msdn magazine

Inside Windows

An In-Depth Look into the Win32 Portable Executable File Format, Part 2

Matt Pietrek

From the March 2002 issue of MSDN Magazine

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

Level of Difficulty 1 2 3

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 1 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 2. 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**).

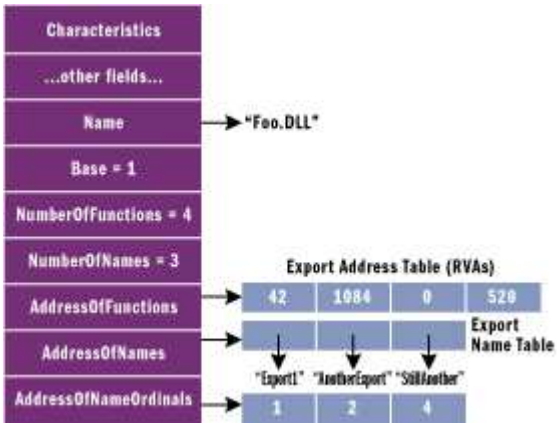

Figure 3 The IMAGE_EXPORT_DIRECTORY Structure

Let's go through an example to show exports at work. Figure 4 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 NTÂ®, WindowsÂ® 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
â€¢â€¢â€¢
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 5 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.

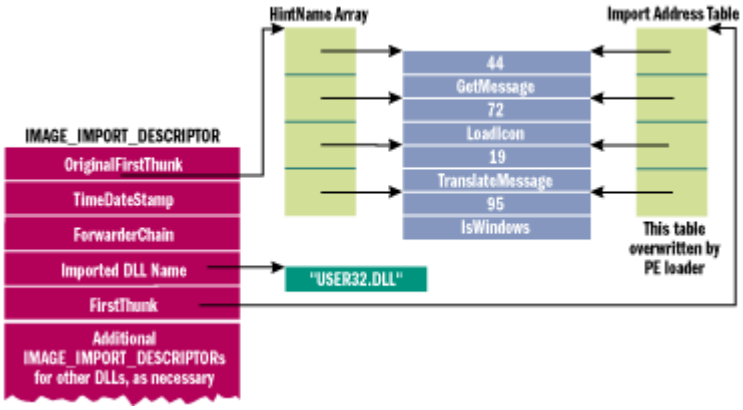

Figure 6 Two Parallel Arrays of Pointers

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 StudioÂ® .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 systemâ€”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.

For related articles see:
Part 1 of this series
Peering Inside the PE: A Tour of the Win32 Portable Executable File Format

For background information see:
The Common Object File Format (COFF)

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.*
