PE File Structure

The **"portable executable file format" (PE)** is the format of the binary programs (exe, dll, sys, scr) for MS windows NT, windows 95 and win32s. It can also be used for object files (bpl, dpl, cpl, ocx, acm, ax).

The format was designed by **Microsoft** and then in 1993 standardized by the **Tool Interface Standard Committee** (Microsoft, Intel, Borland, Watcom, IBM and others) , apparently based on the **"common object file format"** (COFF), the format used for object files and executables on several UNIX and VMS OSes.

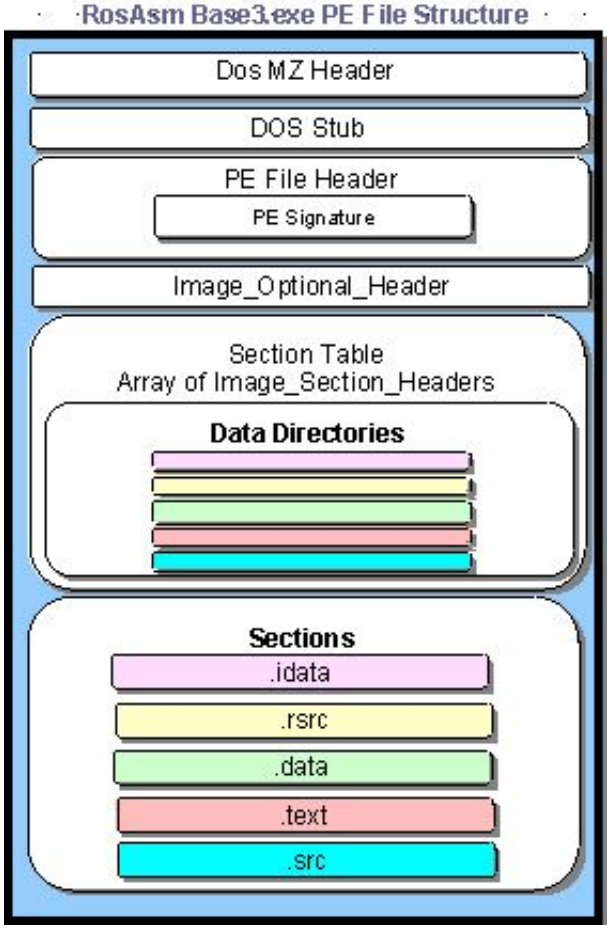
The following 3 paragraphs copied from Mark Pietrek's article on MSDN

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

A very handy aspect of PE files is that the Data **Structures** on disk are the same data structures used in memory. Loading an executable into memory is primarily a matter of mapping certain ranges of a PE file into the address space. Thus, a data structure is identical on disk and in memory. The key point is that if you know how to find something in a PE file, you can almost certainly find the same information after the file is loaded in memory. It's important to note that PE files are not just mapped into memory as a single memory-mapped file. Instead, the Win32 loader looks at the PE file and decides what portions of the file to map in. This mapping is consistent in that higher offsets in the file correspond to higher memory addresses when mapped into memory. The offset of an item in the disk file may differ from its offset once loaded into memory. However, all the information is present to allow you to make the translation from disk offset to memory offset.

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

The **PE** data structures are: **DOS header, DOS stub, PE File Header, Image Optional Header, Section Table -** which has a trailing array of **Section Headers - Data Directories -** these directories contain pointers to data in the individual sections and, lastly, the individual **Sections** themselves.

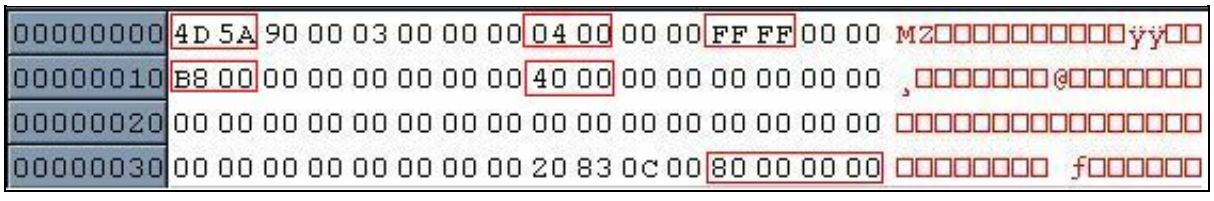


Each section header has some flags about what kind of data it contains ("initialized data"," readable data","writable data" and so on), whether it can be shared etc., and pointers (RVA's). A PE header component is called the "IMAGE_DIRECTORY HEADER". This header holds information about some PE Sections (Resources, Import, and so on). Each Record being [PointerToSection] [Size].

Some sections also have additional headers (for example .rsrc) called **Section Data Headers**. Some don't, directoryless types of contents of sections are, for example, "executable code" or "initialized data". Essentially, the sections' contents is what you really need to execute a program, and all the header and directory stuff is just there to help you or the win32 loader to find it.

PE File Layout RosAsm Base3.exe

Dos MZ Header



This makes a **PE** file an **MS-DOS** executable. First 2 bytes are always: **4D**h **5A**h > "**MZ**" is **dos exe signature** starting at 0h, **Last Page Size** (2 bytes),**Total Pages in File** (2 bytes), **Relocation Items** (2 bytes). The word at offset 8h tells the # of 16 byte paragraphs the DOS header contains > **Dos Header Size** 4h (16*4) = 40h bytes

in length,

Min Size (2 bytes), **Max Size** FFFFh @ offset 0Ch, **Initial Stack Segment** (SP register value at run time 2 bytes), **Initial Stack Pointer** 2 bytes, 2 byte **Checksum for Header** @ offset 12h, **Initial Instruction Pointer** 2 bytes, **Initial Code Segment** 2 bytes, **Relocation Table Offset** 40h @ offset 18h 2 bytes,**Overlay #** default 2 bytes, **Betov's CheckSum** 4 bytes @ offset 38h (located in the last part of 8 reserved words) and the **PE Header Pointer** > 80h 4 bytes @ offset 3Ch.

The above as seen in Quickview. (00h thru 3Fh)

Header Information

Signature:	5a4d
Last Page Size:	0090
Total Pages in File:	0003
Relocation Items:	0000
Paragraphs in Header:	0004
Minimum Extra Paragraphs:	0000
Maximum Extra Paragraphs:	ffff
Initial Stack Segment:	0000
Initial Stack Pointer:	00b8
Complemented Checksum:	0000
Initial Instruction Pointer:	0000
Initial Code Segment:	0000
Relocation Table Offset:	0040
Overlay Number:	0000
Reserved:	0000 0000 0000 0000
	0000 0000 0000 0000
	0000 0000 0000 0000
	0000 0000 8320 000c
Offset to New Header:	00000080
Memory Needed:	2K

In a **Win32 system the PE loader just skips** the following MS-DOS Stub.

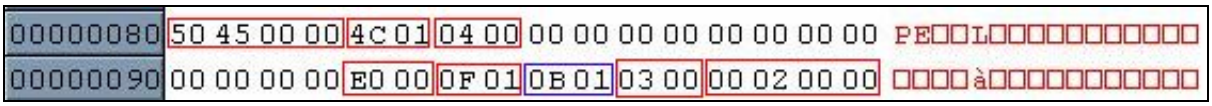
MS-DOS executable ("stub")

```
00000040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 53 70 □□°□□´□Í!¸□LÍ!Sp
00000050 69 6E 64 6F 7A 20 33 32 20 73 70 69 74 20 50 45 indoz 32 spit PE
00000060 66 69 6C 65 20 6D 61 64 65 20 77 69 7A 20 52 6F file made wiz Ro
00000070 73 41 73 6D 20 41 73 73 65 6D 62 6C 65 72 2E 24 sAsm Assembler.$
```

The **DOS stub** is actually a valid EXE for PE-files, it is a MS-DOS 2.0 compatible executable that almost always consists of a small number of bytes that output an error message. It can simply display a string like "This program requires Windows" or "Cannot be run in DOS mode" or the **"RosAsm message above"**.

The bytes from 40h to 4Dh above is the actual code: push cs // pop ds // mov dx 0E // mov ah 09 // int 021 // mov ax 4C01 // int 021 to print the message if the program is run in DOS.

PE File header (in the COFF-format)



First row: The 32-bit-**PE signature** (first 4 bytes contains the number 4550h > "**PE**") @ offset 80h , the 2 byte **IMAGE_FILE_MACHINE** @ offset 84h for x**86**: (**14C**h) for Intel 80386 processor or better, (14Dh) for Intel 80486 processor or better,(14Eh) for Intel Pentium processor or better. 2 bytes @offset 86h contains **how many sections** are in it **04h** . Next is a timestamp **'TimeDateStamp'** (32 bit value).The members **'PointerToSymbolTable'** (32 bit value) and,

Second row:'NumberOfSymbols' 0h @ offset 90h (32 bit value) are used for debugging information. **'SizeOfOptionalHeader'** E0h (16 bit) @ offset 94h:This header tells us how the binary should be loaded: The starting address, the amount of stack to reserve, the size of the data segment etc..**'Characteristics'** is 16 bits 10Fh (1_00001111) @ offset 96h and consists of a collection of flags, most of which are valid for libraries and object files. **Optional Header Starts @** offset 98h the 16-bit-word is **'Magic'** always contains the value 10Bh.The next 2 bytes 3h @ offset 9Ah are the **'Version'** of the linker.**'SizeOfCode'** is last 4 bytes **Size of the executable code.**

The above as seen in Quickview. (80h thru 9Fh)

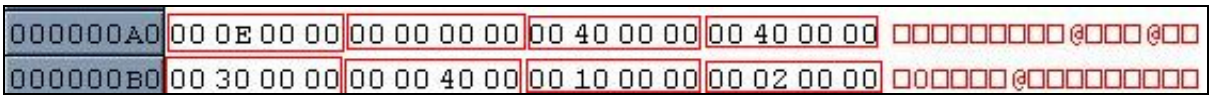
Image File Header

Signature:	00004550
Machine:	Intel 386
Number of Sections:	0004
Time Date Stamp:	00000000
Symbols Pointer:	00000000
Number of Symbols:	00000000
Size of Optional Header	00e0

Image Optional Header

Magic:	010b
Linker Version:	3.00
Size of Code:	00000200

Thusfar in Base3.exe as written by **RosAsm,** we have encountered the **Dos** header, **Dos** stub and the **PE** 'image file header' - that tells us most importantly, what machine it runs on, **how many sections** are in it and the **size of** the '**Image Optional Header**', which immediately follows and we are now in....starting at 98h,with magic and is E0h bytes long, adding the two together we get **178h**... which is the start of the **IMAGE_DATA_DIRECTORY** array as we will see below after we finish with the '**Image Optional Header**' ..

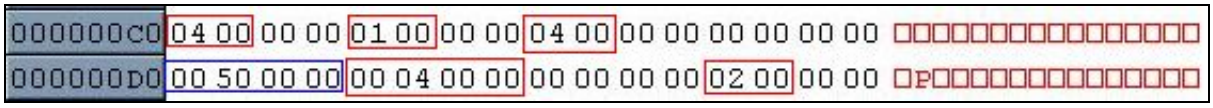


First row: The next 2 longwords @ offset A0h (32 bits each) are intended to be the size of the initialized data (**'SizeOfInitializedData'),** the "data segment" and the size of the uninitialized data ('**SizeOfUninitializedData'**) the so-called "bss segment". This is the size of everything but Code and Virtual data (size of initialized data .data + .rsrc+... + .reloc). Next is a 32-bit-value @ offset A8h that is an **RVA (relative virtual address**). This RVA is the offset to the code's entry point, program execution starts here (**'AppRVAentryPoint'**).The next 2 32-bit-values are the offsets to the executable code (**'AppBaseOfCode')** and,

Second row: The initialized data @ offset B0h (**'SHAppBaseOfData'**).The next entry is a 32-bit-value giving the preferred (linear) load address (**'ImageBase'**) of the entire binary, including all headers 400000h = (The image base linker default value). The next 2 32-bit-values @ offset B8h are the ***alignments*** of the PE-file's sections in **RAM 'Section Alignment'** when the image has been loaded and in the **'File Alignment'** in the file.

The above as seen in Quickview. (A0h thru BFh)

Size of Initialized Data:	00000e00
Size of Uninitialized Data:	00000000
Address of Entry Point:	00004000
Base of Code:	00004000
Base of Data:	00003000
Image Base:	00400000
Section Alignment:	00001000
File Alignment:	00000200

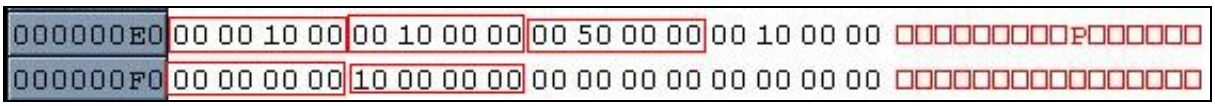


First row: The next 4 16-bit-words are the expected operating system version ('MajorOperatingSystemVersion' and 'MinorOperatingSystemVersion, the next 2 16-bit-words are the expected subsystem version and 4 bytes of reserved space.

Second row: 'SizeOfImage' 32 bit value@ offset D0h.It is the sum of all headers' and sections' lengths if aligned to 'SectionAlignment'. It is a hint to the loader how many pages it will need in order to load the image in RAM. Next **'SizeOfHeaders'** @ offset D4h a 32-bit-value giving the total length of all headers including the data directories and the section headers, it is also the offset to the sections. Then we have got a 32-bit-checksum **'CheckSum'.** The algorithm to compute the checksum is property of Microsoft, and they won't tell you, it is only needed for Driver PEs. The checksum need not be supplied and may be 0. Then there is a 16-bit-word **'Subsystem'** @ offset DCh IMAGE_SUBSYSTEM_WINDOWS_GUI (2h). Windows 95 binaries will always use the Win32 subsystem, so the only legal values for these binaries are 2 and 3. The last thing in the second row is a 16-bit-value that tells, if the image is a DLL.

The above as seen in Quickview. (C0h thru DFh)

Operating System Version:	4.00
Image Version:	1.00
Subsystem Version:	4.00
Reserved1:	00000000
Size of Image:	00005000
Size of Headers:	00000400
Checksum:	00000000
Subsystem:	Image runs in the Windows GUI subsystem.
DLL Characteristics:	0000



First row: The next 4 32-bit-values starting @ offset E0h are the size of stack reserve (**'SizeOfStackReserve'**), the size of initially committed stack (**'SizeOfStackCommit'**) , the size of the reserved heap(**'SizeOfHeapReserve'**) and the size of the committed heap (**'SizeOfHeapCommit'**).

Second row: After these stack- and heap-descriptions, we find 32 bits of **'LoaderFlags'**. Then the **Size of Data Directory**, number of possible entries in the ***following section table*** (**16 records**) 4 bytes @ offset F4h.called **'NumberOfRvaAndSizes'**. The last 8 bytes would be for the Export Directory pointers entry if used..

The above as seen in Quickview. (E0h thru FFh)

Size of Stack Reserve:	00100000
Size of Stack Commit:	00001000
Size of Heap Reserve:	00005000
Size of Heap Commit:	00001000
Loader Flags:	00000000
Size of Data Directory:	00000010

```
00000100 00 10 00 00 3C 00 00 00 00 20 00 00 00 08 00 00 □□□□<□□□□ □□□□□□
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 □□□□□□□□□□□□□□□□
```

First row: The next 4 32-bit-values starting @ offset 100h are Import Directory Virtual Address **('AppBaseOfImport'),** the Import Directory Size **('AppImportSize')**, Resource Directory Virtual Address **('AppBaseOfRsrc')** and the Resource Directory Size **('AppRsrcSize')**.

Second row: The 32-bit-values starting @ offset **110h** and those following to **157h** are filled with zeros and would be entries, RVAs and Sizes for: Exception, Security, Relocation Table, Debug, Copyright, Global Mips gp and Global Pointer, Thread Local Storage and Load Configuration, if used.

The above as seen in Quickview. (100h thru 10Fh)

Import Directory Virtual Address:	1000
Import Directory Size:	003c
Resource Directory Virtual Address:	2000
Resource Directory Size:	0800

```
00000150 00 00 00 00 00 00 00 00 88 10 00 00 4C 00 00 00 □□□□□□□□^□□□L□□□
00000160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 □□□□□□□□□□□□□□□□
00000170 00 00 00 00 00 00 00 00 2E 69 64 61 74 61 00 00 □□□□□□□□.idata□□
```

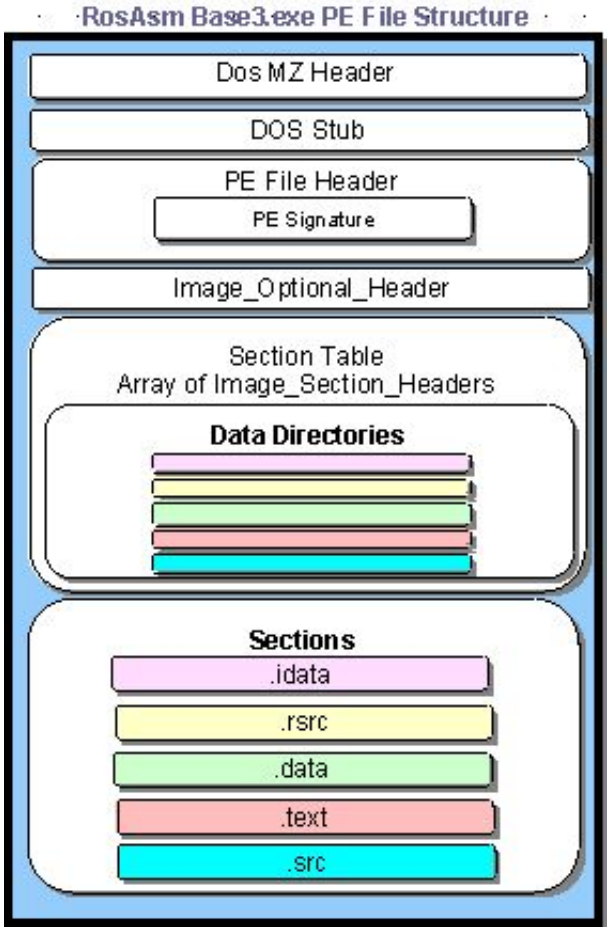
First row: The 32-bit-values starting @ offset 158h are the Second Import Address Table **'AppSecondImport'** and its size **'AppSecondImportSize'** This is explained below.

Second row: The the space from 160h filled with zeros is reserved and the **Image Optional Header** ends at **177h.**

The entire 'Image Optional Header' as seen in Quickview

Image Optional Header

Magic:	010b
Linker Version:	3.00
Size of Code:	00000200
Size of Initialized Data:	00000e00
Size of Uninitialized Data:	00000000
Address of Entry Point:	00004000
Base of Code:	00004000
Base of Data:	00003000
Image Base:	00400000
Section Alignment:	00001000
File Alignment:	00000200
Operating System Version:	4.00
Image Version:	1.00
Subsystem Version:	4.00
Reserved1:	00000000
Size of Image:	00005000
Size of Headers:	00000400
Checksum:	00000000
Subsystem:	Image runs in the Windows GUI subsystem.
DLL Characteristics:	0000
Size of Stack Reserve:	00100000
Size of Stack Commit:	00001000
Size of Heap Reserve:	00005000
Size of Heap Commit:	00001000
Loader Flags:	00000000
Size of Data Directory:	00000010
Import Directory Virtual Address:	1000
Import Directory Size:	003c
Resource Directory Virtual Address:	2000
Resource Directory Size:	0800



Section Headers

Between the PE headers and the raw data for the image's Sections lies the **Section Table**.

There is one section header for each section, and each data directory will point to one of the sections. Several data directories may point to the same section, and there may be sections without a data directory pointing to them.

The sections in the image are sorted by their starting address (RVAs), rather than alphabetically.

Section Table - Image Data Directories

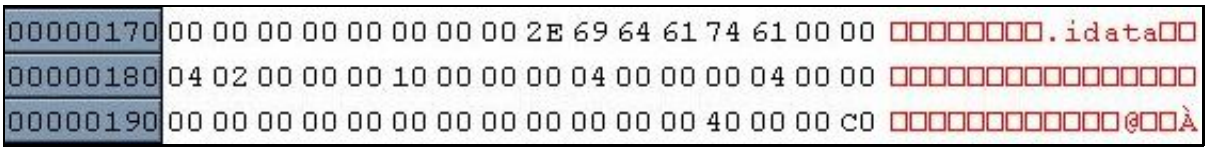
Sections have two alignment values, one within the disk file (**Pointer to Raw Data**) and the other in memory (**Virtual Address**). The **PE file header** specifies both of these values, which can differ. Each **section** starts at an offset that's some multiple of the alignment value. For instance, in the PE file, a typical alignment would be 200h. Thus, every section begins at a **file offset** that's a multiple of 200h. Once mapped into memory, sections always start on at least a **page boundary**. That is, when a PE section is mapped into memory, the first byte of each section corresponds to a memory page. On x86 CPUs pages are 4KB aligned, while on the IA-64 CPUs they are 8KB aligned.

The Section Table is an **array** of IMAGE_NUMBER_OF_DIRECTORY ENTRIES (16 spaces reserved for entries) IMAGE_DATA_DIRECTORYs. Each of these directories describes the location (32 bits RVA called '**Virtual Address**') and size (also 32 bit, called '**Size of Raw Data**') of a particular piece of information, which is located in one of the sections that follow the directory entries. Some elements at the end of the array are currently unused.

This array allows the **loader to quickly** find a particular section of the image (for example, the imported function table), **without having to iterate** through each of the images sections, **comparing names** as it goes along. In the section table the first entry of an array element is of the SHORT_NAME 8 bytes, that make up the name (in ASCII) of the section. If all of the 8 bytes are used there is no 0 terminator for the string... Followed by two sets of Dwords that are the sizes and the second ones that are the RVA addresses for each entry and data characteristics sometimes called flags.

Partial Section Records of Directory entries of Base3.exe each .jpg showing the relevant entry.

Import Table - Data Section Header starts at 178h



'**.idata**': @ offset 178h padded with 2 zeros = 8 bytes, **AppImportTrueSize**: (Virtual Size) 204h @ offset 180h 4 bytes, **AppBaseOfImports**: **RVA** 1000h @ offset 184h 4 bytes, **AppImportAlignedSize**: 400h @ offset 188h 4 bytes, **AppStartOfImport**: 400h @ offset 18Ch 4 bytes,12 bytes not used. The **flags** describing how the section's memory should be treated **DataCharacteristics** D$ **0_C0000040** @ offset 19Ch; readable, writeable, initialized data.

The above as seen in Quickview. (170h thru 19Fh)

Section name:	.idata
Virtual Size:	00000204
Virtual Address:	00001000
Size of raw data:	00000400
Pointer to Raw Data:	00000400
Pointer to Relocations:	00000000
Pointer to Line Numbers:	00000000
Number of Relocations:	0000
Number of Line Numbers:	0000
Characteristics:	Section contains initialized data
	Section is readable
	Section is writeable

As we can see the **.idata** section starts at 400h in the file and is 400h bytes in length...

When you use code or data from a DLL, you're importing it. When any PE file loads, one of the jobs of the ***Win32 loader is to locate all the imported functions and data and make those addresses available*** to the file being loaded. When you link directly against the code and data of another DLL, you're implicitly linking against the DLL. You don't have to do anything to make the addresses of the imported APIs available to your code. Within a PE file, (**Not shown in .jpgs**) there's an **array of data structures**, one per imported DLL. Each of these structures gives the name of the imported DLL and points to an array of function pointers. The array of function pointers is known as the **Import Address Table** (**IAT**). Each imported API has its own reserved spot in the **IAT** where the address of the imported function is written by the **Win32 loader**. This last point is particularly important: once a module is loaded into RAM, the **IAT** contains the address that is invoked when calling imported APIs. The beauty of the IAT is that there's ***just one place in a PE file when loaded into RAM where an imported API's address is stored***. all the calls go through the same function pointer in the IAT.

Actual table as seen in Quickview.

Import Table

KERNEL32.dll

Ordinal	Function Name
0000	GetModuleHandleA
0000	ExitProcess

USER32.dll

Ordinal	Function Name
0000	LoadIconA
0000	LoadCursorA
0000	RegisterClassA
0000	LoadMenuA
0000	CreateWindowExA
0000	ShowWindow
0000	UpdateWindow
0000	TranslateMessage
0000	DispatchMessageA
0000	GetMessageA
0000	DestroyWindow
0000	PostQuitMessage
0000	SendMessageA
0000	MessageBoxA
0000	DefWindowProcA

The important parts of an **import table** are the imported DLL name and the **two arrays** of IMAGE_IMPORT_BY_NAME pointers. In the EXE file, the **two arrays** (pointed to by the Characteristics and FirstThunk fields) run parallel to each other, and are terminated by a NULL pointer entry at the end of each array. The pointers in both arrays point to an IMAGE_IMPORT_BY_NAME structure. Why are there two parallel arrays of pointers to the IMAGE_IMPORT_BY_NAME structures? The array pointed at by the Characteristics field is *left alone, and never modified*. It's sometimes called the hint-name table. The array pointed at by the FirstThunk field **is overwritten by the PE loader**. The loader iterates through each pointer in the array and finds the address of the function that each IMAGE_IMPORT_BY_NAME structure refers to. The **loader then overwrites** in RAM the pointer with the found function's address. Since the array of pointers that are **overwritten** by the loader eventually holds the addresses of all the imported functions, it's called the **Import Address Table**.

Resource - Data Section Header

```
000001A0 2E 72 73 72 63 00 00 00 80 06 00 00 00 20 00 00  .rsrc□□□□□□□□ □□
000001B0 00 08 00 00 00 08 00 00 00 00 00 00 00 00 00 00  □□□□□□□□□□□□□□□□
000001C0 00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00  □□□□@□□@.data□□□
```

'.rsrc': padded with 3 zeros = 8 bytes, **AppRsrcTrueSize**: virtual size 680h 4 bytes, **AppBaseOfRsrcs**: virtual address 2000h 4 bytes, **RVA AppRsrcAlignedSize:** size of raw data 800h 4 bytes, **AppStartOfRsrc**: pointer to raw data 800h 4 bytes, 12 bytes not used.The **flags** describing how the section's memory should be treated, **DataCharacteristics**:D$ 0_40000040 ; readable, initialized data.

The resources, such as dialog boxes, menus, icons and so on, are in the data directory pointed to by IMAGE_DIRECTORY_ENTRY_RESOURCE.

It is in a section that has, at least, the bits 'IMAGE_SCN_CNT_INITIALIZED_DATA' and 'IMAGE_SCN_MEM_READ' set.

The above as seen in Quickview. (1A0h thru 1C7h)

Section name:	.rsrc
Virtual Size:	00000680
Virtual Address:	00002000
Size of raw data:	00000800
Pointer to Raw Data:	00000800
Pointer to Relocations:	00000000
Pointer to Line Numbers:	00000000
Number of Relocations:	0000
Number of Line Numbers:	0000
Characteristics:	Section contains initialized data Section is readable

As we can see the resource section starts at 800h in the file and is 800h bytes in length...

The following was copied from some of Mark Pietrek's articles:

Navigating the resource directory hierarchy is like navigating a hard disk. There's a **'Section Data Header'** - master directory (the root directory), which has subdirectories. The subdirectories have subdirectories of their own that may point to the raw resource data for things like dialog templates. In the PE format, both the root directory of the resource directory hierarchy and all of its subdirectories are structures of type IMAGE_RESOURCE_DIRECTORY.

A directory entry can either point at a subdirectory (that is, to another IMAGE_RESOURCE_DIRECTORY), or it can point to the raw data for a resource. Generally, there are at least three directory levels before you get to the actual raw resource data. The top-level directory (of which there's only one) is always found at the beginning of the resource section (.rsrc). The subdirectories of the top-level directory correspond to the various types of resources found in the file. For example, if a PE file includes dialogs, string tables, and menus, there will be three subdirectories: a dialog directory, a string table directory, and a menu directory. Each of these type subdirectories will in turn have ID subdirectories. There will be one ID subdirectory for each instance of a given resource type.Each ID subdirectory will have either a string name (such as "MyDialog") or the integer ID used to identify the resource in the RC file.

IMAGE_RESOURCE_DIRECTORY Format

DWORD Characteristics - Theoretically this field could hold flags for the resource, but appears to always be 0.

DWORD TimeDateStamp - The time/date stamp describing the creation time of the resource.

WORD *MajorVersion - These *fields would hold a version number for the resource.

WORD *MinorVersion - These *fields appear to always be set to 0.

WORD NumberOfNamed Entries - The number of array elements that use **names** and that follow this structure.

WORD NumberOfIdEntries - The number of array elements that use **integer IDs**, and which follow this structure.

IMAGE_RESOURCE_DIRECTORY_ENTRY - Directory Entries.

This field isn't really part of the IMAGE_RESOURCE_DIRECTORY structure.

Rather, **its an array** of IMAGE_RESOURCE_DIRECTORY_ENTRY structures that immediately follow the IMAGE_RESOURCE_DIRECTORY structure. The **number of elements in the array is the sum** of the NumberOfNamedEntries and NumberOfIdEntries fields. The directory entry elements that have **name identifiers** (rather than integer IDs) come **first** in the array. A directory entry can either **point at a subdirectory** (that is, to another IMAGE_RESOURCE_DIRECTORY), or it can **point to the raw data** for a resource.

IMAGE_RESOURCE_DIRECTORY_ENTRY - Format

DWORD Name

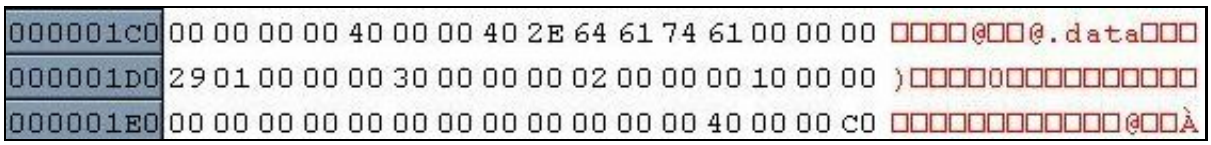
This field contains either **an integer ID or a pointer** to a structure that contains a string name. If the **high bit** (0x80000000) **is zero**, this field is interpreted **as an integer ID**. If the **high bit is nonzero**, the lower 31 bits are an **offset** (relative **to the start** of the resources) to an IMAGE_RESOURCE_DIR_STRING_U structure. This structure contains a WORD character count, followed by a UNICODE string with the resource name. Yes, even PE files intended for non-UNICODE Win32 implementations use UNICODE here. To convert the UNICODE string to an ANSI string, use the WideCharToMultiByte function.

DWORD OffsetToData

This field is either an **offset** to another resource directory or a **pointer** to information about a specific resource instance. If the **high bit** (0x80000000) **is set**, this directory entry refers to a **subdirectory** and the **lower 31 bits are an offset** (relative to the start of the resources) to another IMAGE_RESOURCE_DIRECTORY. If the **high bit isn't set**, the lower 31 **bits point** to an IMAGE_RESOURCE_DATA_ENTRY structure. The **IMAGE_RESOURCE_DATA_ENTRY** structure contains the location of the resource's raw data, its size, and its code page.

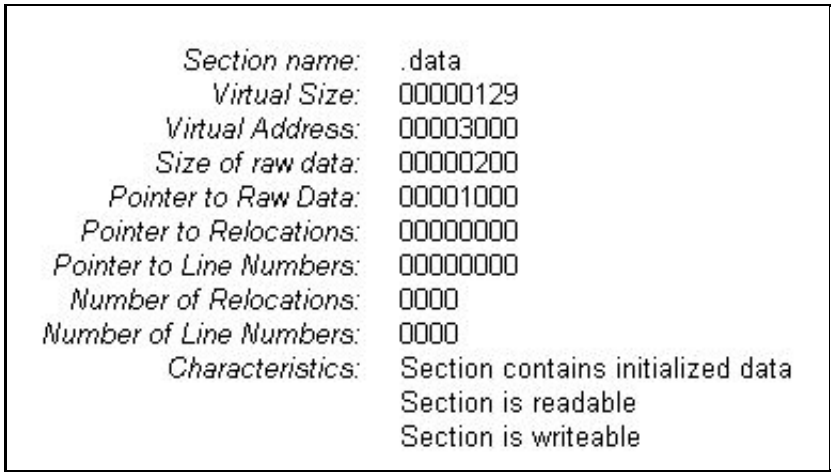
To go further into the resource formats, for each resource type (dialogs, menus, and so on) if you're interested, the RESFMT.TXT file from the Win32 SDK has a detailed description of all the resource type formats.

Data - Data Section Header



'.data':padded with 3 zeros = 8 bytes, **AppDataTrueSize:** virtual size129h 4 bytes, **AppBaseOfData**: virtual address 3000h 4 bytes, **AppDataAlignedSize**: size of raw data 200h 4 bytes, **AppStartOfData**: pointer to raw data 1000h bytes, 12 bytes not used,The **flags** describing how the section's memory should be treated **DataCharacteristics**: 4bytes D$ 0_C0000040 ; readable, writeable, initialized data.

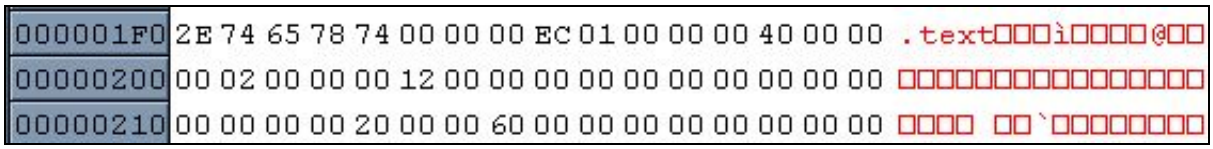
The above as seen in Quickview. (1C0h thru 1E0h)



As we can see the .data section starts at 1000h in the file and is 200h bytes in length...

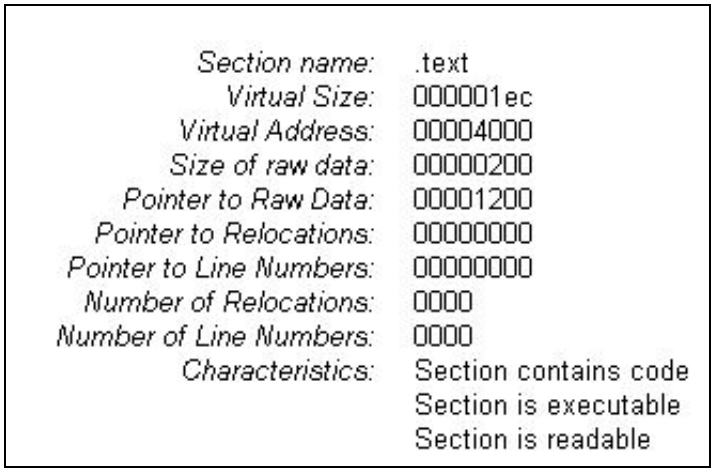
Text - Data Section Header

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



'.text' padded with 3 zeros = 8 bytes, **AppTrueCodeSize**: virtual size 4 bytes; true size of code in file, **AppCodeRVAoffset**: 4 bytes, RVA offset (aligned on 01000 boundary) **AppFileSizeOfCode**:size of raw data 4 bytes, file aligned size of code (200h aligned) , **AppStartOfCode**:pointer to raw data 4 bytes pointer to code (true first code in file - not entry point-). **The four 'dummy' D$ and W$ have no meaning in EXE and DLL files:** D$ 00 ; dummy reloc ptr D$ 00 ; dummy line number ptr W$ 00 ; dummy reloc number W$ 00 ; dummy number of line number = 12 bytes not used. CodeCharacteristics: **DataCharacteristics**:D$ 0_60000020 ; readable, runable/executable.

The above as seen in Quickview. (1F0h thru 21Fh)



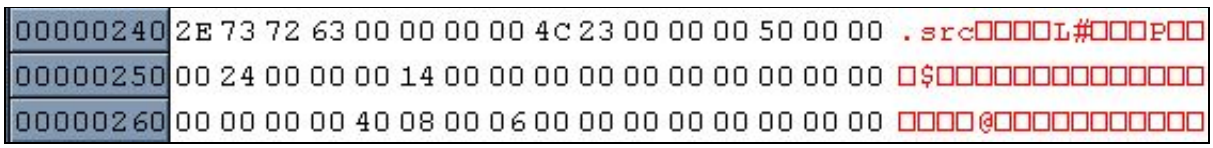
As we can see the .text section starts at 1200h in the file and is 200h bytes in length...

RosAsm 'Source Code' - Source Section Header:

As Betov has hidden the RosAsm 'source code' in the file, Quickview does not see it nor display it.

Used by RosAsm only (not by the loader which only sees 4 sections, not the actual 5).

Can you figure out how he does it ? Clue: Go back to Image File Header....



'.src': padded with 4 zeros = 8 bytes, **AppSrcTrueSize** 234Ch 4 bytes, **AppBaseOfSrc** 5000h **RVA** 4 bytes, **AppSrcAlignedSize** 2400h 4 bytes, **AppStartOfSrc** 1400h 4 bytes, 12 bytes not used, **DataCharacteristics**: D$ 06000840 ; Not readable initialized data; don't keep; don't cache...

RosAsm Base3.exe source code starts at 1400h and is 234Ch in length and continues to the **EOF**.....

EOF is commonly used to signify the end of file...

The above IMAGE_DATA_DIRECTORYs in **178h** to **267h** are followed by unused space and are padded with zeros to the file boundary **3FFh**.... As we saw above the Import Section begins at 400 H , Resource Section at 800h, Data Section at 1000h, Text at 1200h, RosAsm source at 1400h. Each of these sections are also padded with zeros to the file boundaries, which in our files is 200h > 512 bytes and just happens to be the exact size of a single disk sector..... As we have also seen the RVAs and offsets are used by the Win32 loader to map the file into memory differently....

To see how the file is loaded into memory click here........

The Sections found in RosAsm Base3.exe

The **.idata** section contains information about functions (and data) that the module imports from other DLLs.

The **.rsrc** section contains all the resources for the module.

The **.data** section is where your initialized data goes.

The **.text** section is where all general-purpose code emitted by the compiler or assembler resides.

The **.src** section contains all the RosASM source code for the module.

Other Sections that you may encounter in other PE files

The **.bss** section is where any uninitialized static and global variables are stored.

The **.crt** is another initialized data section utilized by the Microsoft C/C++ run-time libraries (hence the name).

The **.edata** section is a list of the functions and data that the PE file exports for other modules.

The **.reloc** section holds a table of base relocations. A base relocation is an adjustment to an instruction or initialized variable value that's needed if the loader couldn't load the file where the linker assumed it would. If the loader is able to load the image at the linker's preferred base address, the loader completely ignores the relocation information in this section.

The **.tls** section, which refers to "thread local storage," and is related to the TlsAlloc family of Win32 functions. When dealing with a .tls section, the memory manager sets up the page tables so that whenever a process switches threads, a new set of physical memory pages is mapped to the .tls section's address space. This permits per-thread global variables.

The **.rdata** section is used for at least two things. First, in Microsoft linker-produced EXEs, the .rdata section holds the debug directory, which is only present in EXE files.The other useful portion of an .rdata section is the description string.

NOTE: There seems to be a wide latitude in the way PE files are written by the various Compilers/Linkers and if you look at a number of different PE files you will find that some of them may not adhere to what is found in this document. Nevertheless most of it applies and the Win32 Loader is capable of uploading them into RAM, therefore the loader seems to be quite flexible in its own right. It also seems that while the Linker Version is present in the Optional Header it is useless as it does not identify which linker it is.... Of course we do know that **RosAsm** wrote

our Base3.exe....

Home page <.>Links Page

Notes about **Relative Virtual Addresses**:The PE format makes heavy use of so-called RVAs. An RVA, aka "relative virtual address", is used to describe a memory address if you don't know the **image base** address. It is the value you need to add to the image base address to get the actual linear address. The base address is the address the PE image is loaded to in RAM, and may vary from one invocation to the next. **Example:** Suppose an executable file is loaded to address 400000h and program execution starts at RVA 4000h. The effective execution start will then be at the address 404000h. If the executable were loaded to 100000h, the execution start would be 104000h.

...Defined **directory indexes** are:

IMAGE_DIRECTORY_ENTRY_EXPORT (0)The directory of exported symbols. mostly used for DLLs.

IMAGE_DIRECTORY_ENTRY_IMPORT (1) The directory of imported symbols.

IMAGE_DIRECTORY_ENTRY_RESOURCE (2) Directory of resources.

IMAGE_DIRECTORY_ENTRY_EXCEPTION (3) Exception directory - structure and purpose unknown.

IMAGE_DIRECTORY_ENTRY_SECURITY (4) Security directory - structure and purpose unknown.

IMAGE_DIRECTORY_ENTRY_BASERELOC (5) Base relocation table. Neccessary for DLLs.

IMAGE_DIRECTORY_ENTRY_DEBUG (6) Debug directory - contents is compiler dependent. Moreover, many compilers stuff the debug information into the code section and don't create a separate section for it.

IMAGE_DIRECTORY_ENTRY_COPYRIGHT (7) Description string - some arbitrary copyright note or the like.

IMAGE_DIRECTORY_ENTRY_GLOBALPTR (8) Machine Value (MIPS GP) - structure and purpose unknown.

IMAGE_DIRECTORY_ENTRY_TLS (9) Thread local storage directory - structure unknown; contains variables that are declared "__declspec(thread)", i.e. per-thread global variable.

IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG (10) Load configuration directory - structure and purpose unknown.

IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (11) Bound import directory - see description of import directory.

IMAGE_DIRECTORY_ENTRY_IAT (12) Import Address Table - see description of import directory.

As an example, if we find at index 7 the 2 longwords 1200h and 33h, and the load address is 10000h, we know that the copyright data is at address 10000h+1200h (in whatever section there may be), and the copyright note is 33h bytes long. If a directory of a particular type is not used in a binary, the Size and VirtualAddresses are both set to 0h.

Flags:

What most programmers call flags, the COFF/PE format calls **characteristics**. This field is a set of flags that indicate the section's attributes (such as code/data, readable, or writeable,). For a complete list of all possible section attributes, see the IMAGE_SCN_XXX_XXX #defines in WINNT.H. Some of the more important flags are shown below:

0x00000020 This section **contains code**. Usually set in conjunction with the executable flag (0x80000000).

0x00000040 This section **contains initialized data**. Almost all sections except executable and the .bss section have this flag set.

0x00000080 This section **contains uninitialized data** (for example, the .bss section).

0x00000200 This section **contains comments** or some other type of information. A typical use of this section is the .drectve section emitted by the compiler, which contains commands for the linker.

0x00000800 This section's contents shouldn't be put in the final EXE file. These sections are used by the compiler/assembler to pass information to the linker.

0x02000000 This section **can be discarded**, since it's not needed by the process once it's been loaded. The most common discardable section is the base relocations (.reloc).

0x10000000 This section is **shareable**. When used with a DLL, the data in this section will be shared among all processes using the DLL. The default is for data sections to be nonshared, meaning that each process using a DLL gets its own copy of this section's data. In more technical terms, a shared section tells the memory manager to set the page mappings for this

section such that all processes using the DLL refer to the same physical page in memory. To make a section shareable, use the SHARED attribute at link time. For example

0x20000000 This section is **executable**. This flag is usually set whenever the "contains code" flag (0x00000020) is set.

0x40000000 This section is **readable**. This flag is almost always set for sections in EXE files.

0x80000000 The section is **writeable**. If this flag isn't set in an EXE's section, the loader should mark the memory mapped pages as read-only or execute-only. Typical sections with this attribute are .data and .bss. Interestingly, the .idata section also has this attribute set.

/ More Section characteristics.

0x00000000 // Reserved.

0x00000001 // Reserved.

0x00000002 // Reserved.

0x00000004 // Reserved.

0x00000008 // Reserved.

0x00000010 // Reserved.

0x00000020 // Section contains code.

0x00000040 // Section contains initialized data.

0x00000080 // Section contains uninitialized data.

0x00000100 // Reserved.

0x00000200 // Section contains comments or some other type of information.

0x00000400 // Reserved.

0x00000800 // Section contents will not become part of image.

0x00001000 // Section contents comdat.

0x00002000 // Reserved.

Obsolete 0x00004000

0x00008000 // Section content can be accessed relative to GP.

Acknowledgements:

To complete this tutorial I primarily used information that I learned from:

RosAsm source code, Matt Pietrek's article on MSDN & from B.Luevelsmeyer at iplan.heitec.net and other information, such as winnt.h file with Borland Bcc55, I had in my files.

Words of Matt Pietrek:

"A good understanding of the Portable Executable (PE) file format leads to a good understanding of the operating system. If you know what's in your DLLs and EXEs, you'll be a more knowledgeable programmer."
.

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

Home page < **>Links Page**