Chapter **11**

Portable Executable File Format

IN THIS CHAPTER

+ Understanding the structure of a PE file
+ Talking in terms of RVAs
+ Detailing the PE format
+ The importance of indices in the data directory
+ How the loader interprets a PE file

MICROSOFT INTRODUCED A NEW executable file format with Windows NT. This format is called the *Portable Executable (PE)* format because it is supposed to be portable across all 32-bit operating systems by Microsoft. The same PE format executable can be executed on any version of Windows NT, Windows 95, and Win32s. Also, the same format is used for executables for Windows NT running on processors other than Intel x86, such as MIPS, Alpha, and Power PC. The 32-bit DLLs and Windows NT device drivers also follow the same PE format.

It is helpful to understand the PE file format because PE files are almost identical on disk and in RAM. Learning about the PE format is also helpful for understanding many operating system concepts. For example, how operating system loader works to support dynamic linking of DLL functions, the data structures involved in dynamic linking such as import table, export table, and so on.

The PE format is not really undocumented. The WINNT.H file has several structure definitions representing the PE format. The Microsoft Developer's Network (MSDN) CD-ROMs contain several descriptions of the PE format. However, these descriptions are in bits and pieces, and are by no means complete. In this chapter, we try to give you a comprehensive picture of the PE format.

Microsoft also provides a DLL with the SDK that has utility functions for interpreting PE files. We also discuss these functions and correlate them with other information about the PE format.

Overview of a PE File

In this section, we discuss the overall structure of a PE file. In the sections that follow, we go into detail about the PE format. A PE file comprises various sections. Because Microsoft's 32-bit operating systems follow the flat memory model, an executable no longer contains segments. Still, different parts of an executable, such as code and data, have different characteristics. These different parts of an executable are stored as different sections. Thus, a PE file is a concatenation of data stored in sections.

A few sections are always present in a PE file generated by the Microsoft linker. Other linkers may generate similar sections with different names. A PE file generated with the Microsoft linker has a .text section that contains the code bytes concatenated from all the object files. As for the data, it can be classified into different categories. The .data section contains all the initialized global and static data, while the .bss section contains the uninitialized data. The read-only data, such as string literals and constants, is stored in the .rdata section. This section also contains some other read-only structures, such as the debug directory, the Thread Local Storage (TLS) directory, and so on, which we explain later in this chapter. The .edata section contains information about the functions exported from a DLL, while the .idata section stores information about the functions imported by an executable or a DLL. The .rsrc section contains various resources, such as menus and dialog boxes. The .reloc section stores the information required for relocating the image while loading.

The names of the sections do not have any significance. As mentioned earlier, different linkers may use different names for the sections. Programmers can also create new sections of their own. The #pragma code_seg and #pragma data_seg macros can be used to create new sections while working with Microsoft compiler. The operating system loader locates the required piece of information from the data directories present in the file headers. Shortly, we will present an overview of file headers and then look at them in more detail.

Structure of a PE File

Apart from the sections consisting of the actual data, a PE file contains various headers that describe the sections and the important information present in the sections.

If you look at the hex dump of a PE file, the first 2 bytes might look familiar. Aren't they M and Z? Yes, a PE file starts with the DOS executable header. It is followed by a small program that prints an error message saying that the program cannot be run in DOS mode. It's the same idea that was used in 16-bit Windows executables. This program code is executed, if the PE image is run under DOS.

After the DOS header and the DOS executable stub comes the PE header. A field in the DOS header points to this new header. The PE header starts with the 4-byte signature "PE" followed by two nulls. The PE format is based on the Common Object

File Format (COFF) used by Unix. The PE signature is followed by the object file header borrowed from COFF. This header is present also for the object files produced by Microsoft's 32-bit compilers. This header contains some general information about the file, such as the target machine ID, the number of sections in the file, and so forth. The COFF style header is followed by the optional header. This header is optional in the sense that it is not required for the object files. As far as executables and DLLs are concerned, this header is mandatory. The optional header has two parts. The first part is inherited from COFF and can be found in all COFF files. The second part is an NT-specific extension of COFF. Apart from other NT-specific information, such as the subsystem type, this part also contains the data directory. The data directory is an array in which each entry points to some important piece of information. One of the entries in the data directory points to the import table of the executable or DLL, another entry points to the export table of the DLL, and so on.

We will look at the detailed formats of the different pieces of information later in this chapter.

The data directory is followed by the section table. The section table is an array of section headers. A section header summarizes the important information about the respective section. Finally, the section table is followed by the sections themselves.

We hope that this gives you an overview of the organization of a PE file. Before diving into the details of the PE format, let's discuss a concept that is vital in interpreting a PE file. . . . .

Relative Virtual Address

All the offsets within a PE file are denoted as Relative Virtual Addresses (RVAs). An RVA is an offset from the base address at which an executable is loaded in memory. This is not the same as the offset within the file because of the section alignment requirements. The PE header specifies the section alignment requirements for an executable image. A section has to be loaded at a memory address that is a multiple of the section alignment. The section alignment has to be a multiple of the page size. This is because different sections have different page attribute requirements; for example, the .data section needs read-write permissions, while the .text section needs read-execute permissions. Hence, a page cannot span section boundaries.

Because the PE format always talks in terms of RVAs, it's difficult to find the location of the required information within a file. A common practice while accessing a PE file is to map the file in memory using the Win32 memory mapping API. It's a bit complicated to calculate the address for the given RVA in this

memory-mapped file. You first need to find out the section in which the given RVA lies. You can accomplish this by iterating through the section table. Each section header stores the starting RVA for the section and the size of the section. A section is guaranteed to be contiguously loaded in memory. Hence, the offset from the start of the section for a particular piece of data is bound to be the same whether the file is memory mapped or loaded by the operating system loader for execution. Hence, to find out the address in a memory-mapped file, you simply need to add this offset to the base address of the section in the memory-mapped file. Now, this base address can be calculated from within the file offset of the section, which is also stored in the respective section header. Quite an easy procedure, isn't it?

ImageRvaToVaQ

Don't worry, there is an easier way out. Microsoft comes to our rescue here with IMAGEHLP.DLL. This DLL exports a function that computes the address in the memory-mapped file, given an RVA.

```
LPVOID ImageRvaToVa(
PIMAGE_NT_HEADERSNtHeaders,
LPVOID Base,
DWORD  Rva,
PIMAGE_SECTION_HEADER*LastRvaSection
);
```

PARAMETERS

NtHeaders	Pointer to an IMAGE_NT_HEADERS structure. This structure represents the PE header and is defined in the WINNT.h file. A pointer to the PE header within a PE file can be obtained using the ImageNtHeader() function exported by IMAGEHLP.DLL.
Base	Base address where the PE file is mapped into memory using the Win32 API for the memory mapping of files.
Rva	Given relative virtual address.
LastRvaSection	Last RVA section. This is an optional parameter, and you can pass NULL. When specified, it points to a variable that contains the last section value used for the specified image to translate an RVA to a VA. This is used for optimizing the section search, in case the given RVA also falls within the same section as the one for the previous call to the function. The LastRVASection is checked first, and the regular sequential search for the section is carried out only if the given RVA does not fall within the LastRVASection.

RETURN VALUES
If the function succeeds, the return value is the virtual address in the mapped file; otherwise, it is NULL. The error number can be retrieved using the GetLastError() function.

ImageNtHeader()

The ImageRvaToVaQ function needs a pointer to the PE header. The ImageNtHeader exported from the IMAGEHLP.DLL can provide you this pointer.

```
PIMAGE_NT_HEADERS ImageNtHeader(
LPVOID  ImageBase
);
```

PARAMETERS

ImageBase	Base address where the PE file is mapped into memory using the Win32 API for the memory mapping of files.

RETURN VALUES
If the function succeeds, the return value is a pointer to the IMAGE_NT_HEADERS structure within the mapped file; otherwise, it returns NULL.

MapAndLoad()

The IMAGEHLP.DLL can also take care of memory mapping a PE file for you. The MapAndLoad() function maps the requested PE file in memory and fills in the LOADED_IMAGE structure with some useful information about the mapped file.

```
BOOL  MapAndLoad(
LPSTR ImageName,
LPSTR DllPath,
PLOADED_IMAGE LoadedImage.
BOOL  DotDll.
BOOL  Readonly
);
```

PARAMETERS

ImageName	Name of the PE file that is loaded.
DllPath	Path used to locate the file if the name provided cannot be found. If NULL is passed, then normal rules for searching using the PATH environment variable are applied.

LoadedImage	The structure LOADEDJMAGE is defined in the IMAGEHLP.H file. The structure has the following members:
ModuleName	Name of the loaded file.
hFile	Handle obtained through the call to CreateFile.
MappedAddress	Memory address where the file is mapped.
FileHeader	Pointer to the PE header within the mapped file.
LastRvaSection	The function sets it to the first section (see ImageRvaToVa).
NumberOfSections	Number of sections in the loaded PE file.
Sections	Pointer to the first section header within the mapped file.
Characteristics	Characteristics of the PE file (this is explained in more detail later in this chapter).
fSystemImage	Flag indicating whether it is a kernel-mode driver/DLL.
fDOSImage	Flag indicating whether it is a DOS executable.
Links	List of loaded images.
SizeOfImage	Size of the image.

The function sets the members in the structure appropriately after loading the PE file.

DotDll	If the file needs to be searched and does not have an extension, then either the .exe or the .dll extension is used. If the DotDll flag is set to TRUE, the .dll extension is used; otherwise, the .exe extension is used.
Readonly	If the flag is set to TRUE, the file is mapped as read-only.

RETURN VALUES
If the function succeeds, the return value is TRUE; otherwise, it is FALSE.

UnMapAndLoad()

After you are done with the mapped file, you should call the UnMapAndLoadO function. This function unmaps the PE file and deallocates the resources allocated by the MapAndLoad() function.

```
BOOL UnMapAndLoad(
PLOADED_IMAGE LoadedImage
);
```

PARAMETERS

LoadedImage Pointer to a LOADEDJMAGE structure that is returned from a call to the MapAndLoad() function.

RETURN VALUES

If the function succeeds, the return value is TRUE; otherwise, it is FALSE.

We will discuss the other useful functions from this DLL as we continue in this chapter.

Details of the PE Format

The WINNT.H file has the structure definitions representing the PE format. We refer to these structure definitions while describing the PE format. Let's begin at the beginning. The DOS header that comes at the beginning of a PE file does not contain much important information from the PE viewpoint. The fields in this header have values pertaining to the DOS executable stub that follows this header. The only important field as far as PE format is considered is e_lfanew, which holds the offset to the PE header. You can add this offset to the base of the memory-mapped file to get the address of the PE header. You can also use the ImageNtHeaderO function explained earlier, or simply use the FileHeader field from the LOADEDJMAGE after a call to the MapAndLoad() function.

The IMAGE_NT_HEADERS structure that represents the PE header is defined as follows in the WINNT.H file:

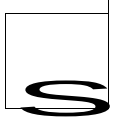
```
typedef struct _IMAGE_NT_HEADERS {
DWORD Signature;
IMAGE_FILE_HEADER FileHeader;
IMAGE_OPTIONAL_HEADEROptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

The signature is PE followed by two nulls, as mentioned earlier. The COFF style header is represented by the IMAGE_FILE_HEADER structure and is followed by the optional header represented by the IMAGE_OPTIONAL_HEADER structure. The fields in the COFF style header are as follows:

MachineTarget machine ID. Various values are defined in the WINNT.H file — for example, Oxl4C is used for Intel 80386 (and compatibles) and 0x184 is used for Alpha AXP.

NumberOfSections	Number of sections in the file.
TimeDateStamp	Time and date when the file was created.
PointerToSymbolTable	Offset to the COFF symbol table. This field is used only for COFF style object files and PE files with COFF style debug information.
NumberOfSymbols	Number of symbols present in the symbol table.
SizeOfOptionalHeader	Size, in bytes, of the optional header that follows this header. This data can be used in locating the string table that immediately follows the symbol table. This field is set to 0 for the object files because the optional header is absent in them.
Characteristics	Attributes of the file. The flag values are defined m the WINNT.H file. This field contains an OR of these flags. The important flags are as follows:
IMAGE_FILE_EXECUTABLE_IMAGE	Set for an executable file.
IMAGE_FILE_SYSTEM	Indicates that it is a kernel-mode driver/DLL.
IMAGE_FILE_DLL	The file is a dynamic link library (DLL).
IMAGE_FILE_UP_SYSTEM_ONLY	This file should be run only on an UP machine.
IMAGE_FILE_LINE_ NUMS_STRIPPED	Indicates that the COFF line numbers have been removed from the file.
IMAGE_FILE_LOCAL_ SYMS_STRIPPED	Indicates that the COFF symbol table has been removed from the file.
IMAGE_FILE_DEBUG_STRIPPED	Indicates that the debugging information has been removed from the file.
IMAGE_FILE_RELOCS_STRIPPED	Indicates that the base relocation information is stripped from this file, and the file can be loaded only at the preferred base address. If the loader cannot load such an image at the preferred base address, it fails because it cannot relocate the image.

IMAGE_FILE_AGGRESIVE_ WS_TRIM	Aggressively trim working set.
IMAGE_FILE_BYTES_ REVERSED_LO	Little endian: the least significant bit (LSB) precedes the most significant bit (MSB) in memory, but they are stored in reverse order.
IMAGE_FILE_BYTES_ REVERSED_HI	Big endian: the MSB precedes the LSB in memory, but they are stored in reverse order.
IMAGE_FILE_32BLT_MACHINE	The target machine is based on 32-bit-word architecture.
IMAGE_FILE_REMOVABLE_ RUN_FROM_SWAP	If this flag is set and the file is run from a removable media, such as a floppy, the loader copies the file to the swap area and runs it from there.
IMAGE_FILE_NET_RUN_ FROM_SWAP	Similar to the previous flag. It is run from swap if the file is run from a network drive.



The COFF style header is followed by the optional header. The optional header is absent in the object files. The format of the optional header is defined as the IMAGE OPTIONAL HEADER structure in theWINNT.H file. The first few fields in this structure are inherited from COFF. •; -- - " - -

Magic	This field is set to Ox lOb for a normal executable/DLL.
MajorLinkerVersion, MinorLinkerVersion	Version of the linker that produced the file.
SizeOfCode	Size of the code section. If there are multiple code sections, this field contains the sum of sizes of all these sections.
SizeOflnitializedData	Size of the initialized data section. If there are multiple initialized data sections, this field contains the sum of sizes of all these sections.
SizeOfUninitializedData	Same as SizeOflnitializedData, but for the uninitialized data (BSS) section.

AddressOfEntryPoint	RVA of the entry point.
BaseOfCode	RVA of the start of the code section.
BaseOfData	RVA of the start of the data section.

Microsoft added some NT-specific fields to the optional header. These fields are as follows:

ImageBase — If the file is loaded at this address in memory, the loader need not do any base relocations. This is because the linker resolves all the base relocations at the time of linking, assuming that the file will be loaded at this address. We discuss this in more detail in the section on the relocation table. For now, it is enough to know that the loading time is reduced if a file gets loaded at the preferred base address. A file may not get loaded at the preferred base address because of the nonavailability of the address. This happens when more than one DLL used by an executable use the same preferred base address. The default preferred base address is 0x400000. You may want to have a different preferred base address for your DLL so that it does not clash with that of any other DLL used by your application. You can change the preferred base address using a linker switch. You can also change the base address of a file using the rebase utility that comes with the Win32 SDK.

ReBaseImage()

The ReBaseImageO function from the IMAGEHLP.DLL also enables you to change the preferred base address.

```
BOOL ReBaseImage(
LPSTR CurrentImageName,
LPSTR SymbolPath,
BOOL fReBase,
BOOL fRebaseSysfileOk,
BOOL fGoingDown,
DWORD CheckImageSize,
LPDWORD OldImageSize,
LPDWORD OldImageBase,
LPDWORD NewImageSize,
LPDWORD NewImageBase,
DWORD TimeStamp
);
```

PARAMETERS

CurrentlmageName	Filename that is rebased.
SymbolPath	In case the symbolic debug information is stored as a separate file, the path to find the corresponding symbol file. This is required to update the header information and timestamp of the symbol file.
fReBase	The file is really rebased only if this value is TRUE.
fRebaseSysfileOk	If the file is a system file with the preferred base address above 0x80000000, it is rebased only if this flag is TRUE.
fGoingDown	If you want the loaded image of the file to lie entirely below the given address, set this flag to TRUE. For example, if the loaded size of a DLL is 0x2000 and you call the function with the fGoingDown flag as TRUE and give the address as 0x600000, the DLL will be rebased at 0x508000.
ChecklmageSize	Rebasing might change the loaded image size of the file because of the section alignment requirements. If this parameter is nonzero, the file is rebased only if the changed size is less than this parameter.
OldlmageSize	Original image size before the rebase operation is returned here.
OldlmageBase	Original image base before the rebase operation is returned here.
NewImageSize	New loaded image size after the rebase operation is returned here.
NewImageBase	New base address. Upon return, it contains the actual address where the file is rebased.
TimeStamp	New timestamp for the file.

RETURN VALUES

If the function succeeds, the return value is TRUE; otherwise, it is FALSE.

The other fields in the optional header are as follows:

SectionAlignment	A section needs to be loaded at an address that is a multiple of the section alignment. Refer to the discussion on RVA for more information.
FileAlignment	In the file, a section always starts at an offset that is a multiple of the file alignment. This value is some multiple of the sector size.
MajorOperatingSystemVersion, MinorOperatingSystemVersion	Minimum operating system version required to execute this file.
MajorlmageVersion, MinorlmageVersion	A developer can use these fields to version his or her files. It can be specified with a linker flag.
Maj orSubsystemVersion, MinorSubsystemVersion	Minimum subsystem version required to execute this file.
Win32VersionValue	Reserved for future use.
SizeOflmage	Size of the image after considering the section alignment. This amount of virtual memory needs to be reserved for loading the file.
SizeOfHeaders	Total size of the headers, including the DOS header, the PE header, and the section table. The sections containing the actual data start at this offset in the file.
Checksum	This is used only for the kernel-mode drivers/DLLs. It can be set to 0 for user-mode executables/DLLs.
Subsystem	Subsystem used by the file. The following values are defined in the WINNT.H file:
IMAGE_SUBSYSTEM_NATIVE	Image doesn't require a subsystem. The kernel-mode drivers and native applications such as CSRSS.EXE have this value for the field.
IMAGE_SUBSYSTEM_ WINDOWS_GUI	File uses the Win32 GUI interface.
IMAGE_SUBSYSTEM_ WINDOWS_CUI	File uses the character-based user interface.
IMAGE_SUBSYSTEM_OS2_CUI	File requires the OS/2 subsystem.

IMAGE_SUBSYSTEM_ POSIX_CUI	File uses the POSIX API.
DllCharacteristics	Obsolete.
SizeOfStackReserve	Address space to be reserved for the stack. Only the virtual address space is marked - the swap space is not allocated.
SizeOfStackCommit	Actual memory committed for the stack. This much swap space is initially allocated. The committed stack size is increased on demand until it reaches the SizeOfStackReserve.
SizeOfHeapReserve	Address space to be reserved for the heap. Similar to the SizeOfStackReserve field.
SizeOfHeapCommit	Actual committed heap space. Similar to the SizeOfStackCommit field.
LoaderFlags	Obsolete.
NumberOfRvaAndSizes	Number of entries in the data directory that follows this field. It is always set to 16.
DataDirectory[TMAGE_ NUMBEROF_DIRECTORY _ENTRIES]	As mentioned earlier, each entry in the data directory points to some important piece of information. Each of these entnes is of the type IMAGE_DATA_DIRECTORY, which is defined as follows:

```
typedef struct _IMAGE_DATA_DIRECTORY {
DWORD   VirtualAddress;
DWORD    Size;
) IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

ImageDirectoryEntryToData()

The VirtualAddress field contains the RVA of the respective piece of information, and the Size field contains the size of the data. To get to the actual data, you need to convert the RVA to the actual address in the memory-mapped PE file. This can be accomplished with the ImageDirectoryEntryToDataO function exported by IMAGEHLP.DLL.

```
PVOID ImageDirectoryEntryToData(
    LPVOID Base,
    BOOLEAN MappedAsImage,
    USHORT DirectoryEntry,
    PULONG Size
);
```

PARAMETERS ?t»-

Base	Base address where the file is mapped in memory.
MappedAsImage	Set this flag to TRUE if the system loader maps the file. Otherwise, set the flag to FALSE.
DirectoryEntry	Index into the data directory array. -« - V .. "
Size	Upon return, the size from the data directory is filled here.

RETURN VALUES
If the function succeeds, the return value is the address in the memory-mapped file where the required data resides. Otherwise, the function returns NULL.

Indices in the Data Directory

Each index in the data directory (except a few at the end that are still unused) represents some important piece of information. In the following sections, we discuss some of the important entries in this directory and the format in which the respective information is stored.

Export Directory

The data directory entry at the IMAGE_DIRECTORY_ENTRY_EXPORT index points to the export directory for the file. The RVA in this directory entry points to the .edata section. The information about the functions exported by the file (generally a DLL) is stored here. The data directory entry points to the export directory that is defined as the IMAGE_EXPORT_DIRECTORY structure in the WINNT.H file. The fields in this structure are as follows:

Characteristics	Reserved field. Always set to 0.
TimeDateStamp	Date and time of creation.

MajorVersion, MinorVersion	Developer can set the version of the export table.
Name	RVA of the zero-terminated name of the DLL.
Base	Starting ordinal for the exported functions - that is, the least of the ordinals. Generally, this field is 1.
NumberOfFunctions	Total number of functions exported from the DLL.
NumberOfNames	Number of functions that are exported by name. Some functions may be exported only by ordinal, so this number may be less than NumberOfFunctions.
AddressOfFunctions	RVA of an array (let's call it as the export-functions array) that has an entry for each function exported from the DLL. Hence, the size of this array is equal to the NumberOfFunctions field. The entry at index *i* corresponds to the function exported with ordinal *i* + Base. Each entry in this array is also an RVA. If the RVA for a particular array entry points within the export section, then it is a forwarder. *Forwarder* means that the function is not present in this DLL, but it is a forwarder reference to some function in another DLL. In such a case, the RVA points to an ASCIIZ string that stores the name of the other DLL and the function name separated by a period. In case the target DLL exports the function by ordinal, the function name is formed as # followed by the ordinal printed in decimal. For example, the KERNEL32.DLL for Windows NT forwards the HeapAllocO function to the RtlAllocateHeap() function in the NTDLL.DLL. Hence, the corresponding RVA in this case points to a location within the export section that holds the string NTDLLRtlAllocateHeap. The Win32 applications can import the HeapAllocO function from the KERNEL32.DLL without worrying about all these details. When the application runs on Windows 95, the loader resolves the import reference to the function in the KERNEL32.DLL. When the same application runs on Windows NT, the loader finds that the function is forwarded to the NTDLL.DLL. Hence, the loader automatically loads the NTDLL.DLL and resolves the imported function to the RtLAllocateHeapO function.

When an export-functions array entry is not a forwarder - that is, the RVA does not lie within the export section - the RVA points to the entry point of the function or to the location of the exported variable.

The export-functions array may have gaps. This is beacause some ordinals might be left unused while exporting functions, and some ordinals might not have any corresponding export. In such a case, the corresponding array entry is set to 0.

AddressOfNames	RVA of an array called as the export-names array that has an entry for every function that is exported by name. Hence, the size of this array is equal to the NumberOfNames field. Each entry in this array is an RVA pointing to an ASCIIZ string containing the export name. The array is sorted on the lexical order so as to allow binary search.
AddressOfNameOrdinals	RVA of an array of ordinals henceforth called as the export-ordinals array. This array has the size same as that of the AddressOfNames array. All three arrays, namely, export-names, export-ordinals, and export-functions, are instrumental in resolving imports by name. For resolving an import by name, the loader first searches the name in the export-names array. If the name matches an entry with index *i*, the *i*th entry in the export-ordinals array is the ordinal of the function. Finally, the address of the function can be found from the export-functions array.

Import Directory

The next index in the data directory, IMAGE_DIRECTORY_ENTRY_IMPORT, is reserved for the import directory of an executable/DLL. The RVA in this data directory entry points to the import directory, which is nothing but a variable-sized array of IMAGE_IMPORT_DESCRIPTORs, one for each imported DLL. The first field in this structure is a union. If the Characteristics field in this union is 0, it indicates the end of the variable-sized import descriptors array. Otherwise, the union is interpreted using the other member, OriginalFirstThunk.

OriginalFirstThunk	This is an RVA of what Microsoft calls as the Import Lookup Table (ILT). Each entry in the ILT is a 32-bit number. If the MSB of this number is set, it is treated as an import by ordinal. The bits 0 through 30 are treated as the ordinal of the imported function. If the MSB is not set, the number is treated as an RVA to the IMAGE IMPORT BY NAME structure. The first member

of this structure is a hint for searching for the imported name in the export directory of the imported DLL. The loader uses this hint as the starting index in the export-names array when it does a binary search while resolving the import reference. The hint is followed by an ASCIIZ name of the import reference.

The WINNT.H file provides the IMAGE_SNAP_BY_ORDINAL macro to determine whether it's an import by ordinal. It also provides the IMAGE_ORDINAL macro to get the ordinal from the 32-bit number in the ILT. The ILT is a variable-sized array. The end of the ILT is marked with a 0.

The other members in the IMAGE IMPORT_DESCRLPTOR structure are as follows:

TimeDateStamp	This field is set to 0, unless the imports are bound. Soon, we discuss what's meant by binding the imports of a PE file.
ForwarderChain	The field is used only if the imports are bound.
Name	RVA of the ASCIIZ string that stores the name of the imported DLL.
FirstThunk	RVA of the Import Address Table (LAT). The IAT is another array parallel to the ILT, unless the image is bound. The IAT also has ordinals or pointers to the IMAGE_IMPORT_BY_NAME structures. When the loader resolves the import references, it replaces the entries in the IAT with the actual addresses of the corresponding functions. Astonishingly, that is all it needs to do to achieve dynamic linking — everything else is already set in place by the linker and import librarian. Let's see how all these components work together to achieve dynamic linking.

DYNAMIC LINKING WITH PE FILES

Every DLL has an import library that can either be created using an import librarian or may be generated by the linker itself while creating the DLL. The import library has stub functions with names the same as those of the functions exported from the DLL. The import library also has a .idata section containing an import table that has entries for all the functions from the DLL. Each stub function is an indirect jump that refers to the appropriate entry in the LAT in the .idata section. When an executable is linked with the import library, the linker resolves the imported function calls to the stub functions in the import library. The linker also concatanates the .text section from the import library that contains the stub functions with the .text section of the generated executable. The .idata sections and, incidentally, the import directories are also concatenated. The stage is now set for loading.

While loading, the entries in the IAT are replaced by the actual function addresses, and that's it. Now when the function is called, the control is transferred to the stub function that performs an indirect jump. As the IAT entry contains the address of the actual function from the DLL, the control is transferred to the required function.

The situation is a bit different if you use the new __declspec(dllimport) directive while prototyping an imported function. In that case, the compiler itself generates an import table. In addition, it generates an indirect call referring to the appropriate location in the generated IAT. This method does away with the overhead of an extra jump.

BINDING IMPORTS FOR A PE FILE

A major portion of loading time is spent on resolving the imports. The loader has to search each imported symbol in the export directory of the imported DLL to find out the virtual address of the symbol. The loading time can be drastically reduced if the IAT contains the actual address of the symbol instead of the name or ordinal. Such a PE file is called as a bound image. The imported symbol addresses are calculated assuming that the imported DLL will be loaded at the preferred base address at the time of loading. The IMAGE_IMPORT_DESCRIPTORs, in a bound PE file, are also modified. The TimeDateStamp field stores the timestamp of the imported DLL. At the time of loading, if this timestamp does not match with that of the DLL, the imports need to be resolved again. Because the IAT is modified and does not contain the symbol names or ordinals, the ILT is used, in this case, to resolve the imports.

The forwarded functions pose another problem with binding. The addresses of the forwarded functions cannot be calculated at bind time, and so these functions have to be resolved at load time. A list of all the forwarded functions for an imported DLL is maintained through the ForwarderChain member in the corresponding IMAGE_IMPORT_DESCRIPTOR. This member stores the index of a forwarded function in the IAT. The IAT entry at this index stores the index of the next forwarded function, and so on, forming a list of forwarded functions. The list is terminated by a -1 entry.

BindImage()

The bind utility that is shipped with Win32 SDK enables binding of PE files. Also, the BindImage and BindImageExO functions in the IMAGEHLP.DLL provide this functionality.

```
BOOL  BindImage(
LPSTR ImageName,
LPSTR DllPath,
LPSTR Symbol Path
);
```

PARAMETERS

ImageName	The filename of the file to be bound. This can contain only a filename, a partial path, or a full path.
DllPath	A root path to search for ImageName if the filename contained in ImageName cannot be opened.
SymbolPath	A root path to search for the corresponding symbol file. If the symbol file is stored separately, the header of the symbol file is changed to reflect the changes in the PE file. -

RETURN VALUES

If the function succeeds, the return value is TRUE; otherwise, it is FALSE.

BindImageEx()

This function is very similar to BindImage function except it provides more customization such as getting a periodic callback during the progress of binding process.

```
BOOL BindImageEx(
    IN DWORD Flags,
    IN LPSTR ImageName,
    IN LPSTR DllPath,
    IN LPSTR SymbolPath,
    IN PIMAGEHLP_STATUS_ROUTINE StatusRoutine
);
```

PARAMETERS

This function has the following additional parameters:

Flags	The field controls the behavior of the function. It is set to as an OR of the flag values defined in the IMAGEHLP.H file. The following flag values are defined in the IMAGEHLP.H file:
BIND_NO_BOUND_IMPORTS	Do not generate a new import address table.
BIND_NO_UPDATE	Do not make any changes to the file.
BIND_ALL_IMAGES	Bind all images that are in the call tree for this file.
StatusRoutine	Pointer to a status routine. The status routine is called during the progress of the image binding process.

RETURN VALUES
If the function succeeds, the return value is TRUE; otherwise, it is FALSE.

Calling BindImage is equivalent to calling BindImageEx with Flags as 0 and StatusRoutine as NULL. That is, calling BindImage(ImageName, DllPath, SymbolPath) is equivalent to calling BindImageEx(0, ImageName, DllPath, SymbolPath, NULL).

Resource Directory

The next index in the data directory, IMAGE_DIRECTORY_ENTRY_RESOURCE, refers to the resource directory for a PE file. The resource directory and the resources themselves are generally stored in a section named .rsrc section. The resources are maintained in a tree structure similar to that in a file system. The root directory contains subdirectories. A subdirectory can contain subdirectories or resource data. The subdirectories can be nested to any level. But Windows NT only uses a three-level structure. At each level, the resource directory branches according to certain characteristics of the resources. At the first level, the type of the resource - bitmap, menu, and so on - is considered. All the bitmaps are stored under one subtree, all the menus are stored under another subtree, and so on. At the next level, the name of the resource is considered, and the third level classifies the resource according to the language ID. The third-level resource directory points to a leaf node that stores the actual resource data.

A resource directory consists of summary information about the directory followed by the directory entries. Each directory entry has a name or ID that is interpreted as a type ID, a name ID, or a language ID, depending on the level of the directory. A directory entry can point either to the resource data or to a subdirectory that has a similar format.

The format of the resource directory is defined as the IMAGE_RESOURCE_DIRECTORY structure in WINNT.H.

Characteristics	Currently unused. Set to 0.
TimeDateStamp	Date and time when the resource was generated by the resource compiler.
MajorVersion, MinorVersion	Can be set by the user.
NumberOfNamedEntries	Number of directory entries having string names. These entries immediately follow the directory summary information and are sorted.
NumberOfIdEntries	Number of directory entries that use integer IDs as the names. These entries follow the ones having string names.

This summary information is followed by the directory entries. Each directory has a format as defined by the IMAGE_RESOURCE_DIRECTORY_ENTRY structure in WINNT.H. This structure is composed of two unions. The first union stores the ID of the entry. If the MSB is set, then the lower 31 bits in this field is an RVA of the Unicode string that stores the name of the entry. The Unicode string consists of the length of the string followed by the 16-bit Unicode characters. If the MSB is not set, then the union stores the integer ID of the resource. This first union stores the type ID, the name ID, or the language ID, depending on the level of the directory. The second union, in the IMAGE_RESOURCE_DIRECTORY_ENTRY structure, points either to another resource directory or to the resource data, depending on the MSB. If the bit is set, the lower 31 bits is an RVA of another subdirectory. If the MSB is not set, then it's an RVA of the resource data entry that forms a leaf node of the resource directory tree structure. The format of the resource data entry is defined as the IMAGE_RESOURCE_DATA_ENTRY structure in the WINNT.H file and has following members:

OffsetToData	RVA of the actual resource data.
Size	Size of the resource data.
CodePage	Code page used to decode code point values within the resource data. Typically, the code page would be the Unicode code page.

Relocation Table

A PE file needs only based relocations. The linker resolves all the relative relocations, assuming that the file will get loaded at the preferred base address. For example, if a function foo has the RVA as 0x100 and the preferred base address is 0x400000, the linker resolves the call to foo as a call to address 0x400100. At run time, if the file is loaded at the preferred base address of 0x400000, then no relocation needs to be preformed. If, for some reason, the file cannot be loaded at the base address of 0x400000, the loader needs to patch the call. If the loader manages to load the file at a base address of 0x600000, it needs to change the call address to 0x600100. In general, it needs to add the difference of 0x200000 to all the to-be-patched locations. This process is called as the based relocation. The list of the to-be-patched locations, also called as fixups, is maintained in the relocation table that is generally present in the .reloc section and is pointed to by the data directory entry at the IMAGE_DIRECTORY_ENTRY_BASERELOC index. The relocation table is nothing but a series of relocation blocks, each representing the fixups for a 4K page. Each relocation block has a header followed by the relocation entries for the corresponding page. The relocation block format is defined as the IMAGE_BASE_RELOCATION structure in the WINNT.H file, and it has following fields:

VirtualAddress	RVA of the page to be patched.
SizeOfBlock	Total size of the relocation block, including the header and the relocation entries.

Each relocation entry is a 16-bit word. The higher 4 bits indicate the type of relocation, and the lower 12 bits are the offset of the fixup location within the 4K page. The address-to-patched is calculated by adding the base address for loading, the RVA of the page to be patched, and the 12-bit offset within the page. The relocation types are defined in the WINNT.H file - only two of them are used on Intel machines:

IMAGE_REL_BASED_ABSOLUTE	The relocation is skipped. This type can be used to pad a relocation block so that the next block starts at a 4-byte boundary.
IMAGE_REL_BASED_fflGHLOW	The relocation adds the base-address difference to the 32-bit double word at the location denoted by the 12-bit offset.

Debug Directory

The operating system is not concerned with the debug information present in a PE file. The debugging tools access the debug information in a PE file. There are various debugging tools, which expect the debug information in different formats. The corresponding compilers/linkers also store the debug information in different formats. The PE format allows the debug information to be stored in different formats, such as COFF, Frame Pointer Omission (FPO), CodeView (CV4), and so on. A single file may contain debug information in more than one format. The debug directory pointed to by the IMAGE_DIRECTORY_ENTRY_DEBUG entry in the data directory is an array of debug directory entries, one for each debug information format. The IMAGE_DEBUG_DIRECTORY structure in the WINNT.H file represents the format of a debug directory entry.

Characteristics	Currently unused. Set to 0.
TimeDateStamp	Date and time when the debug data was created.
MajorVersion, MinorVersion	Version of the debug data format.
Type	Type of the debug data format.
SizeOfData	Size of the debug data.
AddressOfRawData	RVA of the debug data.
PointerToRawData	Within file offset to the debug data.

Of the different debug information formats, three are frequently encountered in PE files. The first one is the format used by the popular CodeView debugger. This format is defined in the CV4 specification. The FPO format is used to describe non-standard stack frames. Not all the files in a PE file need have an FPO format debug entry. The functions without one are assumed to have a normal stack frame. The third important format is COFF, which is the native debug information format for PE files. The PE header itself points to the COFF symbol table. The COFF debug information consists of symbols and line numbers.

Thread Local Storage

The threads executing in a process share the same global data space. Sometimes, it may be required that each thread has some storage local to itself. For example, say a variable *i* needs to be local for each thread.

In such a case, each thread gets a private copy of *i*. Whenever a particular thread is running, its own private copy of *i* should be automatically activated. This is achieved in Windows NT using the Thread Local Storage (TLS) mechanism. Let's see how it works.

Do not confuse the local data of a thread with the local variables that are created on stack. Each thread has a separate stack and local variables that are created and destroyed separately for each thread as the stack grows and shrinks. In this section, the phrase *local data* means global variables that have a separate copy for each thread.

The operating system maintains a structure called as the Thread Environment Block (TEB) for every thread running in the system. The FS segment register is always set such that the address FS:0 points to the TEB of the thread being executed. The TEB contains a pointer to the TLS array. The TLS array is an array of 4-byte DWORDs. Similar to the TEB, a separate TLS array is present for each thread. A thread can store its local data in the TLS array. Generally, programs store pointers to local data in some slot in the TLS array. The slot allocation for the TLS array is controlled by the API functions TlsAlloc() and TlsFree(). The Win32 API also provides functions to set and get the value at a particular index in the TLS array.

It is cumbersome to access the thread-specific data using the API functions. An easier way is to use the _declspec(thread) specification while declaring global variables that need to have a private copy for each thread. All such variables are gathered by the compiler/linker, and a single TLS array index is automatically allotted to this bunch of data. The TLS array entry at this index contains the pointer to a local data buffer that stores all these variables. These variables are accessed as any other normal variable in the program. Whenever such a variable is accessed, the compiler takes care to generate the code to access the TLS array entry and the data at a proper offset within the local data buffer.

This discussion is bit off the track. However, it is necessary before discussing the IMAGE_DIRECTORY_ENTRY_TLS data directory entry. The TLS directory structure is defined as IMAGE_TLS_DIRECTORY in the WINNT.H. Let's have a look at this structure and see how it fits in the TLS mechanism.

StartAddressOfRawData	Each time a new thread is created, the operating system allocates a new local data buffer for the thread and initializes the buffer with the data that is pointed to by this field. Note that this address is not an RVA, but it is a proper virtual address that has a relocation entry in the .reloc section.
EndAddressOfRawData	Virtual address of the end of the initialization data. The rest of the local data buffer is filled with zeros.
AddressOfIndex	Address in the data section where the loader should store the automatically allotted TLS index. The code accessing TLS variables accesses the index from this location.
AddressOfCallBacks	Pointer to a null-terminated array of TLS callback functions. Each function in this array is called whenever a new thread is created. These functions can perform additional initialization (for example, calling constructors) for the TLS data. The TLS callback has the same parameters as the DLL entry-point function.
SizeOfZeroFill	Size of the local data that is to be initialized to zero. The total size of the local data is (EndAddressOfRawData StartAddressOfRawData) + SizeOfZeroFill.
Characteristics	Reserved.

Section Table

We've roamed through the PE format without bothering about the section formats. This is possible because of the data directory that directly locates the important pieces of information within a PE file. You need not know about the sections at all to interpret a PE file. Nevertheless, in case you need to modify a PE file, you may be required to know about the sections and section headers. For example, you may want to add, remove, or extend a particular section, and this requires changes to the section table, among other things.

As mentioned earlier, the PE header is followed by the section table. The section table is an array of section headers. The format of the section header is defined by the IMAGE_SECTION_HEADER structure in the WINNT.H file. The members of a section header are as follows:

Name	Character array of size IMAGE_SIZEOF_SHORT_NAME. Contains the name of the section.
VirtualSize	Size of the section.
VirtualAddress	RVA of the section data when loaded in memory.
SizeOfRawData	Size of the section as stored in the file. This is equal to the VirtualSize rounded to the next file alignment multiple.
PointerToRawData	Within file offset to the section data. If you memory map a PE file, this field needs to be used to get to the section data.
PointerToRelocations	Used only in the object files.
PointerToLinenumbers	Within file offset to the COFF style line number information.
NumberOfRelocations	Used only in the object files.
NumberOfLinenumbers	Number of records in the line number information.
Characteristics	The attributes of the section. It is an OR of the section characteristics flags defined in the WINNT.H file. Some of the important flags are as follows:
IMAGE_SCN_CNT_CODE	Section contains executable code.
IMAGE_SCN_CNT_ INITIALIZED_DATA	Section contains initialized data.
IMAGE_SCN_CNT_ UNINITIALIZED_DATA	Section contains uninitialized data.
IMAGE_SCN_LNK_REMOVE	Section will not become part of the loaded image. The .debug section may have this flag set.
IMAGE_SCN_MEM_ DISCARDABLE	Section can be discarded. The relocation table and debug information can be discarded after the loading process is over. Hence, the .debug and .reloc sections have this flag set.
IMAGE SCN_MEM_NOT CACHED	Section cannot be cached.

IMAGE_SCN_MEM_NOT_PAGED	Section is not pageable.
IMAGE_SCN_MEM_SHARED	Section can be shared in memory. If a DLL has the data section with this flag set, all the instances of the DLL in different processes share the same data.
IMAGE_SCN_MEM_EXECUTE	Section can be executed. For the code sections, both the IMAGE_SCN_CNT_CODE and IMAGE_SCN_MEM_EXECUTE flags are set.
TMAGE_SCN_MEM_READ	Section can be read.
IMAGE SCN MEM WRITE	Section can be written to.

Loading Procedure

Let's see how the loader interprets a PE file and prepares a memory image for execution. The loader needs to find the free virtual address space to map the file in memory. The loader tries to load the image at the preferred base address. After this is done, the loader maps the sections in memory. The loader goes through the section table and maps each section at the address calculated by adding the RVA of the section to the base address. The page attributes are set according to the section's characteristic requirements. After mapping the section in memory, the loader performs based relocation if the base address is not equal to the preferred base address. Then, the import table is checked and the required DLLs are loaded. The same procedure for loading an executable - mapping sections, based relocation, resolving imports, and so on - is applied while loading a DLL. After loading each DLL, the IAT is fixed to point to the actual imported function address.

That's it! The image is ready for execution.

Summary

Microsoft introduced the Portable Executable (PE) file format with Windows NT. The PE format serves as the executable file format for all the 32-bit Microsoft operating systems (that is, the various versions of Windows NT and Windows 95/98) though these operating systems still support the older executable file formats, including the DOS executable file format.

Various components in a PE file are addressed using the relative virtual address (RVA). The IMAGEHLP.DLL provides us with utility functions to memory map a PE file to find the address in the memory corresponding to the RVA specified in the PE file. A PE file is composed of the file headers, the data directory, the section table, and the various sections. The data directory points to the important parts of the PE file: the export directory, the import directory, the relocation table, the debug directory, and the Thread Local Storage. The export directory lists the symbols exported from the PE file, which is most likely a DLL. The import directory lists all the symbols imported by the PE file. When a PE file is loaded in memory for execution, the loader resolves the imported symbols to actual virtual addresses in the DLL that exports the symbols. This process is termed dynamic linking.

The PE headers are followed by the section table that points to all the sections, including the ones pointed to by the various data directory entries. The loader reads the section table and maps various sections of a PE file in memory. Then it prepares the image for execution by relocating the image for the mapped address and resolving various imported symbols after loading the required DLLs.