From the February 2002 issue of MSDN Magazine

Inside Windows

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

Matt Pietrek

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

Level of Difficulty 1 2 3

Download the code for this article: PE.exe (98KB)
[http://download.microsoft.com/download/msdnmagazine/code/Feb02/WXP/EN-US/PE.exe]

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.

A 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 [http://msdn2.microsoft.com/en-us/magazine/ms809762(printer).aspx] ," 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**).

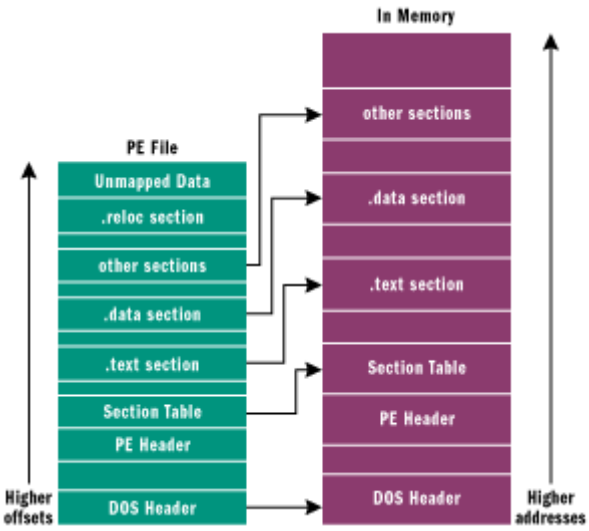

Figure 1 Offsets

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 [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 [http://www.microsoft.com/msj/0797/hood0797.htm] 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
  01 .text     VirtSize: 00074658  VirtAddr:  00001000
    raw data offs:   00000400  raw data size: 00074800
•••
  02 .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 [http://msdn2.microsoft.com/en-us/magazine/bb985997(printer).aspx] 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 [http://msdn2.microsoft.com/en-us/magazine/bb985997(printer).aspx] 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 [http://msdn2.microsoft.com/en-us/magazine/bb985997(printer).aspx] lists the common values of IMAGE_FILE_*xxx*. Figure 5 [http://msdn2.microsoft.com/en-us/magazine/bb985997(printer).aspx] 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 [http://msdn2.microsoft.com/en-us/magazine/bb985997(printer).aspx] 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.

For related articles see:
Peering Inside the PE: A Tour of the Win32 Portable Executable File Format
[http://msdn2.microsoft.com/en-us/magazine/ms809762(printer).aspx] **For background information see:**
The Common Object File Format (COFF) [http://support.microsoft.com/default.aspx?scid=kb;en-us;q121460]

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 [http://www.wheaty.net/] , has a FAQ page and information on previous columns and articles.*