31 bits are an offset (relative to the start of the resources) to an IMAGE_RESOURCE_DIR_STRING_U structure. This structure contains a WORD character count, followed by a UNICODE string with the resource name. Yes, even PE files intended for non-UNICODE Win32 implementations use UNICODE here. To convert the UNICODE string to an ANSI string, use the WideCharToMultiByte function.
DWORD OffsetToData	This field is either an offset to another resource directory or a pointer to information about a specific resource instance. If the high bit (0x80000000) is set, this directory entry refers to a subdirectory. The lower 31 bits are an offset (relative to the start of the resources) to another IMAGE_RESOURCE_DIRECTORY. If the high bit isn't set, the lower 31 bits point to an IMAGE_RESOURCE_DATA_ENTRY structure. The IMAGE_RESOURCE_DATA_ENTRY structure contains the location of the resource's raw data, its size, and its code page.

To go further into the resource formats, I'd need to discuss the format of each resource type (dialogs, menus, and so on). Covering these topics could easily fill up an entire article on its own.

PE File Base Relocations

When the linker creates an EXE file, it makes an assumption about where the file will be mapped into memory. Based on this, the linker puts the real addresses of code and data items into the executable file. If for whatever reason the executable ends up being loaded somewhere else in the virtual address space, the addresses the linker plugged into the image are wrong.

The information stored in the .reloc section allows the PE loader to fix these addresses in the loaded image so that they're correct again. On the other hand, if the loader was able to load the file at the base address assumed by the linker, the .reloc section data isn't needed and is ignored. The entries in the .reloc section are called base relocations since their use depends on the base address of the loaded image.

Unlike relocations in the NE file format, base relocations are extremely simple. They boil down to a list of locations in the image that need a value added to them. The format of the base relocation data is somewhat quirky.

The base relocation entries are packaged in a series of variable length chunks. Each chunk describes the relocations for one 4KB page in the image. Let's look at an example to see how base relocations work. An executable file is linked assuming a base address of 0x10000.
At offset 0x2134 within the image is a pointer containing the address of a string.

The string starts at physical address 0x14002, so the pointer contains the value 0x14002. You then load the file, but the loader decides that it needs to map the image starting at physical address 0x60000. The difference between the linker-assumed base load address and the actual load address is called the delta. In this case, the delta is 0x50000. Since the entire image is 0x50000 bytes higher in memory, so is the string (now at address 0x64002). The pointer to the string is now incorrect.

The executable file contains a base relocation for the memory location where the pointer to the string resides. To resolve a base relocation, the loader adds the delta value to the original value at the base relocation address. In this case, the loader would add 0x50000 to the original pointer value (0x14002), and store the result (0x64002) back into the pointer's memory. Since the string really is at 0x64002, everything is fine with the world.

Each chunk of base relocation data begins with an IMAGE_BASE_RELOCATION structure that looks like Table 14. Table 15 shows some base relocations as shown by PEDUMP. Note that the RVA values shown have already been displaced by the VirtualAddress in the IMAGE_BASE_RELOCATION field.

Figure 15. IMAGE_BASE_RELOCATION Format

DWORD VirtualAddress:	This field contains the starting RVA for this chunk of relocations. The offset of each relocation that follows is added to this value to form the actual RVA where the relocation needs to be applied.
DWORD SizeOfBlock:	The size of this structure plus all the WORD relocations that follow. To determine the number of relocations in this block, subtract the size of an IMAGE_BASE_RELOCATION (8 bytes) from the value of this field, and then divide by 2 (the size of a WORD). For example, if this field contains 44, there are 18 relocations that immediately follow: (44 - sizeof(IMAGE_BASE_RELOCATION)) / sizeof(WORD) = 18 WORD TypeOffset This isn't just a single WORD, but rather an array of WORDs, the number of which is calculated by the above formula. The bottom 12 bits of each WORD are a relocation offset, and need to be added to the value of the Virtual Address field from this relocation block's header. The high 4 bits of each WORD are a relocation type. For PE files that run on Intel CPUs, you'll only see two types of relocations:

	0	IMAGE_REL_BASED_ABSOLUTE	This relocation is meaningless and is only used as a place holder to round relocation blocks up to a DWORD multiple size.
	3	IMAGE_REL_BASED_HIGHLOW	This relocation means add both the high and low 16 bits of the delta to the DWORD specified by the calculated RVA.

Table 16. The Base Relocations from an EXE File

```
Virtual Address: 00001000  size: 0000012C
  00001032 HIGHLOW
  0000106D HIGHLOW
  000010AF HIGHLOW
  000010C5 HIGHLOW
  // Rest of chunk omitted...
Virtual Address: 00002000  size: 0000009C
  000020A6 HIGHLOW
  00002110 HIGHLOW
  00002136 HIGHLOW
  00002156 HIGHLOW
  // Rest of chunk omitted...
Virtual Address: 00003000  size: 00000114
  0000300A HIGHLOW
  0000301E HIGHLOW
  0000303B HIGHLOW
  0000306A HIGHLOW
  // Rest of relocations omitted...
```

Differences Between PE and COFF OBJ Files

There are two portions of the PE file that are not used by the operating system. These are the COFF symbol table and the COFF debug information. Why would anyone need COFF debug information when the much more complete CodeView information is available?

If you intend to use the Windows NT system debugger (NTSD) or the Windows NT kernel debugger (KD), COFF is the only game in town. For those of you who are interested, I've included a detailed description of these parts of the PE file in the online posting that accompanies this article (available on all MSJ bulletin boards).

At many points throughout the preceding discussion, I've noted that many structures and tables are the same in both a COFF OBJ file and the PE file created from it. Both COFF OBJ and PE files have an IMAGE_FILE_HEADER at or near their beginning. This header is followed by a section table that contains information about all the sections in the file. The two formats also share the same line number and symbol table formats, although the PE file can have additional non-COFF symbol tables as well. The amount of commonality between the OBJ and PE EXE formats is evidenced by the large amount of common code in PEDUMP *(see COMMON.C on any MSJ bulletin board)*.
This similarity between the two file formats isn't happenstance. The goal of this design is to make the linker's job as easy as possible. Theoretically, creating an EXE file from a single OBJ should be just a matter of inserting a few tables and modifying a couple of file offsets within the image. With this in mind, you can think of a COFF file as an embryonic PE file. Only a few things are missing or different, so I'll list them here.

- COFF OBJ files don't have an MS-DOS stub preceding the IMAGE_FILE_HEADER, nor is there a "PE" signature preceding the IMAGE_FILE_HEADER.
- OBJ files don't have the IMAGE_OPTIONAL_HEADER. In a PE file, this structure immediately follows the IMAGE_FILE_HEADER. Interestingly, COFF LIB files do have an IMAGE_OPTIONAL_HEADER. Space constraints prevent me from talking about LIB files here.
- OBJ files don't have base relocations. Instead, they have regular symbol-based fixups. I haven't gone into the format of the COFF OBJ file relocations because they're fairly obscure. If you want to dig into this particular area, the PointerToRelocations and NumberOfRelocations fields in the section table entries point to the relocations for each section. The relocations are an array of IMAGE_RELOCATION structures, which is defined in WINNT.H. The PEDUMP program can show OBJ file relocations if you enable the proper switch.
- The CodeView information in an OBJ file is stored in two sections *(.debug$S and .debug$T)*. When the linker processes the OBJ files, it doesn't put these sections in the PE file. Instead, it collects all these sections and builds a single symbol table stored at the end of the file. This symbol table isn't a formal section *(that is, there's no entry for it in the PE's section table)*.

Using PEDUMP

PEDUMP is a command-line utility for dumping PE files and COFF OBJ format files. It uses the Win32 console capabilities to eliminate the need for extensive user interface work. The syntax for PEDUMP is as follows:

```
PEDUMP [switches] filename
```

The switches can be seen by running PEDUMP with no arguments. PEDUMP uses the switches shown in Table 17. By default, none of the switches are enabled. Running PEDUMP without any of the switches provides most of the useful information without creating a huge amount of output. PEDUMP sends its output to the standard output file, so its output can be redirected to a file with an > on the command line.

Table 17. PEDUMP Switches

/A	Include everything in dump (essentially, enable all the switches)
/H	Include a hex dump of each section at the end of the dump
/L	Include line number information (both PE and COFF OBJ files)
/R	Show base relocations (PE files only)
/S	Show symbol table (both PE and COFF OBJ files)

Summary

With the advent of Win32, Microsoft made sweeping changes in the OBJ and executable file formats to save time and build on work previously done for other operating systems. A primary goal of these file formats is to enhance portability across different platforms.

Windows 2000 Loader

What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

This article assumes you're familiar with Win32 APIs and C++

SUMMARY DLLs are a cornerstone of the Windows operating system. Every day they quietly perform their magic, while programmers take them for granted. But for anyone who's ever stopped to think about how the DLLs on their system are loaded by the operating system, the whole process can seem like a great mystery. This article explores DLL loading and exposes what really goes on inside the Windows 2000 loader. Knowing how DLLs are loaded and where, and how the loader keeps track of them really comes in handy when debugging your applications. Here that process is explained in detail.

ver since I first encountered a definition of dynamic link libraries in a description of the then-new operating system OS/2, the idea of DLLs has always fascinated me. This beautifully simple concept of modules that could be loaded and unloaded as needed with well-defined interfaces that outlined routines written beforehand and, perhaps by other programmers, was a powerful jolt to me because I was more accustomed to statically linked code in mainframe or MS-DOS® programs. And, like many others new to programming for Windows®, the first utility I built enumerated DLLs that were already loaded into the system in order to demonstrate this concept at work. Now, even with the Windows world changing at a frenetic pace, employing COM interfaces and their ActiveX® components, and moving toward common language runtimes with their assemblies of managed code, the humble DLL remains at the center of things, providing services to the system on an as-needed basis.

During this long association with DLLs, I accepted their loading by the operating system as if it were magic and never truly appreciated the amount of work required by LoadLibrary and its variations. This article is an attempt to rectify that oversight by looking inside NTDLL.DLL. Since I do not have access to the source files, much of what I discuss here falls under that nebulous category of undocumented information and is therefore subject to change or obsolescence in future releases of the operating system.

The details that I'll cover are based on an examination of the binaries available when I wrote this article, Windows 2000 Professional (Build 2195: Service Pack 1). Access to a properly installed set of debug symbol files, .DBG and .PDB dated July 9, 2000, and working with a suitable debugger will make the information easier to understand.

This article can be seen as a precursor to Matt Pietrek's Under The Hood column in the September 1999 issue of *MSJ*, where Matt writes pseudocode for the operation of LdrpRunInitializeRoutines for Windows NT® 4.0 SP3 and describes how a library is initialized and when DllMain gets called. Note that I will refer to this column frequently.

My discussion will begin with a brief look at LoadLibrary, starting with LdrLoadDll, and will conclude when LdrpRunInitializeRoutines is invoked. While trying to follow the execution path needed to load a simple DLL using a debugger, you can easily become confused by the numerous unconditional jump statements and lost in the recursion common in the later stages of DLL loading, so I'll guide you carefully through the call to LoadLibrary.

Note that all code modules mentioned in this article can be found at the link at the top of this article.

All Paths Lead to LoadLibraryExW

There are several ways to get to LoadLibraryExW. For example, LoadTypeLibEx and CoLoadLibrary in the COM universe eventually call LoadLibraryExW. The two most familiar routes to LoadLibraryExW are LoadLibraryA and

LoadLibraryW. All you need to do is specify one parameter—the name of the DLL—and you are on your way.

But if you examine a disassembly of LoadLibraryA and LoadLibraryW, you will discover that they are merely thin wrappers around the more versatile LoadLibraryExA and LoadLibraryExW APIs, respectively. With LoadLibraryA, there is a curious test for the DLL twain_32.dll, but normally two zeroes are passed as the second and third parameters to LoadLibraryExA before continuing. LoadLibraryW is even more direct; the two zeroes are pushed onto the stack and the code moves directly onto LoadLibraryExW. These paths merge with an examination of LoadLibraryExA. There is a call to a helper routine to convert the DLL's name into a Unicode string before the code proceeds onto LoadLibraryExW.

LoadLibraryExW is a fairly involved routine which must decide between at least four different variations on the type of DLL-loading that the program wants to perform. In addition to the first parameter (which contains the name of the DLL), and the second parameter (which must be NULL according to the SDK documentation), there is a flag parameter that specifies the action to take when loading the module. You can ignore the flag value LOAD_LIBRARY_AS_DATAFILE, since it does not lead to the desired goal: LdrLoadDll, an API exported by NTDLL.DLL. The remaining valid values—0, DONT_RESOLVE_DLL_REFERENCES, and LOAD_WITH_ALTERED_SEARCH_PATH—all eventually find their way to LdrLoadDll.

It is interesting to note that there is no sanity check for the flag parameter that returns something like STATUS_INVALID_PARAMETER if values other than the documented ones are passed to LoadLibraryExW. For example, plugging in a reasonable value such as 4 results in a normal DLL load. In any case, the alternate paths taken with DONT_RESOLVE_DLL_REFERENCES or LOAD_WITH_ALTERED_SEARCH_PATH will be noted later in this article. For now, I will concentrate on the normal case in which dwFlags has a value of 0.

There's one more API exported from NTDLL.DLL that leads to LdrLoadDll, but I have found neither documentation for it nor any references to it in existing DLLs. The name of the API is LoadOle32Export. It accepts two parameters, the image base for Ole32.DLL and the name of the routine to find, and it returns the address to the requested function. Whether this is a mysterious secret path or some kind of holdover from past versions of the operating system is uncertain.

APIs Exported by NTDLL.DLL

Figure 21 lists the exported APIs in NTDLL.DLL beginning with the prefix "Ldr", and Figure 22 lists the internal routines beginning with "Ldrp". The main difference between the names for the Ldr*XXX* and Ldrp*XXX* routines is that the "p" indicates that these functions are private—hidden from the outside world. As you can see by examining the lists, familiar APIs such as GetProcAddress are wrappers around NTDLL exports like LdrGetProcedureAddress. In fact, many APIs in Kernel32 that are familiar to the SDK programmer, such as GetProcAddress, ExitProcess, ExpandEnvironmentStrings, and CreateSemaphore, pass the real work along to an NTDLL surrogate *(LdrGetProcedureAddress, NtTerminateProcess, RtlExpandEnvironmentStrings_U, and NtCreateSemaphore, respectively)*.

Now, let's take a close look at LdrLoadDll using DumpBin or my disassembler, PEBrowse *(available from http://www.smidgeonsoft.com)*.

```
0X77F889A9 PUSH 0X1
0X77F889AB PUSH DWORD PTR [ESP+0X14]
0X77F889AF PUSH DWORD PTR [ESP+0X14]
0X77F889B3 PUSH DWORD PTR [ESP+0X14]
0X77F889B7 PUSH DWORD PTR [ESP+0X14]
```

```
0X77F889BB CALL 0X77F887E0 ; SYM:LdrpLoadDll
0X77F889C0 RET 0X10
```

Do not be deceived by the apparent simplicity of this routine, which looks like merely a wrapper around the internal procedure, LdrpLoadDll *(that I'll discuss in the next section)*. The first parameter points to a wide-string representation of the search path. The second parameter will hold a DWORD with the value of 2 if DONT_RESOLVE_DLL_REFERENCES was specified in the call to LoadLibraryExW. The third parameter contains a pointer to a Unicode string structure that encases the name of the DLL that is to be loaded. The fourth item is an output parameter and will receive the address at which the module was loaded when LdrpLoadDll has finished its work.

But what does the fifth parameter, hardcoded with a value of 1, mean? As you will see later, LdrpLoadDll can be called recursively in LdrpSnapThunk. Here, the value means normal processing, but later you will see that a value of 0 means process forwarded APIs.

LdrpLoadDll

I have provided a small project in the code download for this article that contains a main executable, ingenuously named Test, and three DLLs: TestDll, Forwarder, and Forwarded. The DLLs demonstrate different variations that illustrate some of the scenarios LdrpLoadDll will commonly encounter. See the Readme.txt file in the code download for important information about setting environment variables and predefined breakpoints.

Figure 23 shows some of the internal loader routines you will bump into when you pass one of the documented flags to LoadLibraryExW. If you concentrate for a moment on the typical situation (#1 in Figure 23), you will see that there are six subroutines called directly by LdrpLoadDll: LdrpCheckForLoadedDll, LdrpMapDll, LdrpWalkImportDescriptor, LdrpUpdateLoadCount, LdrpRunInitializeRoutines, and LdrpClearLoadInProgress. *(I'll discuss the first four subroutines later in this article.)* LdrpRunInitializeRoutines has already been described in Matt Pietrek's column, so I won't go into it here. LdrpClearLoadInProgress is briefly mentioned in that column as well.

Let's take a high-level look at the steps taken by LdrpLoadDll, which occur as follows:

1. Check to see if the module is already loaded.
2. Map the module and supporting information into memory.
3. Walk the module's import descriptor table (that is, find out what other modules this one is adding).
4. Update the module's load count as well as any others brought in by this DLL.
5. Initialize the module.
6. Clear some sort of flag, indicating that the load has finished.

Looking more closely at the routines listed, you will notice that LdrpCheckForLoadedDll can be and is called from several locations. It is therefore reasonable to assume that this is a helper function. Also, note that LdrpSnapThunk and LdrpUpdateLoadCount, as well as the previously mentioned case for LdrpLoadDll, are all candidates for recursion.

The code for LdrpLoadDll.cpp contains pseudocode for the operations found in this and the other loader routines. My pseudocode is not a copy of the actual code for LdrpLoadDll and should not be compiled; it represents intelligent guesswork obtained by studying a disassembly of the routine. It also shows how much information can be brought together about any section of code just by taking the time to study the available binaries. The pseudocode by no means covers every detail.

In many cases, the variable names were assigned based on their role in the code. Others were taken from parameter descriptions found in *Windows NT/2000 Native API Reference* by Gary Nebbett (New Riders Publishing, 2000). Take a

look at this book to see what takes place inside NTDLL.DLL. It provides a reference for the Nt/Zw routines—the so-called native APIs—and where possible relates these routines back to their Win32® counterparts.

Now I'll describe in some detail what happens when your code loads the typical DLL, then I'll throw in a few options for variety. LdrpLoadDll first sets up a __try/__except block, then checks the flag, LdrpInLdrInit. If the flag is turned on, then it sets up a critical section block to prevent updates in the data structures that it will be referencing and modifying. Next, it checks on the length of the incoming DLL's name against the maximum, 532 bytes or 266-wide characters, which is close to the better-known constant MAX_PATH and its value of 260. If the length of the name exceeds the maximum value, then the routine quits with a return code of STATUS_NAME_TOO_LONG.

At this point, LdrpLoadDll attempts to locate the position of the period character (dot) in the module's name. If the dot is missing, then the routine appends the string ".dll" to the end of the module's name, assuming this does not exceed the maximum size. There are two items to note here: first, you do not need to pass the .dll extension in your calls to LoadLibrary, and second, some of the features, such as the API-forwarding discussed later in this article, will not work with any extension other than .dll. (For a preview, try the following: change the output name of the Forwarded project to "Forwarded.cpl" and run the test program, making certain that the GetProcAddress stuff is included. You'll see that the GetProcAddress call fails and the messagebox that should display "Hello World!" will not appear!)

The test for ShowSnaps and enabling this handy debugging aide on your system were explained in Matt's column. ShowSnaps occupies the second bit position in the Registry value GlobalFlag, located at HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager. When enabled, this provides feedback in a debugger's output window about actions the loader is taking. If you decide not to enable "snaps" on your system, then the text strings appearing throughout the code will provide useful diagnostic hints. *(In the download is an ASCII text file, LdrSnap(Forwarder).TXT, that includes the snap outputs produced by loading the Forwarder test DLL. Later in the article you will find figures showing this information for Forwarded.DLL and TestDll.DLL.)* LdrpLoadDll then constructs a Unicode string of the module's name that will be used by the first subroutine, LdrpCheckForLoadedDll.

LdrpCheckForLoadedDll

The pseudocode for this routine can be found in LdrpCheckForLoadedDll.cpp. You'll see that the code checks all of the possible lists for loaded DLLs and if the initial search fails, the routine tries the alternative. Following all of the paths in this routine is rather complicated, but the essence of this routine is rather simple to understand. There are two places to start a search: an optimization based on a hash table found inside of NTDLL.DLL, and walking the module list maintained inside the process's environment block (PEB). The definition of the module list and its location in memory will follow shortly. LdrpLoadDll, the calling routine, starts the search by setting the UseLdrpHashTable parameter to 0. Later on, I'll show you a similar case in which the parameter is set to 1 and the hash table is used. If the DllName parameter does not contain a path, the comparison using RtlEqualUnicodeString is made on the simple file name entries.

There is a small wrinkle in the search option when the hash table is not used, however. If the incoming DllName contains a path, LdrpCheckForLoadedDll calls on RtlDosSearchPath_U to validate the path, then examines the entries in the module list that contain fully qualified file names. In case you were wondering how the Microsoft® .NET Common Language Runtime (CLR) supports side-by-side execution of different versions of the same assembly, this information should give you a head start. If the module is found in the PEB list but contains a load count of 0, LdrpCheckForLoadedDll forces the load by issuing a sequence of commands similar to what will be seen in LdrpMapDll. But describing this sequence now would be jumping too far ahead.

If the search has succeeded and the loading DLL is already part of the current process, then LdrpLoadDll has a few

remaining details to take care of. The loaded count for this image will be incremented if the module is a DLL and if the loaded count field has not been set to -1 (which seems to mean "do not update the load count.") The module flag will also be cleared of its load-pending bit, which was set by the routine LdrpUpdateLoadCount. LdrpLoadDll will then leave the critical section block and set the ImageBase parameter to the address where the DLL was loaded.

More than likely, though, this is not the quick exit you are looking for—unless you are wasting machine cycles by using LoadLibrary to return an HINSTANCE of an already loaded module. Otherwise, GetModuleHandle is a better alternative. Instead, let's see what happens when the search fails.

LdrpMapDll

Now that LdrpLoadDll knows that the requested DLL has not already been loaded into the process, it calls upon its second helper routine, LdrpMapDll, to perform the duties of finding the DLL's housing (the actual file), loading the DLL into memory (but not initializing it), and creating and adding a structure that I have tagged, MODULEITEM, to the PEB's module list. First, orient yourself in Figure 23 and become familiar with the support routines you can see in LdrpMapDll, (LdrpCheckForKnownDll, LdrpResolveDllName, LdrpCreateDllSection, LdrpAllocateDataTableEntry, LdrpFetchAddressOfEntryPoint, and LdrpInsertMemoryTableEntry). In the file LdrpMapDll.cpp, you will find the pseudocode that shows the fine points of what actually takes place here.

LdrpCheckForKnownDll checks to see if the loading DLL can be located in the directory specified in the Unicode string, LdrpKnownDllPath, located at 0x77FCE008. Examining this memory location with a debugger reveals that this variable contains the value "C:\WINNT\SYSTEM32" (or something similar, depending upon how your system is configured).

If you start up WinObj (found at http://www.sysinternals.com) or my utility, NtObjects (found at http://www.smidgeonsoft.com), and select View | Executive Objects, you will find an object directory called KnownDlls. Under this directory, you will see the item KnownDllPath (a symbolic-link object) that contains the value for your system. LdrpCheckForKnownDll allocates two Unicode strings to hold the fully qualified DLL file name and the file name by itself. By calling NtOpenSection with the fully qualified file name, a FileExists operation is performed and LdrpCheckForKnownDll returns a section handle if the DLL is known. Otherwise, the two Unicode strings are freed and LdrpMapDll must call on the services of LdrpResolveDllName and LdrpCreateDllSection.

LdrpResolveDllName returns two Unicode strings, the loading DLL's file name and its fully qualified file name. If a search path was not provided, the routine uses the path found at the hardcoded address 0X77FCE30C in NTDLL.DLL, which points to a default search path. RtlDosSearchPath_U, though, performs the real work in this routine. If RtlDosSearchPath_U can find the loading DLL, it returns the length of the path where it resolved the DLL's name.

If you try to load a DLL that cannot be found in the search path, the search returns 0. LdrpMapDll then responds to this result and returns with the status code STATUS_DLL_NOT_FOUND, which leads to a quick exit from LdrpLoadDll. If you change the test program so that it tries to load a DLL with the name "bogus," you can check this for yourself.

Assuming all is well and the loading DLL has been found, LdrpCreateDllSection must take over and create the all important section handle. Since sections are classified as kernel objects, the path needs to be converted into something that the Executive Object Manager can understand. This is where the routine RtlDosPathNameToNtPathName comes into play. It takes a fully qualified file name "C:\Projects\LoadLibrary\Debug\TestDll.dll" and returns something like "\??\C:\Projects\LoadLibrary\Debug\TestDll.dll." If the file name cannot be interpreted, then the routine returns STATUS_OBJECT_PATH_SYNTAX_BAD and ends execution.

LdrpCreateDllSection is just a thin wrapper around the so-called native API, NtCreateSection. First, a file handle is obtained with NtOpenFile. If that call fails, then a return code of STATUS_INVALID_IMAGE_FORMAT is generated. Otherwise, the Object Manager creates the section handle and the file handle is closed via NtClose.

Now that LdrpMapDll has the section handle, it can actually load the DLL into the process's address. The DLL is brought in as a memory-mapped file through the services of NtMapViewOfSection. First, the defaults are set for the base address and the size of the mapping object—with the latter, a value of 0 indicates to the system that the entire section object will be mapped. Then, a field in the PEB reserved for subsystem calls is loaded with the value of the fully qualified file name.

Those of you who have written debug loops and have handled the debug event LOAD_DLL_DEBUG_EVENT may be interested to know that the lpImageName field of the LOAD_DLL_DEBUG_INFO structure is filled with the contents of this field, which is reserved for subsystem calls in the process that's being debugged.

Now, NtMapViewOfSection is called. This triggers the notification of the loading DLLs as seen in debuggers like the one in Visual C++®. LdrpMapDll restores the previous value of the PEB's subsystem data field and checks on the success of the operation. If the DLL has been successfully mapped, LdrpMapDll now possesses an actual memory address with which it can work. How NtMapViewOfSection returns the image base when no hint was passed to it is a question that could be resolved with further exploration.

PEB Load List

After a sanity check on the image now mapped into memory, LdrpMapDll continues with some bookkeeping. Earlier I mentioned a data structure I named MODULEITEM that is created and added to the process's PEB. In the code for LdrpLoadDll.h you will find a reconstruction of this structure. The job of LdrpAllocateDataTableEntry is to allocate this object and initialize the ImageBase field using three values: the HMODULE handle returned by NtMapViewOfSection, the ImageSize field using the SizeOfImage value found in the portable executable (PE) file's optional header, and the TimeDateStamp from the equivalent field in the PE file header. If the memory allocation fails, the routine returns a NULL pointer. LdrpMapDll then removes the file mapping, closes the section handle, and fails, returning STATUS_NO_MEMORY.

Assuming that all is well, the LoadCount field in the newly built ModuleItem is zeroed out and ModuleFlags gets initialized. (LdrpLoadDll.h provides the possible values for this field.) The two Unicode string fields containing the full path to the DLL and the file name are filled in. Then a call placed to the small helper routine, LdrpFetchAddressOfEntryPoint, inserts the structure's EntryPoint field.

With the ModuleItem partially initialized, LdrpMapDll calls LdrpInsertMemoryTableEntry to insert the entry into the process's module list located in the PEB. At offset 0x0C in the PEB, which can usually be found at the address of 0x7FFDF000, you will find the pointer to a structure I have named MODULELISTHEADER. Windows NT and Windows 2000 maintain three doubly linked lists that describe the load, memory, and initialization order for the modules in a process.

If you have ever used the PSAPI routines or the newly available ToolHelp32 functions in Windows 2000 to enumerate the modules loaded in a process's address space, you should be having an epiphany. You have an alternate method to gather this information, in three different flavors, by using ReadProcessMemory and these undocumented structures. These structures have remained stable from Windows NT 4.0 through Windows 2000, and through the betas for Windows XP that were available when I did the research for this article. LdrpInsertMemoryTableEntry adds the ModuleItem structure to the end of the load and memory chains and fixes up the links appropriately. In my experience, I have found that the load and memory chains possess the same order; only the initialization list varies with its own order of the modules. See Matt Pietrek's column for more information on the initialization chain.

Returning to LdrpMapDll, you can now see the ModuleFlags field receiving some additional attention. If the executable is not IMAGE_FILE_LARGE_ADDRESS_AWARE and it is not of type IMAGE_DLL, the EntryPoint field gets

zeroed out. LdrpMapDll tests the return value from NtMapViewOfSection to determine if the image was loaded at its preferred image base. Unfortunately, scenarios in which your DLL is parked in a different location is beyond the scope of this discussion, but now you know the way, so you can investigate this phenomenon on your own.

Finally, after a validation of the image on multiprocessor systems, LdrpMapDll closes the section handle obtained from LdrpCreateDllSection and returns with the results of its work. The validation only proceeds if your DLL contains data in the directory IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG and the LockPrefixTable field in this directory is nonzero. (NTDLL and Kernel32 contain the IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG directory.) You can use PEBrowse to examine this directory, and those of you fortunate enough to have a multiprocessor machine can continue your own exploration down this path.

LdrpWalkImportDescriptor

You may think that the route through LdrpLoadDll has been relatively straight and uncomplicated so far. But there still may be a maze of passages ahead. Your attempt to load LdrpLoadDll may have generated the need for additional modules, and this is where LdrpWalkImportDescriptor comes in. (In order to better understand my pseudocode, it will help to have some knowledge of the PE header definitions found in WinNt.h, especially those relevant to imports and exports. You can take a look at "Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format" by Matt Pietrek in the February 2002 issue of *MSDN Magazine*.

Your typical module load takes you through the twists and turns of LdrpWalkImportDescriptor. However, if you specify DONT_RESOLVE_DLL_REFERENCES in your call to LoadLibraryExW, then you will avoid the upcoming maze. You should read the SDK documentation carefully to make certain that this is what you want. There is also a mechanism, which I'll explain later, to help you avoid the loops and recursion that are part of LdrpSnapIAT and LdrpSnapThunk. But if you don't take either of these alternatives, then you need to know what happens with LdrpWalkImportDescriptor.

Orient yourself once again by reexamining Figure 23, locating LdrpWalkImportDescriptor. LdrpWalkImportDescriptor has two subroutines: LdrpLoadImportModule and LdrpSnapIAT. This does not seem so bad, but one tip-off that this code will soon become interesting is that there are four nesting levels in the routines for LdrpSnapIAT. The number and depth of nested functions is one metric that indicates the complexity of code. You should take note that recursion is possible in not one, but two locations in LdrpSnapIAT. You may recall that in the section on APIs exported by NTDLL.DLL I mentioned the apparent simplicity of the call to LdrpLoadDll and the fifth parameter that took a 0 or a 1. LdrpSnapIAT can also be recursive inside LdrpGetProcedureAddress. Finally, to make things even more complex than they already were, it's possible that a typical DLL may import other modules that start a cascade of additional library loads. The loader will need to loop through each module, checking to see if it needs to be loaded and then checking its dependencies.

With that in mind, let's take a look at the pseudocode found in LdrpWalkImportDescriptor.cpp. (If you are following along with the debugger, change the test program to load the Forwarded.DLL module and restart the debugger.) Execution starts with two calls to RtlImageDirectoryEntryToData to locate the Bound Imports Descriptor and the regular Import Descriptor tables. For the moment, ignore the call for that bound import thing except to notice that the code checks for its presence first. (I'll discuss binding later.) In Forwarded.DLL, LdrpWalkImportDescriptor detects two imported modules, User32.DLL and Kernel32.DLL, and now calls upon LdrpLoadImportModule for assistance.

LdrpLoadImportModule constructs a Unicode string for each DLL found in the import table and then employs LdrpCheckForLoadedDll, using the hash table in NTDLL that was mentioned earlier to see if they have already been loaded. Note that the call here is made with only the file name (no fully qualified path) and the process's search path. If you have ever had your application complain that it cannot find a DLL, you should realize that it may not be the loading

module's fault. Check to see that all of its dependent modules can be found. If a module is found and already loaded, life is good because there is one less module to worry about. If it has not been loaded, then you've found an instance of recursion. A call to LdrpMapDll to bring the DLL into the process' address space is followed by a call to LdrpWalkImportDescriptor. Now you're in the middle of the twisting mazes I mentioned.

The IAT and Forwarded API Processing

Once LdrpWalkImportDescriptor knows that the module is in memory (either through a call to LoadLibraryExW way back at the beginning or via the LdrpMapDll call in LdrpLoadImportModule), the next step is to examine each and every API referenced in Forwarded's imported module list with the call to LdrpSnapIAT. Why is this necessary? There are two reasons. First, you want to replace the placeholders in the Import Address Table (IAT) with real entry points. Second, you need to locate and process APIs that may have been forwarded on to another DLL. (Forwarding is one technique Microsoft employs to expose a common Win32 API set and to hide the low-level differences between the Windows NT and Windows 9*x* platforms.)

You may have observed either in the debug output strings or by carefully stepping through the loader code that at some point during Kernel32 processing, a check on NTDLL.DLL was made. Since just about any DLL with some executable code contains references to Kernel32, you may wonder why this is happening at all. You already know that NTDLL.DLL is loaded into every process. The answer is that Kernel32 contains APIs that are "forwarded" to NTDLL. Refer to Figure 24 for a complete list of these APIs for Kernel32 (version 5.0.2195.1600).

You may also notice that the list contains functions that are common requirements for just about any kind of serious programming. Remember, LdrpLoadDll needs to load every module referenced by the loading DLL, including those "hidden" references contained in forwarded APIs. Thus, a reasonable conclusion from all of this is that an import table walk will take place every time you load a DLL—at least for Kernel32's forwarded APIs and for any additional forwarded APIs you may have decided to include in your application!

To allow you to experiment with this concept, I have included Forwarder.DLL and Forwarded.DLL in the download, as I've mentioned previously. The code for Forwarder.DLL is extremely simple—a DllMain with an export pragma. If you run DumpBin or PEBrowse on this module, you will see that it exports only one routine, and that the routine is marked with a designation that it is forwarded on to Forwarded.ShowMessage.

Now, turn your attention to Forwarded.DLL and take a look at what it exports. You will see a typical DLL with a DllMain and the single API, ShowMessage. However, if you changed the test program to load Forwarder and then observed what happened in the debugger, you might have wondered why no reference to Forwarded appeared. You will encounter this confusion unless you also included the GetProcAddress statement in your compile. Make certain that you include the GetProcAddress line and you will now see that both DLLs are loaded. You're observing another form of delay-load occurring and, perhaps, another example of the optimization that has been done on the loading engine in Windows.

Returning to LdrpWalkImportDescriptor, you'll find that it tests several items before calling LdrpSnapIAT. (For the purposes of following the next few steps, change the sample back to use Forwarded.DLL.) If you look up Section 6.4.1 of the Portable Executable Specification (found on the MSDN Library CD under Specifications), you will read that the time/date stamp field of an import directory will contain a value of 0 unless the DLL has been bound. One of the fields tested is the time/date stamp field, which is 0 in Forwarded.DLL, so let's step into LdrpSnapIAT.

LdrpSnapIAT wraps its execution around a __try/__except block first before locating the IAT in Forwarded.DLL, and then hunts for the export directory in the module that Forwarded is attempting to load (assume it is Kernel32 for now). It then changes the memory protection on the IAT of Forwarded.DLL to PAGE_READWRITE and proceeds to examine

each entry in the IAT. (If you are able to examine the protection for this chunk of memory, you will see that it is normally PAGE_READONLY for your executables.) Going a bit further, you'll encounter LdrpSnapThunk.

LdrpSnapThunk requires an ordinal to locate an entry point and to determine whether or not the API is forwarded. If the hint value in Forwarded.DLL's import directory is correct, you can use that (generally, I have found this not to be the correct value). Otherwise, LdrpSnapThunk calls on the services of the helper routine, LdrpNameToOrdinal, to look up the correct value. Observe that LdrpNameToOrdinal uses a binary search on the export table to quickly locate the ordinal—more optimization in the loader—and note that the table must be sorted in alphabetical order for the search to work.

Now that you have an ordinal, you can look up the entry point for the API in Kernel32. LdrpSnapThunk first plugs the loading module's IAT entry with an address derived from the export table for Kernel32; see Section 6.4.4 of the PE specifications *(which can be found on the October 2001 MSDN CD under Specifications | Microsoft Portable Executable and Common Object File Format)* for more information. (This explains why the page protection was changed in LdrpSnapIAT.)

Section 6.3.2 of the PE specifications says:

> *If the address (the entry-point) is not within the export section (as defined by the address and length indicated in the Optional Header), the field is an Export RVA: an actual address in code or data. Otherwise, the field is a Forwarder RVA, which names a symbol in another DLL.*

So now you can finally decide whether or not the API has been forwarded. In the vast majority of cases, the API is not forwarded, but let's assume you are looking at Forwarded's reference to HeapAlloc. (Check the math first. Kernel32's image base (0x77e80000) + HeapAlloc's entry point (0x0005b658) is 0x77edb658, which is inside the range for the export table, 0x77ed5c20 to 0x77edb770.)

LdrpSnapThunk now proceeds to break apart the forwarded reference for HeapAlloc, which will have the format NTDLL.RtlAllocateHeap, and then calls LdrpLoadDll to obtain NTDLL's image base—hmm, this looks like you're back at the beginning. But note that the fifth parameter is passed with a value of 0. Also note that the DLL name that was constructed before making the call to LdrpLoadDll lacks the .DLL extension. Fortunately, the call to LdrpLoadDll will succeed when LdrpCheckForLoadedDll figures out that NTDLL.DLL is already loaded.

But do you remember that experiment where I changed the extension for Forwarded to .cpl? Try this again and you will see that LdrpLoadDll now fails on the LdrpMapDll call with a STATUS_DLL_NOT_FOUND return code. Now I have an explanation for the earlier results. With the module name out of the way, LdrpSnapThunk grabs the API name, RtlAllocateHeap, and forges on to LdrpGetProcedureAddress.

At this point, I am getting close to the end of this forwarded API processing, I promise. LdrpGetProcedureAddress is another routine that wraps its processing around a __try/__except block. The routine determines what type of API information it has been handed, either an ordinal if the API name parameter is null, or the name itself. The test is needed in order to properly set up parameters for an upcoming call to LdrpSnapThunk. But wait a moment. Didn't LdrpSnapThunk just bring us to this point? The two parameters are a pointer to the API's entry in the IAT and an overloaded item that contains either the image base of the loading DLL or a pointer to an IAT entry. If the flag, LdrpInLdrInit, is turned on, the process's critical section is entered. And now let's really dive in deep and step into LdrpCheckforLoadedDllHandle.

Fortunately, the functionality here is pretty simple to describe and understand. I need a MODULEITEM before I can continue. LdrpCheckforLoadedDllHandle first examines a handle cache residing at LdrpLoadedDllHandleCache to see

if the image base there is the same as its input parameter, hDll. If not, the routine perseveres by walking the LoadOrder list, searching for the MODULEITEM whose image base matches hDll and whose linkage in the memory order list has been established. Once it finds an entry matching these criteria, it updates the cache with it and hands back to LdrpGetProcedureAddress the MODULEITEM that it found, or returns a 0 to indicate failure.

With a MODULEITEM in hand, LdrpGetProcedureAddress now calls an old friend, RtlImageDirectoryEntryToData, to locate the item's export directory and starts that twisty, recursive call to LdrpSnapThunk. What is going on here? Why is this call necessary? The answer, in part, is that this new API itself may be forwarded! I am not aware of such a situation in Windows 2000, but the possibility certainly exists. Happily, HeapAlloc's processing ends with RtlAllocateHeap inside of NTDLL, and LdrpSnapThunk returns an IAT entry with the entry point to this API. LdrpGetProcedureAddress frees up any work areas it might have created, exits the critical section (if it was acquired), and returns. Whew!

Next, LdrpSnapThunk checks the return code and returns STATUS_ENTRYPOINT_NOT_FOUND if the API was not found. Otherwise, it replaces the entry in the IAT with the API's entry point and continues on. Study Section 6.4.4 in the PE specifications and especially the references to binding for a more complete picture of what is happening.

Now let's return to LdrpSnapIAT and move on to the next imported API in Kernel32 (or break from the loop if the LdrpSnapThunk call failed). Once all of the entries are processed in Kernel32's import table, LdrpSnapIAT restores the memory protection it changed at the beginning of its work, calls NtFlushInstructionCache to force a cache refresh on the memory block containing the IAT, and returns back to LdrpWalkImportDescriptor. The cache refresh might be a little surprising, but in many executables the IAT can be found in the .text section where code is found. If LdrpWalkImportDescriptor does not flush the memory block containing the updated IAT, then all of the previous work will have been for naught because the processor may continue to use the old version of the memory block. (For more information read the SDK documentation for the Kernel32 API FlushInstructionCache, which is just a thin wrapper around NtFlushInstructionCache.)

If you want to see the results of RunTime Binding via LdrpSnapIAT and LdrpSnapThunk on the IAT for Forwarded.DLL (SP1), take a look at Figure 25.

Bound DLL Processing

You may recall that LdrpWalkImportDescriptor tested for the existence of two directories or descriptors, the regular Import Descriptor and something called the Bound Imports Descriptor, and tried to use the Bound Imports Descriptor first if it was available. Also, LdrpSnapIAT examined the time/date stamp in the Import Directory Table for a special value of -1 before moving on to LdrpSnapThunk. There just may be an alternative to all of this import table munging by pre-binding your DLL. Now is the time to change my test project to load TestDll and see what happens in the Windows 2000 loader with a module that has been bound ahead of time.

If you have not created the environment variable "MSSdk" as I mentioned in the readme.txt file, and set it equal to the root directory where your Platform SDK is located, do so now and rebuild TestDll (a post-link step should kick in that performs the binding operation). Or, from a command line you can enter and run the following command:
bind -u testdll.dll

When you examine the resulting executable, you should see that a new directory in the optional header array has been filled: the slot corresponding to IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT. Launching the result in the debugger, you will also observe that the tests for the Bound Imports Descriptor will succeed. Try the test without a call to GetProcAddress on "fnTestDll" and you will see that when LdrpWalkImportDescriptor issues its call to LdrpLoadImportModule, the check for an already loaded module (Kernel32.DLL) will succeed and that means you can avoid the nasty bumps and turns that made the code very complex earlier in the discussion of

LdrpWalkImportDescriptor. My DLL loads faster because there is no looping through the APIs I imported from Kernel32 since the fix-ups have already been done (including the forwarded references). I feel like I've found the Holy Grail.

But this old cup loses some of its shine when I change the sample to call fnTestDll. Before continuing, see if you can foretell why you will be walking that twisty maze again. The reason is that Forwarded.DLL, the module that contains the real code for my forwarded API, fnTestDll, was not itself bound. Run the bind utility on Forwarded.DLL and the brilliance of my newfound treasure returns. The moral of this little exercise is that in order to gain the full measure of efficiency that pre-binding a module provides, make certain that all the subordinate modules have been bound, too.

There is a slight downside to pre-binding a DLL, though. What happens when the next version of the operating system appears with a new version of Kernel32 and new locations for the exported functions? Or consider the consequences of another module loading and occupying the slot reserved in memory for that DLL you have bound your executable to. Your bindings, the hardcoded addresses in the IAT, will be incorrect and considered stale by the loader. Under these conditions, the binding is effectively ignored, the LdrpWalkImportDescriptor processing takes place, and you are no better off than you were before.

On the other hand, if you can manage to keep your DLLs in sync with the current versions of the system DLLs and any others you may use, you should see an improvement in your module loads. As the SDK documentation states: "You can minimize load time by using Bind to bypass this lookup." (For more discussion concerning BIND.EXE and other load issues, see Under the Hood in the May 2000 issue of *MSDN Magazine*. Also, note that the system DLLs, Kernel32, GDI32, User32, AdvApi32, and so on have been pre-bound.) Figure 26 shows the results of pre-binding using the SDK Bind utility on the IAT for TestDll.DLL (SP1).

LdrpUpdateLoadCount

If you take stock of what you have seen and learned so far, you will realize that the first three parts in LdrpLoadDll's processing have been completed. The last part of LdrpLoadDll to explore involves an update to module reference counts. That is the job for LdrpUpdateLoadCount.

LdrpUpdateLoadCount is a dual-purpose routine; it is called when the DLL is both loading and unloading. It attempts to walk either the Bound Imports table or the Imports table, and it will recurse on itself for any subordinate modules. The result is code that will likely be difficult for you to follow, but LdrpUpdateLoadCount.cpp contains my attempt to write pseudocode for this procedure. Distilling the essence of the pseudocode leaves the following: LdrpUpdateLoadCount walks through either the Bound Imports Descriptor or the Imports Descriptor looking for imported modules using LdrpCheckForLoadedDll and the NTDLL hash table. If the module was newly loaded by a LoadLibraryExW call, then LdrpUpdateLoadCount updates its reference count and walks its tables for any imports.

You can easily imagine a tree structure that describes the relationships between DLLs and their imports, and LdrpUpdateLoadCount must walk the tree completely to update everyone's reference count correctly. Some modules enter the process with a reference count of -1 and are skipped by this update. I leave here a question for future exploration: why do some DLLs have a reference count of -1 and the others contain an actual count?

Longtime readers of Under the Hood may recall a handy utility Matt Pietrek wrote named NukeDll. Back in the ancient days of 16-bit Windows 3.*x*, an application that General Protection Faulted had the nasty habit of leaving wreckage strewn about in memory in the form of orphaned Dlls—their reference counts never reached 0 and were not released from the common address space that was part of Windows. Using NukeDll, you could nuke these orphans out of existence and free precious resources. The need for a utility like that has been virtually eliminated with Windows NT and its successors because of the compartmentalization imposed upon processes by the operating system. Still, the

reference count is important; your application could be loading and unloading modules but leaking HINSTANCE's because of a mismatch in the LoadLibrary/FreeLibrary pairs. But this will only hurt your own buggy application and you will only chomp through your own resources; other processes in Windows NT and Windows 2000 will be isolated from this behavior.

Once LdrpUpdateLoadCount has finished its work and has returned, LdrpLoadDll checks the return code from LdrpWalkImportDescriptor. If the code is STATUS_SUCCESS, processing continues on to DLL initialization (which was described in Matt Pietrek's September 1999 Under the Hood column) and is followed by leaving the process's critical section. But if there was a problem in LdrpWalkImportDescriptor, then LdrpLoadDll must back out all of the work done up until now, mostly by invoking LdrpUnloadDll. An image base is sent back to LoadLibraryExW in the form of an HMODULE.

The LOAD_WITH_ALTERED_SEARCH_PATH Option

There is one scenario that I have not described yet, and that is when a call to LoadLibraryExW is made with dwFlags equal to LOAD_WITH_ALTERED_SEARCH_PATH. Now that you have grown somewhat accustomed to wandering around DLLs, you might want to experiment on your own with this small wrinkle. Change the Test sample program to issue this version of a LoadLibraryExW call and pay particular attention to the first parameter for LdrpLoadDll and note any differences.

You might also want to create two copies of TestDll, storing one copy in your \TEMP directory, and observing what happens. With a minimum amount of effort you will be able to manufacture a situation where two copies of TestDll are loaded, one from the current working directory and the second from the temp directory. That would demonstrate how the .NET CLR support for side-by-side execution of different versions of the same assembly might be implemented (although this side-by-side capability has been available since at least Windows NT 4.0).

What You've Learned

Next time the debugger displays a DLL load notification, you will know with some degree of confidence the state that the module is in: it has been mapped into memory, it has not been added to the PEB's housekeeping area, and, most importantly, it has not been initialized yet.

You also have learned that the PEB contains not one, but three lists enumerating loaded modules in load, memory, and initialization order. (There are also many other fields in the PEB worthy of examination since they lead to other vital pieces of information on your process.) As Matt Pietrek has pointed out, the order of the DLLs you see displayed inside the debugger is not the order in which DLLs are initialized, as many people mistakenly believe.

Probably the most important fact to hold onto is that a simple call to LoadLibrary results in many more things occurring under the covers than might initially be apparent. The loader must examine each and every API that DLL imports from other DLLs in order to calculate a real address in memory and perhaps load additional DLLs and check to see if an API may have been forwarded on to another procedure housed in another DLL.

A loading DLL may bring in additional modules where the process just described will be repeated over and over again.

The overhead that all of this processing brings to your application may be reduced by investigating the use of the SDK utility, BIND.EXE. The loader still checks the reference to each DLL contained in your program, but as long as the entries are not stale (in other words, the entries are still correct), the address calculation and forwarded API processing will be safely bypassed.

Finally, you have seen that DLLs are reference-counted, just as they were in the ancient Windows 3.*x* days. Although this count does not have the same systemwide effect that it once had, a DLL that you are trying to manage dynamically will still produce resource leaks if you have not properly matched up FreeLibrary calls with each LoadLibrary call.

Conclusion

You should prepare yourself for additional trips into NTDLL.DLL during future debugging sessions because, like LoadLibraryEx, many Kernel32 APIs lead inevitably to undocumented routines that reside in NTDLL. If you end your investigation prematurely because of a reluctance to enter this uncharted territory, you may miss the real cause of your bug or, at least, a better understanding of your problem. I plan to maintain the pseudocode at my Web site, http://www.smidgeonsoft.com, so if you find any errors or improvements, please pass them along and I will incorporate them into the code listings.

References Links

For related articles see:

- An In-Depth Look into the Win32 Portable Executable File Format – Part1
- An In-Depth Look into the Win32 Portable Executable File Format – Part2
- Peering Inside the PE: A Tour of the Win32 Portable Executable File Format
- Under the Hoood
- What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

For background information see:
Microsoft PE and COFF Specification - V 8.3

About the Authors

Matt Pietrek *is an independent writer, consultant, and trainer. He was the lead architect for Compuware/NuMega's Bounds Checker product line for eight years and has authored three books on Windows system programming. His Web site, at http://www.wheaty.net, has a FAQ page and information on previous columns and articles.*

Russ Osterlund *is a software engineer in the BoundsChecker group with Compuware/NuMega Labs. He, his wife, and their cat are new residents in the state of New Hampshire. He can be reached at RussellOsterlund@cs.com or at his Web site, http://www.smidgeonsoft.com.*