https://en.wikibooks.org/wiki/X86\_Disassembly/Windows\_Executable\_Files

**MS-DOS COM Files**

COM files are loaded into RAM exactly as they appear; no change is made at all from the harddisk image to RAM. This is possible due to the segmented memory model of the early x86 line. Two 16-bit registers determine the actual address used for a memory access, a “segment” register specifying a 64K byte window into the 1M+64K byte space (in 16-byte increments) and an “offset” specifying an offset into that window. The segment register would be set by DOS and the COM file would be expected to respect this setting and not ever change the segment registers. The offset registers, however, were fair game and served (for COM files) the same purpose as a modern 32-bit register. The downside was that the offset registers were only 16-bit and, therefore, since COM files could not change the segment registers, COM files were limited to using 64K of RAM. The good thing about this approach, however, was that no extra work was needed by DOS to load and run a COM file: just load the file, set the segment register, and jump to it. (The programs could perform 'near' jumps by just giving an offset to jump to.)

COM files are loaded into RAM at offset $100. The space before that would be used for passing data to and from DOS (for example, the contents of the command line used to invoke the program).

Note that COM files, by definition, cannot be 32-bit. Windows provides support for COM files via a special CPU mode.

|  |  |
| --- | --- |
| Information | Notice that MS-DOS COM files (short for "command" files) are not the same as *Component-Object Model* files, which are an object-oriented library technology. |

**MS-DOS EXE Files**

One way MS-DOS compilers got around the 64K memory limitation was with the introduction of **memory models**. The basic concept is to cleverly set different segment registers in the x86 CPU (CS, DS, ES, SS) to point to the same or different segments, thus allowing varying degrees of access to memory. Typical memory models were:

tiny

All memory accesses are 16-bit (segment registers unchanged). Produces a .COM file instead of an .EXE file.

small

All memory accesses are 16-bit (segment registers unchanged).

compact

Data addresses include both segment and offset, reloading the DS or ES registers on access and allowing up to 1M of data. Code accesses don't change the CS register, allowing 64K of code.

medium

Code addresses include the segment address, reloading CS on access and allowing up to 1M of code. Data accesses don't change the DS and ES registers, allowing 64K of data.

large

Both code and data addresses are (segment, offset) pairs, always reloading the segment addresses. The whole 1M byte memory space is available for both code and data.

huge

Same as the large model, with additional arithmetic being generated by the compiler to allow access to arrays larger than 64K.

When looking at an EXE file, one has to decide which memory model was used to build that file.

**PE Files**

A **Portable Executable** (**PE**) file is the standard binary file format for an Executable or DLL under Windows NT, Windows 95, and Win32. The Win32 SDK contains a file, *winnt.h*, which declares various structs and variables used in the PE files. Some functions for manipulating PE files are also included in *imagehlp.dll*. PE files are broken down into various sections which can be examined.

**Relative Virtual Addressing (RVA)**

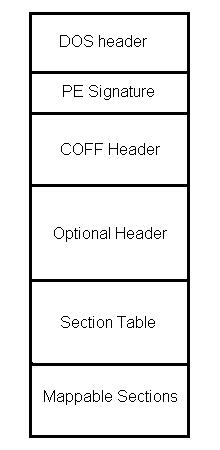
In a Windows environment, executable modules can be loaded at any point in memory, and are expected to run without problem. To allow multiple programs to be loaded at seemingly random locations in memory, PE files have adopted a tool called RVA: Relative Virtual Addresses. RVAs assume that the "base address" of where a module is loaded into memory is not known at compile time. So, PE files describe the location of data in memory as an *offset from the base address*, wherever that may be in memory.

Some processor instructions require the code itself to directly identify where in memory some data is. This is not possible when the location of the module in memory is not known at compile time. The solution to this problem is described in the section on "Relocations".

It is important to remember that the addresses obtained from a disassembly of a module will not always match up to the addresses seen in a debugger as the program is running.

**File Format**

The PE portable executable file format includes a number of informational headers, and is arranged in the following format:

[](https://commons.wikimedia.org/wiki/File:RevEngPEFile.JPG)

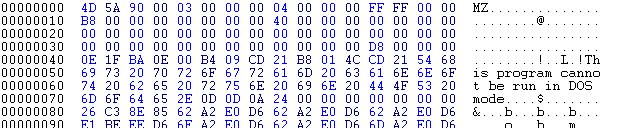
**The basic format of a Microsoft PE file**

**MS-DOS header**

Open any Win32 binary executable in a hex editor, and note what you see: The first 2 letters are **always** the letters "MZ", the initials of Mark Zbikowski, who created the first linker for DOS. To some people, the first few bytes in a file that determine the type of file are called the "magic number," although this book will not use that term, because there is no rule that states that the "magic number" needs to be a single number. Instead, we will use the term "File ID Tag", or simply, File ID. Sometimes this is also known as File Signature.

After the File ID, the hex editor will show several bytes of either random-looking symbols, or whitespace, before the human-readable string "This program cannot be run in DOS mode".

What is this?

[](https://commons.wikimedia.org/wiki/File:RevEngDosHead.JPG)

**Hex Listing of an MS-DOS file header**

What you are looking at is the MS-DOS header of the Win32 PE file. To ensure either a) backwards compatibility, or b) graceful decline of new file types, Microsoft has written a series of machine instructions(an example program is listed below the DOS header structure) into the head of each PE file. When a 32-bit Windows file is run in a 16-bit DOS environment, the program will display the error message: "This program cannot be run in DOS mode.", then terminate.

The DOS header is also known by some as the EXE header. Here is the DOS header presented as a C data structure:

struct DOS\_Header

{

// short is 2 bytes, long is 4 bytes

char signature[2] = "MZ";

short lastsize;

short nblocks;

short nreloc;

short hdrsize;

short minalloc;

short maxalloc;

void \*ss;

void \*sp;

short checksum;

void \*ip;

void \*cs;

short relocpos;

short noverlay;

short reserved1[4];

short oem\_id;

short oem\_info;

short reserved2[10];

long e\_lfanew;

}

After the DOS header there is a stub program mentioned in the paragraph above the DOS header structure. Listed below is a commented example of that program, it was taken from a program compiled with GCC.

;# Using NASM with Intel syntax

push cs ;# Push CS onto the stack

pop ds ;# Set DS to CS

mov dx,message ; point to our message "This program cannot be run in DOS mode.", 0x0d, 0x0d, 0x0a, '$'

mov ah, 09

int 0x21 ;# when AH = 9, DOS interrupt to write a string

;# terminate the program

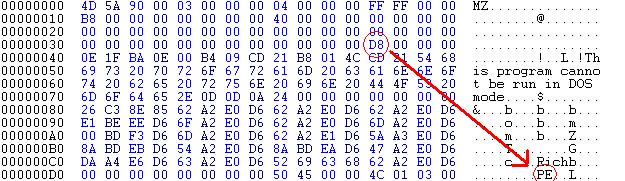
mov ax,0x4c01

int 0x21

message db "This program cannot be run in DOS mode.", 0x0d, 0x0d, 0x0a, '$'

**PE Header**

At offset 60 (0x3C) from the beginning of the DOS header is a pointer to the Portable Executable (PE) File header (e\_lfanew in MZ structure). DOS will print the error message and terminate, but Windows will follow this pointer to the next batch of information.

[](https://commons.wikimedia.org/wiki/File:RevEngPeSig.JPG)

**Hex Listing of a PE signature, and the pointer to it**

The PE header consists only of a File ID signature, with the value "PE\0\0" where each '\0' character is an ASCII NUL character. This signature shows that a) this file is a legitimate PE file, and b) the byte order of the file. Byte order will not be considered in this chapter, and all PE files are assumed to be in "little endian" format.

The first big chunk of information lies in the COFF header, directly after the PE signature.

**COFF Header**

The COFF header is present in both COFF object files (before they are linked) and in PE files where it is known as the "File header". The COFF header has some information that is useful to an executable, and some information that is more useful to an object file.

Here is the COFF header, presented as a C data structure:

struct COFFHeader

{

short Machine;

short NumberOfSections;

long TimeDateStamp;

long PointerToSymbolTable;

long NumberOfSymbols;

short SizeOfOptionalHeader;

short Characteristics;

}

Machine

This field determines what machine the file was compiled for. A hex value of 0x14C (332 in decimal) is the code for an Intel 80386.

Here's a list of possible values it can have.

|  |  |
| --- | --- |
| Value | Description |
| 0x14c | Intel 386 |
| 0x8664 | x64 |
| 0x162 | MIPS R3000 |
| 0x168 | MIPS R10000 |
| 0x169 | MIPS little endian WCI v2 |
| 0x183 | old Alpha AXP |
| 0x184 | Alpha AXP |
| 0x1a2 | Hitachi SH3 |
| 0x1a3 | Hitachi SH3 DSP |
| 0x1a6 | Hitachi SH4 |
| 0x1a8 | Hitachi SH5 |
| 0x1c0 | ARM little endian |
| 0x1c2 | Thumb |
| 0x1d3 | Matsushita AM33 |
| 0x1f0 | PowerPC little endian |
| 0x1f1 | PowerPC with floating point support |
| 0x200 | Intel IA64 |
| 0x266 | MIPS16 |
| 0x268 | Motorola 68000 series |
| 0x284 | Alpha AXP 64-bit |
| 0x366 | MIPS with FPU |
| 0x466 | MIPS16 with FPU |
| 0xebc | EFI Byte Code |
| 0x8664 | AMD AMD64 |
| 0x9041 | Mitsubishi M32R little endian |
| 0xc0ee | clr pure MSIL |

NumberOfSections

The number of sections that are described at the end of the PE headers.

TimeDateStamp

32 bit time at which this header was generated: is used in the process of "Binding", see below.

SizeOfOptionalHeader

this field shows how long the "PE Optional Header" is that follows the COFF header.

Characteristics

This is a field of bit flags, that show some characteristics of the file.

|  |  |
| --- | --- |
| Value | Description |
| 0x02 | Executable file |
| 0x200 | file is non-relocatable (addresses are absolute, not RVA) |
| 0x2000 | File is a DLL Library, not an EXE |

**PE Optional Header**

The "PE Optional Header" is not "optional" per se, because it is required in Executable files, but not in COFF object files. The Optional header includes lots and lots of information that can be used to pick apart the file structure, and obtain some useful information about it.

The PE Optional Header occurs directly after the COFF header, and some sources even show the two headers as being part of the same structure. This wikibook separates them out for convenience.

Here is the PE Optional Header presented as a C data structure:

struct PEOptHeader

{

/\*

char is 1 byte

short is 2 bytes

long is 4 bytes

\*/

short signature; //decimal number 267 for 32 bit, 523 for 64 bit, and 263 for a ROM image.

char MajorLinkerVersion;

char MinorLinkerVersion;

long SizeOfCode;

long SizeOfInitializedData;

long SizeOfUninitializedData;

long AddressOfEntryPoint; //The RVA of the code entry point

long BaseOfCode;

long BaseOfData;

/\*The next 21 fields are an extension to the COFF optional header format\*/

long ImageBase;

long SectionAlignment;

long FileAlignment;

short MajorOSVersion;

short MinorOSVersion;

short MajorImageVersion;

short MinorImageVersion;

short MajorSubsystemVersion;

short MinorSubsystemVersion;

long Win32VersionValue;

long SizeOfImage;

long SizeOfHeaders;

long Checksum;

short Subsystem;

short DLLCharacteristics;

long SizeOfStackReserve;

long SizeOfStackCommit;

long SizeOfHeapReserve;

long SizeOfHeapCommit;

long LoaderFlags;

long NumberOfRvaAndSizes;

data\_directory DataDirectory[NumberOfRvaAndSizes]; //Can have any number of elements, matching the number in NumberOfRvaAndSizes.

} //However, it is always 16 in PE files.

/\*

long is 4 bytes

\*/

struct data\_directory

{

long VirtualAddress;

long Size;

}

Signature

Contains a signature that identifies the image.

|  |  |  |
| --- | --- | --- |
| **Constant Name** | **Value** | **Description** |
| IMAGE\_NT\_OPTIONAL\_HDR32\_MAGIC | 0x10b | 32 bit executable image. |
| IMAGE\_NT\_OPTIONAL\_HDR64\_MAGIC | 0x20b | 64 bit executable image |
| IMAGE\_ROM\_OPTIONAL\_HDR\_MAGIC | 0x107 | ROM image |

MajorLinkerVersion

The major version number of the linker.

MinorLinkerVersion

The minor version number of the linker.

SizeOfCode

The size of the code section, in bytes, or the sum of all such sections if there are multiple code sections.

SizeOfInitializedData

The size of the initialized data section, in bytes, or the sum of all such sections if there are multiple initialized data sections.

SizeOfUninitializedData

The size of the uninitialized data section, in bytes, or the sum of all such sections if there are multiple uninitialized data sections.

AddressOfEntryPoint

A pointer to the entry point function, relative to the image base address. For executable files, this is the starting address. For device drivers, this is the address of the initialization function. The entry point function is optional for DLLs. When no entry point is present, this member is zero.

BaseOfCode

A pointer to the beginning of the code section, relative to the image base.

BaseOfData

A pointer to the beginning of the data section, relative to the image base.

ImageBase

The preferred address of the first byte of the image when it is loaded in memory. This value is a multiple of 64K bytes. The default value for DLLs is 0x10000000. The default value for applications is 0x00400000, except on Windows CE where it is 0x00010000.

SectionAlignment

The alignment of sections loaded in memory, in bytes. This value must be greater than or equal to the FileAlignment member. The default value is the page size for the system.

FileAlignment

The alignment of the raw data of sections in the image file, in bytes. The value should be a power of 2 between 512 and 64K (inclusive). The default is 512. If the SectionAlignment member is less than the system page size, this member must be the same as SectionAlignment.

MajorOSVersion

The major version number of the required operating system.

MinorOSVersion

The minor version number of the required operating system.

MajorImageVersion

The major version number of the image.

MinorImageVersion

The minor version number of the image.

MajorSubsystemVersion

The major version number of the subsystem.

MinorSubsystemVersion

The minor version number of the subsystem.

Win32VersionValue

This member is reserved and must be 0.

SizeOfImage

The size of the image, in bytes, including all headers. Must be a multiple of SectionAlignment.

SizeOfHeaders

The combined size of the following items, rounded to a multiple of the value specified in the FileAlignment member.

* e\_lfanew member of DOS\_Header
* 4 byte signature
* size of COFFHeader
* size of optional header
* size of all section headers

CheckSum

The image file checksum. The following files are validated at load time: all drivers, any DLL loaded at boot time, and any DLL loaded into a critical system process.

Subsystem

The Subsystem that will be invoked to run the executable

|  |  |  |
| --- | --- | --- |
| **Constant Name** | **Value** | **Description** |
| **IMAGE\_SUBSYSTEM\_UNKNOWN** | 0 | Unknown subsystem |
| **IMAGE\_SUBSYSTEM\_NATIVE** | 1 | No subsystem required (device drivers and native system processes) |
| **IMAGE\_SUBSYSTEM\_WINDOWS\_GUI** | 2 | Windows graphical user interface (GUI) subsystem |
| **IMAGE\_SUBSYSTEM\_WINDOWS\_CUI** | 3 | Windows character-mode user interface (CUI) subsystem |
| **IMAGE\_SUBSYSTEM\_OS2\_CUI** | 5 | OS/2 CUI subsystem |
| **IMAGE\_SUBSYSTEM\_POSIX\_CUI** | 7 | POSIX CUI subsystem |
| **IMAGE\_SUBSYSTEM\_WINDOWS\_CE\_GUI** | 9 | Windows CE system |
| **IMAGE\_SUBSYSTEM\_EFI\_APPLICATION** | 10 | Extensible Firmware Interface (EFI) application |
| **IMAGE\_SUBSYSTEM\_EFI\_BOOT\_SERVICE\_DRIVER** | 11 | EFI driver with boot services |
| **IMAGE\_SUBSYSTEM\_EFI\_RUNTIME\_DRIVER** | 12 | EFI driver with run-time services |
| **IMAGE\_SUBSYSTEM\_EFI\_ROM** | 13 | EFI ROM image |
| **IMAGE\_SUBSYSTEM\_XBOX** | 14 | Xbox system |
| **IMAGE\_SUBSYSTEM\_WINDOWS\_BOOT\_APPLICATION** | 16 | Boot application |

DLLCharacteristics

The DLL characteristics of the image

|  |  |  |
| --- | --- | --- |
| **Constant Name** | **Value** | **Description** |
| No constant name | 0x0001 | Reserved |
| No constant name | 0x0002 | Reserved |
| No constant name | 0x0004 | Reserved |
| No constant name | 0x0008 | Reserved |
| **IMAGE\_DLLCHARACTERISTICS\_DYNAMIC\_BASE** | 0x0040 | The DLL can be relocated at load time |
| **IMAGE\_DLLCHARACTERISTICS\_FORCE\_INTEGRITY** | 0x0080 | Code integrity checks are forced |
| **IMAGE\_DLLCHARACTERISTICS\_NX\_COMPAT** | 0x0100 | The image is compatible with data execution prevention (DEP) |
| **IMAGE\_DLLCHARACTERISTICS\_NO\_ISOLATION** | 0x0200 | The image is isolation aware, but should not be isolated |
| **IMAGE\_DLLCHARACTERISTICS\_NO\_SEH** | 0x0400 | The image does not use structured exception handling (SEH). No handlers can be called in this image |
| **IMAGE\_DLLCHARACTERISTICS\_NO\_BIND** | 0x0800 | Do not bind the image |
| No constant name | 0x1000 | Reserved |
| **IMAGE\_DLLCHARACTERISTICS\_WDM\_DRIVER** | 0x2000 | A WDM driver |
| No constant name | 0x4000 | Reserved |
| **IMAGE\_DLLCHARACTERISTICS\_TERMINAL\_SERVER\_AWARE** | 0x8000 | The image is terminal server aware |

SizeOfStackReserve

The number of bytes to reserve for the stack. Only the memory specified by the SizeOfStackCommit member is committed at load time; the rest is made available one page at a time until this reserve size is reached.

SizeOfStackCommit

The number of bytes to commit for the stack.

SizeOfHeapReserve

The number of bytes to reserve for the local heap. Only the memory specified by the SizeOfHeapCommit member is committed at load time; the rest is made available one page at a time until this reserve size is reached.

SizeOfHeapCommit

The number of bytes to commit for the local heap.

LoaderFlags

This member is obsolete.

NumberOfRvaAndSizes

The number of directory entries in the remainder of the optional header. Each entry describes a location and size.

DataDirectory

Possibly the most interesting member of this structure. Provides RVAs and sizes which locate various data structures, which are used for setting up the execution environment of a module. The details of what these structures do exist in other sections of this page. The most interesting entries in DataDirectory are as follows, Export Directory, Import Directory, Resource Directory, and the Bound Import directory. Note that the offsets in bytes are relative to the beginning of the optional header.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Constant Name** | **Value** | **Description** | **Offset PE(32 bit)** | **Offset PE32+(64 bit)** |
| **IMAGE\_DIRECTORY\_ENTRY\_EXPORT** | 0 | Export Directory | 96 | 112 |
| **IMAGE\_DIRECTORY\_ENTRY\_IMPORT** | 1 | Import Directory | 104 | 120 |
| **IMAGE\_DIRECTORY\_ENTRY\_RESOURCE** | 2 | Resource Directory | 112 | 128 |
| **IMAGE\_DIRECTORY\_ENTRY\_EXCEPTION** | 3 | Exception Directory | 120 | 136 |
| **IMAGE\_DIRECTORY\_ENTRY\_SECURITY** | 4 | Security Directory | 128 | 144 |
| **IMAGE\_DIRECTORY\_ENTRY\_BASERELOC** | 5 | Base Relocation Table | 136 | 152 |
| **IMAGE\_DIRECTORY\_ENTRY\_DEBUG** | 6 | Debug Directory | 144 | 160 |
| **IMAGE\_DIRECTORY\_ENTRY\_ARCHITECTURE** | 7 | Architecture specific data | 152 | 168 |
| **IMAGE\_DIRECTORY\_ENTRY\_GLOBALPTR** | 8 | Global pointer register relative virtual address | 160 | 176 |
| **IMAGE\_DIRECTORY\_ENTRY\_TLS** | 9 | Thread Local Storage directory | 168 | 184 |
| **IMAGE\_DIRECTORY\_ENTRY\_LOAD\_CONFIG** | 10 | Load Configuration directory | 176 | 192 |
| **IMAGE\_DIRECTORY\_ENTRY\_BOUND\_IMPORT** | 11 | Bound Import directory | 184 | 200 |
| **IMAGE\_DIRECTORY\_ENTRY\_IAT** | 12 | Import Address Table | 192 | 208 |
| **IMAGE\_DIRECTORY\_ENTRY\_DELAY\_IMPORT** | 13 | Delay Import table | 200 | 216 |
| **IMAGE\_DIRECTORY\_ENTRY\_COM\_DESCRIPTOR** | 14 | COM descriptor table | 208 | 224 |
| No constant name | 15 | Reserved | 216 | 232 |

**Code Sections**

The PE Header defines the number of sections in the executable file. Each section definition is 40 bytes in length. Below is an example hex from a program I am writing:

2E746578\_74000000\_00100000\_00100000\_A8050000 .text

00040000\_00000000\_00000000\_00000000\_20000000

2E646174\_61000000\_00100000\_00200000\_86050000 .data

000A0000\_00000000\_00000000\_00000000\_40000000

2E627373\_00000000\_00200000\_00300000\_00000000 .bss

00000000\_00000000\_00000000\_00000000\_80000000

The structure of the section descriptor is as follows:

Offset Length Purpose

------ ------- ------------------------------------------------------------------

0x00 8 bytes Section Name - in the above example the names are .text .data .bss

0x08 4 bytes Size of the section once it is loaded to memory

0x0C 4 bytes RVA (location) of section once it is loaded to memory

0x10 4 bytes Physical size of section on disk

0x14 4 bytes Physical location of section on disk (from start of disk image)

0x18 12 bytes Reserved (usually zero) (used in object formats)

0x24 4 bytes Section flags

A PE loader will place the sections of the executable image at the locations specified by these section descriptors (relative to the base address) and usually the alignment is 0x1000, which matches the size of pages on the x86.

Common sections are:

1. .text/.code/CODE/TEXT - Contains executable code (machine instructions)
2. .textbss/TEXTBSS - Present if Incremental Linking is enabled
3. .data/.idata/DATA/IDATA - Contains initialised data
4. .bss/BSS - Contains uninitialised data

**Section Flags**

The section flags is a 32-bit bit field (each bit in the value represents a certain thing). Here are the constants defined in the WINNT.H file for the meaning of the flags:

#define IMAGE\_SCN\_TYPE\_NO\_PAD 0x00000008 // Reserved.

#define IMAGE\_SCN\_CNT\_CODE 0x00000020 // Section contains code.

#define IMAGE\_SCN\_CNT\_INITIALIZED\_DATA 0x00000040 // Section contains initialized data.

#define IMAGE\_SCN\_CNT\_UNINITIALIZED\_DATA 0x00000080 // Section contains uninitialized data.

#define IMAGE\_SCN\_LNK\_OTHER 0x00000100 // Reserved.

#define IMAGE\_SCN\_LNK\_INFO 0x00000200 // Section contains comments or some other type of information.

#define IMAGE\_SCN\_LNK\_REMOVE 0x00000800 // Section contents will not become part of image.

#define IMAGE\_SCN\_LNK\_COMDAT 0x00001000 // Section contents comdat.

#define IMAGE\_SCN\_NO\_DEFER\_SPEC\_EXC 0x00004000 // Reset speculative exceptions handling bits in the TLB entries for this section.

#define IMAGE\_SCN\_GPREL 0x00008000 // Section content can be accessed relative to GP

#define IMAGE\_SCN\_MEM\_FARDATA 0x00008000

#define IMAGE\_SCN\_MEM\_PURGEABLE 0x00020000

#define IMAGE\_SCN\_MEM\_16BIT 0x00020000

#define IMAGE\_SCN\_MEM\_LOCKED 0x00040000

#define IMAGE\_SCN\_MEM\_PRELOAD 0x00080000

#define IMAGE\_SCN\_ALIGN\_1BYTES 0x00100000 //

#define IMAGE\_SCN\_ALIGN\_2BYTES 0x00200000 //

#define IMAGE\_SCN\_ALIGN\_4BYTES 0x00300000 //

#define IMAGE\_SCN\_ALIGN\_8BYTES 0x00400000 //

#define IMAGE\_SCN\_ALIGN\_16BYTES 0x00500000 // Default alignment if no others are specified.

#define IMAGE\_SCN\_ALIGN\_32BYTES 0x00600000 //

#define IMAGE\_SCN\_ALIGN\_64BYTES 0x00700000 //

#define IMAGE\_SCN\_ALIGN\_128BYTES 0x00800000 //

#define IMAGE\_SCN\_ALIGN\_256BYTES 0x00900000 //

#define IMAGE\_SCN\_ALIGN\_512BYTES 0x00A00000 //

#define IMAGE\_SCN\_ALIGN\_1024BYTES 0x00B00000 //

#define IMAGE\_SCN\_ALIGN\_2048BYTES 0x00C00000 //

#define IMAGE\_SCN\_ALIGN\_4096BYTES 0x00D00000 //

#define IMAGE\_SCN\_ALIGN\_8192BYTES 0x00E00000 //

#define IMAGE\_SCN\_ALIGN\_MASK 0x00F00000

#define IMAGE\_SCN\_LNK\_NRELOC\_OVFL 0x01000000 // Section contains extended relocations.

#define IMAGE\_SCN\_MEM\_DISCARDABLE 0x02000000 // Section can be discarded.

#define IMAGE\_SCN\_MEM\_NOT\_CACHED 0x04000000 // Section is not cachable.

#define IMAGE\_SCN\_MEM\_NOT\_PAGED 0x08000000 // Section is not pageable.

#define IMAGE\_SCN\_MEM\_SHARED 0x10000000 // Section is shareable.

#define IMAGE\_SCN\_MEM\_EXECUTE 0x20000000 // Section is executable.

#define IMAGE\_SCN\_MEM\_READ 0x40000000 // Section is readable.

#define IMAGE\_SCN\_MEM\_WRITE 0x80000000 // Section is writeable.

**Imports and Exports - Linking to other modules**

**What is linking?**

Whenever a developer writes a program, there are a number of subroutines and functions which are expected to be implemented already, saving the writer the hassle of having to write out more code or work with complex data structures. Instead, the coder need only declare one call to the subroutine, and the linker will decide what happens next.

There are two types of linking that can be used: static and dynamic. Static uses a library of precompiled functions. This precompiled code can be inserted into the final executable to implement a function, saving the programmer a lot of time. In contrast, dynamic linking allows subroutine code to reside in a different file (or *module*), which is loaded at runtime by the operating system. This is also known as a "Dynamically linked library", or DLL. A *library* is a module containing a series of functions or values that can be *exported*. This is different from the term *executable*, which *imports* things from libraries to do what it wants. From here on, "module" means any file of PE format, and a "Library" is any module which exports and imports functions and values.

Dynamically linking has the following benefits:

* It saves disk space, if more than one executable links to the library module
* Allows instant updating of routines, without providing new executables for all applications
* Can save space in memory by mapping the code of a library into more than one process
* Increases abstraction of implementation. The method by which an action is achieved can be modified without the need for reprogramming of applications. This is extremely useful for backward compatibility with operating systems.

This section discusses how this is achieved using the PE file format. An important point to note at this point is that *anything* can be imported or exported between modules, including variables as well as subroutines.

**Loading**

The downside of dynamically linking modules together is that, at runtime, the software which is initialising an executable must link these modules together. For various reasons, you cannot declare that "The function in this dynamic library will always exist in memory *here*". If that memory address is unavailable or the library is updated, the function will no longer exist there, and the application trying to use it will break. Instead, each module (library or executable) must declare what functions or values it *exports* to other modules, and also what it wishes to *import* from other modules.

As said above, a module cannot declare where in memory it expects a function or value to be. Instead, it declares where *in its own memory* it expects to find a **pointer** to the value it wishes to import. This permits the module to address any imported value, wherever it turns up in memory.

**Exports**

*Exports* are functions and values in one module that have been declared to be shared with other modules. This is done through the use of the "Export Directory", which is used to translate between the name of an export (or "Ordinal", see below), and a location in memory where the code or data can be found. The start of the export directory is identified by the IMAGE\_DIRECTORY\_ENTRY\_EXPORT entry of the resource directory. All export data must exist in the same section. The directory is headed by the following structure:

struct IMAGE\_EXPORT\_DIRECTORY {

long Characteristics;

long TimeDateStamp;

short MajorVersion;

short MinorVersion;

long Name;

long Base;

long NumberOfFunctions;

long NumberOfNames;

long \*AddressOfFunctions;

long \*AddressOfNames;

long \*AddressOfNameOrdinals;

}

The "Characteristics" value is generally unused, TimeDateStamp describes the time the export directory was generated, MajorVersion and MinorVersion should describe the version details of the directory, but their nature is undefined. These values have little or no impact on the actual exports themselves. The "Name" value is an RVA to a zero terminated ASCII string, the name of this library name, or module.

**Names and Ordinals**

Each exported value has both a name and an "ordinal" (a kind of index). The actual exports themselves are described through AddressOfFunctions, which is an RVA to an array of RVAs, each pointing to a different function or value to be exported. The size of this array is in the value NumberOfFunctions. Each of these functions has an ordinal. The "Base" value is used as the ordinal of the first export, and the next RVA in the array is Base+1, and so forth.

Each entry in the AddressOfFunctions array is identified by a name, found through the RVA AddressOfNames. The data where AddressOfNames points to is an array of RVAs, of the size NumberOfNames. Each RVA points to a zero terminated ASCII string, each being the name of an export. There is also a second array, pointed to by the RVA in AddressOfNameOrdinals. This is also of size NumberOfNames, but each value is a 16 bit word, each value being an ordinal. These two arrays are parallel and are used to get an export value from AddressOfFunctions. To find an export by name, search the AddressOfNames array for the correct string and then take the corresponding value from the AddressOfNameOrdinals array. This value is then used as index to AddressOfFunctions (yes, it's 0-based index actually, NOT base-biased ordinal, as the official documentation suggests!).

**Forwarding**

As well as being able to export functions and values in a module, the export directory can *forward* an export to another library. This allows more flexibility when re-organising libraries: perhaps some functionality has branched into another module. If so, an export can be forwarded to that library, instead of messy reorganising inside the original module.

Forwarding is achieved by making an RVA in the AddressOfFunctions array point into the section which contains the export directory, something that normal exports should not do. At that location, there should be a zero terminated ASCII string of format "LibraryName.ExportName" for the appropriate place to forward this export to.

**Imports**

The other half of dynamic linking is importing functions and values into an executable or other module. Before runtime, compilers and linkers do not know where in memory a value that needs to be imported could exist. The import table solves this by creating an array of pointers at runtime, each one pointing to the memory location of an imported value. This array of pointers exists inside of the module at a defined RVA location. In this way, the linker can use addresses inside of the module to access values outside of it.

**The Import directory**

The start of the import directory is pointed to by both the IMAGE\_DIRECTORY\_ENTRY\_IAT and IMAGE\_DIRECTORY\_ENTRY\_IMPORT entries of the resource directory (the reason for this is uncertain). At that location, there is an array of IMAGE\_IMPORT\_DESCRIPTOR structures. Each of these identify a library or module that has a value we need to import. The array continues until an entry where all the values are zero. The structure is as follows:

struct IMAGE\_IMPORT\_DESCRIPTOR {

long \*OriginalFirstThunk;

long TimeDateStamp;

long ForwarderChain;

long Name;

long \*FirstThunk;

}

The TimeDateStamp is relevant to the act of "Binding", see below. The Name value is an RVA to an ASCII string, naming the library to import. ForwarderChain will be explained later. The only thing of interest at this point, are the RVAs OriginalFirstThunk and FirstThunk. Both these values point to arrays of RVAs, each of which point to a IMAGE\_IMPORT\_BY\_NAMES struct. The arrays are terminated with an entry that is equal to zero. These two arrays are parallel and point to the same structure, in the same order. The reason for this will become apparent shortly.

Each of these IMAGE\_IMPORT\_BY\_NAMES structs has the following form:

struct IMAGE\_IMPORT\_BY\_NAME {

short Hint;

char Name[1];

}

"Name" is an ASCII string of any size that names the value to be imported. This is used when looking up a value in the export directory (see above) through the AddressOfNames array. The "Hint" value is an index into the AddressOfNames array; to save searching for a string, the loader first checks the AddressOfNames entry corresponding to "Hint".

To summarise: The import table consists of a large array of IMAGE\_IMPORT\_DESCRIPTORs, terminated by an all-zero entry. These descriptors identify a library to import things from. There are then two parallel RVA arrays, each pointing at IMAGE\_IMPORT\_BY\_NAME structures, which identify a specific value to be imported.

**Imports at runtime**

Using the above import directory at runtime, the loader finds the appropriate modules, loads them into memory, and seeks the correct export. However, to be able to use the export, a pointer to it must be stored somewhere in the importing module's memory. This is why there are two parallel arrays, OriginalFirstThunk and FirstThunk, identifying IMAGE\_IMPORT\_BY\_NAME structures. Once an imported value has been resolved, the pointer to it is stored in the FirstThunk array. It can then be used at runtime to address imported values.

**Bound imports**

The PE file format also supports a peculiar feature known as "binding". The process of loading and resolving import addresses can be time consuming, and in some situations this is to be avoided. If a developer is fairly certain that a library is not going to be updated or changed, then the addresses in memory of imported values will not change each time the application is loaded. So, the import address can be precomputed and stored in the FirstThunk array *before* runtime, allowing the loader to skip resolving the imports - the imports are "bound" to a particular memory location. However, if the versions numbers between modules do not match, or the imported library needs to be relocated, the loader will assume the bound addresses are invalid, and resolve the imports anyway.

The "TimeDateStamp" member of the import directory entry for a module controls binding; if it is set to zero, then the import directory is not bound. If it is non-zero, then it is bound to another module. However, the TimeDateStamp in the import table must match the TimeDateStamp in the bound module's FileHeader, otherwise the bound values will be discarded by the loader.

**Forwarding and binding**

Binding can of course be a problem if the bound library / module forwards its exports to another module. In these cases, the non-forwarded imports can be bound, but the values which get forwarded must be identified so the loader can resolve them. This is done through the ForwarderChain member of the import descriptor. The value of "ForwarderChain" is an index into the FirstThunk and OriginalFirstThunk arrays. The OriginalFirstThunk for that index identifies the IMAGE\_IMPORT\_BY\_NAME structure for a import that needs to be resolved, and the FirstThunk for that index is the index of another entry that needs to be resolved. This continues until the FirstThunk value is -1, indicating no more forwarded values to import.

**Resources**

**Resource structures**

Resources are data items in modules which are difficult to be stored or described using the chosen programming language. This requires a separate compiler or resource builder, allowing insertion of dialog boxes, icons, menus, images, and other types of resources, including arbitrary binary data. A number of API calls can then be used to retrieve resources from the module. The base of resource data is pointed to by the IMAGE\_DIRECTORY\_ENTRY\_RESOURCE entry of the data directory, and at that location there is an IMAGE\_RESOURCE\_DIRECTORY structure:

struct IMAGE\_RESOURCE\_DIRECTORY

{

long Characteristics;

long TimeDateStamp;

short MajorVersion;

short MinorVersion;

short NumberOfNamedEntries;

short NumberOfIdEntries;

}

Characteristics is unused, and TimeDateStamp is normally the time of creation, although it doesn't matter if it's set or not. MajorVersion and MinorVersion relate to the versioning info of the resources: the fields have no defined values. Immediately following the IMAGE\_RESOURCE\_DIRECTORY structure is a series of IMAGE\_RESOURCE\_DIRECTORY\_ENTRYs, the number of which are defined by the total of NumberOfNamedEntries and NumberOfIdEntries. The first portion of these entries are for named resources, the latter for ID resources, depending on the values in the IMAGE\_RESOURCE\_DIRECTORY struct. The actual shape of the resource entry structure is as follows:

struct IMAGE\_RESOURCE\_DIRECTORY\_ENTRY

{

long NameId;

long \*Data;

}

The NameId value has dual purpose: if the most significant bit (or sign bit) is clear, then the lower 16 bits are an ID number of the resource. Alternatly, if the top bit is set, then the lower 31 bits make up an offset from the start of the resource data to the name string of this particular resource. The Data value also has a dual purpose: if the most significant bit is set, the remaining 31 bits form an offset from the start of the resource data to another IMAGE\_RESOURCE\_DIRECTORY (i.e. this entry is an interior node of the resource tree). Otherwise, this is a leaf node, and Data contains the offset from the start of the resource data to a structure which describes the specifics of the resource data itself (which can be considered to be an ordered stream of bytes):

struct IMAGE\_RESOURCE\_DATA\_ENTRY

{

long \*Data;

long Size;

long CodePage;

long Reserved;

}

The Data value contains an RVA to the actual resource data, Size is self-explanatory, and CodePage contains the Unicode codepage to be used for decoding Unicode-encoded strings in the resource (if any). Reserved should be set to 0.

**Layout**

The above system of resource directory and entries allows simple storage of resources, by name or ID number. However, this can get very complicated very quickly. Different types of resources, the resources themselves, and instances of resources in other languages can become muddled in just one directory of resources. For this reason, the resource directory has been given a structure to work by, allowing separation of the different resources.

For this purpose, the "Data" value of resource entries points at another IMAGE\_RESOURCE\_DIRECTORY structure, forming a tree-diagram like organisation of resources. The first level of resource entries identifies the *type* of the resource: cursors, bitmaps, icons and similar. They use the ID method of identifying the resource entries, of which there are twelve defined values in total. More user defined resource types can be added. Each of these resource entries points at a resource directory, naming the actual resources themselves. These can be of any name or value. These point at yet another resource directory, which uses ID numbers to distinguish languages, allowing different specific resources for systems using a different language. Finally, the entries in the language directory actually provide the offset to the resource data itself, the format of which is not defined by the PE specification and can be treated as an arbitrary stream of bytes.

**Relocations**

**Alternate Bound Import Structure**

**Windows DLL Files**

Windows DLL files are a brand of PE file with a few key differences:

* A .DLL file extension
* A DllMain() entry point, instead of a WinMain() or main().
* The DLL flag set in the PE header.

DLLs may be loaded in one of two ways, a) at load-time, or b) by calling the LoadModule() Win32 API function.

**Function Exports**

Functions are exported from a DLL file by using the following syntax:

\_\_declspec(dllexport) void MyFunction() ...

The "\_\_declspec" keyword here is not a C language standard, but is implemented by many compilers to set extendable, compiler-specific options for functions and variables. Microsoft C Compiler and GCC versions that run on windows allow for the \_\_declspec keyword, and the dllexport property.

Functions may also be exported from regular .exe files, and .exe files with exported functions may be called dynamically in a similar manner to .dll files. This is a rare occurrence, however.

**Identifying DLL Exports**

There are several ways to determine which functions are exported by a DLL. The method that this book will use (often implicitly) is to use **dumpbin** in the following manner:

dumpbin /EXPORTS <dll file>

This will post a list of the function exports, along with their ordinal and RVA to the console.

**Function Imports**

In a similar manner to function exports, a program may import a function from an external DLL file. The dll file will load into the process memory when the program is started, and the function will be used like a local function. DLL imports need to be prototyped in the following manner, for the compiler and linker to recognize that the function is coming from an external library:

\_\_declspec(dllimport) void MyFunction();

**Identifying DLL Imports**

If is often useful to determine which functions are imported from external libraries when examining a program. To list import files to the console, use **dumpbin** in the following manner:

dumpbin /IMPORTS <dll file>

You can also use depends.exe to list imported and exported functions. Depends is a a GUI tool and comes with Microsoft Platform SDK.