Microsoft Portable Executable and Common Object File Format Specification

Visual C++™ Business Unit

Microsoft Corporation

Revision 4.0 September 1993

Note This document is provided to aid in the development of tools and applications for Microsoft Windows NT� but is not guaranteed to be a complete specification in all respects. Microsoft reserves the right to alter this document without notice.

1. General Concepts

This document specifies the structure of executable (image) files and object files under Microsoft Windows NT�. These files are referred to as Portable Executable (PE) or Common Object File Format (COFF) files. The name �Portable Executable� refers to the fact that executable files can run on more than one platform�although in MS-DOS�, it is common to run only a small stub program. Certain concepts appear repeatedly througout the specification and are described in the following table:

Name	Description
Image file	Executable file: either a .EXE file or a DLL. An image file can be thought of as a �memory image.� The term �image file� is usually used instead of �executable file,� because the latter sometimes is taken to mean only a .EXE file.
Object file	A file given as input to the linker. The linker produces an image file, which in turn is used as input by the loader. The term �object file� does not necessarily imply any connection to object-oriented programming.
RVA	Relative Virtual Address. In an image file, an RVA is always the address of an item once loaded into memory, with the base address of the image file subtracted from it. The RVA of an item will almost always differ from its position within the file on disk (File Pointer).In an object file, an RVA is less meaningful because memory locations are not assigned. In this case, an RVA would be an address within a section (see below), to which a relocation is later applied during linking. For simplicity, compilers should just set the first RVA in each section to zero.
Virtual Address (VA)	Same as RVA (see above), except that the base address of the image file is not subtracted. The address is called a �Virtual Address� because Windows NT creates a distinct virtual address space for each process, independent of physical memory. For almost all purposes, a virtual address should be considered just an address. A virtual address is not as predictable as an RVA, because the loader might not load the image at its preferred location.

File pointer	Location of an item within the file itself, before being processed by the linker (in the case of object files) or the loader (in the case of image files). In other words, this is a position within the file as stored on disk.
Date/Time Stamp	Date/time stamps are used in a number of places in a PE/COFF file, and for different purposes. The format of each such stamp, however, is always the same: that used by the time functions in the C run-time library.
Section	A section is the basic unit of code or data within a PE/COFF file. In an object file, for example, all code can be combined within a single section, or (depending on compiler behavior) each function can occupy its own section. With more sections, there is more file overhead, but the linker is able to link in code more selectively. A section is vaguely similar to a segment in Intel� 8086 architecture. All the raw data in a section must be loaded contiguously. In addition, an image file can contain a number of sections, such as .tls or .debug, that have special purposes.

2. Overview

Figures 1 and 2 illustrate the Microsoft PE executable format and the Microsoft COFF object-module format.

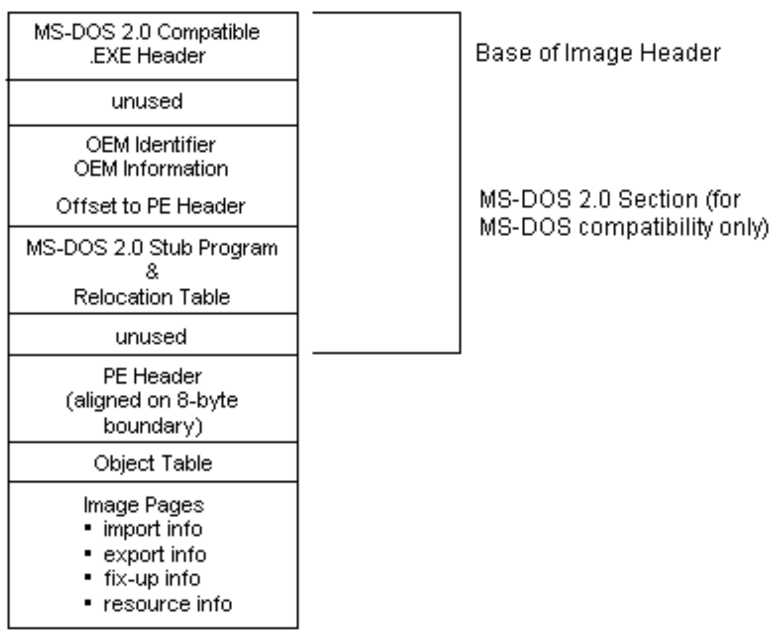

Figure 1. Typical 32-Bit Portable .EXE File Layout

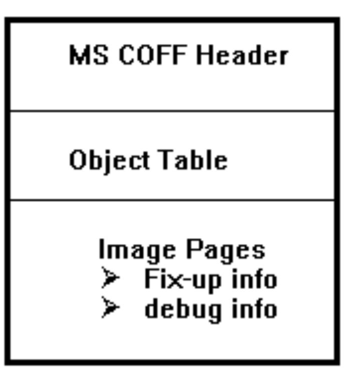

Figure 2. Typical 32-Bit COFF Object Module Layout

3. File Headers

The PE/COFF file headers consist of an MS-DOS stub, file signature, COFF Header, and Optional Header. An object file contains only the COFF Header, but an image file contains all the headers. In both cases, the file headers are followed immediately by section headers.

3.1. MS-DOS Stub (Image Only)

The MS-DOS Stub is a valid application that runs under MS-DOS and is placed at the front of the .EXE image. The linker places a default stub here, which prints out the message �This program cannot be run in DOS mode� when the image is run in MS-DOS. The user can specify another stub by using the /STUB linker option.

At location 0x3c, the stub has the file offset to the Portable Executable (PE) signature. This information enables Windows NT to properly execute the image file, even though it has a DOS Stub. This file offset is placed at location 0x3c during linking.

3.2. Signature (Image Only)

After the MS-DOS stub, there is a 4-byte signature identifying the file as a Win32� image file. Currently, this signature is �PE\0\0� (the letters �P� and �E� followed by two null bytes).

3.3. COFF Header (Object & Image)

At the beginning of an object file, or immediately after the signature of an image file, there is a standard COFF header of the following format:

Offset	Size	Field	Description
0	2	**Machine**	Number identifying type of target machine. See Section 3.3.1, �Machine Types, � for more information.
2	2	**Number of Sections**	Number of sections; indicates size of the Section Table, which immediately follows the headers.
4	4	**Time/Date Stamp**	Time and date the file was created.
8	4	**Pointer to Symbol Table**	Offset, within the COFF file, of the symbol table.
12	4	**Number of Symbols**	Number of entries in the symbol table. This data can be used in locating the string table, which immediately follows the symbol table.
16	2	**Optional Header Size**	Size of the optional header, which is included for executable files but not object files. An object file should have a value of 0 here. The format is described in the section �Optional Header.�
18	2	**Characteristics**	Flags indicating attributes of the file. See Section 3.3.2, �Characteristics,� for specific flag values.

3.3.1. Machine Types

The Machine field has one of the following values, defined below, which specify its machine (CPU) type. An image file can be run only on the specified machine, or a system emulating it.

Constant	Value	Description
IMAGE_FILE_MACHINE_UNKNOWN	0x0	Contents assumed to be applicable to any machine type.
IMAGE_FILE_MACHINE_I386	0x14c	Intel 386.
IMAGE_FILE_MACHINE_R4000	0x166	MIPS� little endian.

IMAGE_FILE_MACHINE_ALPHA	0x184	Alpha AXP�.

3.3.2. Characteristics

The Characteristics field contains flags that indicate attributes of the object or image file. The following flags are currently defined:

Flag	Value	Description
IMAGE_FILE_RELOCS_STRIPPED	0x0001	Image only. Indicates that the file does not contain base relocations and must therefore be loaded at its preferred base address. If the base address is not available, the loader reports an error. Operating systems running on top of MS-DOS (Win32s�) are generally not able to use the preferred base address and so cannot run these images.
IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	Image only. Indicates that the image file is valid and can be run. If this flag is not set, it generally indicates a linker error.
IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004	COFF line numbers have been removed.
IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008	COFF symbol table entries for local symbols have been removed.
IMAGE_FILE_MINIMAL_OBJECT	0x0010	Reserved for future use.
IMAGE_FILE_UPDATE_OBJECT	0x0020	Reserved for future use.
IMAGE_FILE_16BIT_MACHINE	0x0040	Machine based on 16-bit-word architecture. This flag will generally not be set for Windows NT.
IMAGE_FILE_BYTES_REVERSED_LO	0x0080	Little endian: LSB precedes MSB in memory.
IMAGE_FILE_32BIT_MACHINE	0x0100	Machine based on 32-bit-word architecture.
IMAGE_FILE_DEBUG_STRIPPED	0x0200	Debugging information removed from image file.
IMAGE_FILE_PATCH	0x0400	Reserved for future use.
IMAGE_FILE_SYSTEM	0x1000	The image file is a system file, not a user program.
IMAGE_FILE_DLL	0x2000	The image file is a dynamic-link library (DLL). Such files are considered executable files for almost all purposes, although they cannot be directly run.
IMAGE_FILE_BYTES_REVERSED_HI	0x8000	Big endian: MSB precedes LSB in memory.

3.4. Optional Header (Image Only)

Every image file has an Optional Header that provides information to the loader. This header is optional in the sense that some files (specifically, object files) do not have it. For image files, this header is strictly required. This header is also referred to the PE Header.

The Optional Header itself has three major parts:

Offset	Size	Header part	Description
0	28	Standard fields	These are defined for all implementations of COFF, including UNIX�.
28	68	NT-specific fields	These include additional fields to support specific features of Windows NT (for example, subsystem).
96	128	Data directories	These fields are address/size pairs for special tables, found in the image file and used by Windows NT (for example, Import Table and Export Table).

3.4.1. Optional Header Standard Fields (Image Only)

The first nine fields of the Optional Header are standard fields, defined for every implementation of COFF. These fields contain general information useful for loading and running an executable file.

Offset	Size	Field	Description
0	2	**Magic**	Unsigned integer identifying the state of the image file. The most common number is 0413 octal (0x10B), identifying it as a normal executable file. 0407 (0x107) identifies a ROM image.
2	1	**LMajor**	Linker major version number.
3	1	**LMinor**	Linker minor version number.
4	4	**Code Size**	Size of the code (text) section, or first code section if there are multiple code sections.
8	4	**Initialized Data Size**	Size of the initialized data section, or first data section if there are multiple data sections.
12	4	**Uninitialized Data Size**	Size of the uninitialized data section (BSS), or first such section if there are multiple BSS sections.
16	4	**Entry Point RVA**	Address of entry point, relative to image base, when executable file is loaded into memory. For program images, this is the starting address. For DLL images, this is the address of the initialization function.
20	4	**Base Of Code**	Address of beginning of code section, when loaded into memory.
24	4	**Base Of Data**	Address of beginning of data section, when loaded into memory.

3.4.2. Optional Header Windows NT-Specific Fields (Image Only)

The next twenty-one fields are an extension to the COFF Optional Header format and contain additional information needed by the linker and loader in Windows NT.

Offset	Size	Field	Description
28	4	**Image Base**	Preferred address of first byte of image when loaded into memory; must be a multiple of 64K.
32	4	**Section Alignment**	Alignment (in bytes) of sections when loaded into memory. Must be a power of 2 between 512 and 256M inclusive. Default is 64K.
36	4	**File Alignment**	Alignment factor (in bytes) used to align pages in image file. The value should be a power of 2 between 512 and 64K inclusive.
40	2	**OS Major**	Major version number of required OS.
42	2	**OS Minor**	Minor version number of required OS.
44	2	**User Major**	Major version number of image.
46	2	**User Minor**	Minor version number of image.
48	2	**SubSys Major**	Major version number of subsystem.
50	2	**SubSys Minor**	Minor version number of subsystem.
52	4	Reserved	
56	4	**Image Size**	Size, in bytes, of image, including all headers; must be a multiple of Section Alignment.
60	4	**Header Size**	Combined size of MS-DOS Header, PE Header, and Object Table.
64	4	**File Checksum**	Image file checksum. The algorithm for computing is incorporated into IMAGHELP.DLL. The following are checked for validation at load time: all drivers, any DLL loaded at boot time, and any DLL that ends up in the server.
68	2	**SubSystem**	Subsystem required to run this image. See �Windows NT Subsystem� below for more information.
70	2	**DLL Flags**	Flags indicating when the initialization function is called. Set to zero if not a DLL. See the section �DLL Flags� below for more information.

72	4	**Stack Reserve Size**	Size of stack to reserve. Only the Stack Commit Size is committed; the rest is made available one page at a time, until reserve size is reached.
76	4	**Stack Commit Size**	Size of stack to commit.
80	4	**Heap Reserve Size**	Size of local heap space to reserve. Only the Heap Commit Size is committed; the rest is made available one page at a time, until reserve size is reached.
84	4	**Heap Commit Size**	Size of local heap space to commit.
88	4	**Loader Flags**	Flags that affect loader behavior. See the section �Loader Flags� below for more information.
92	4	**Number of Data Directories**	Number of data-dictionary entries in the remainder of the Optional Header. Each describes a location and size.

Windows NT Subsystem

The following values are defined for the Subsystem field of the Optional Header. They determine what, if any, Windows NT subsystem is required to run the image.

Constant	Value	Description
IMAGE_SUBSYSTEM_UNKNOWN	0	Unknown subsystem.
IMAGE_SUBSYSTEM_NATIVE	1	Image does not require a subsystem.
IMAGE_SUBSYSTEM_WINDOWS_GUI	2	Image runs in the Windows� graphical user interface (GUI) subsystem.
IMAGE_SUBSYSTEM_WINDOWS_CUI	3	Image runs in the Windows character subsystem.
IMAGE_SUBSYSTEM_OS2_CUI	5	Image runs in the OS/2� character subsystem.
IMAGE_SUBSYSTEM_POSIX_CUI	7	Image runs in the Posix character subsystem.

DLL Flags

The following values are defined for the DLL Flags field of the Optional Header. They determine under what situations the DLL initialization function is called. This initialization function is called for either initialization or termination, or both, depending on how the flags are set. Note that no initialization function need be present in the user�s source code, but in that case, all these flags must be turned off.

Flag	Value	Description
IMAGE_LIBRARY_PROCESS_INIT	0x0001	Function called just after process initialization.
IMAGE_LIBRARY_PROCESS_TERM	0x0002	Function called just before process termination.
IMAGE_LIBRARY_THREAD_INIT	0x0004	Function called just after thread initialization. This does not apply to the first thread, which is allocated during process initialization.
IMAGE_LIBRARY_THREAD_TERM	0x0008	Function called just before thread initialization; does not apply to the first thread allocated.

Loader Flags

The following flags supply additional directions to the loader, when the image is loaded into memory for execution. These flags affect behavior of a DLL if it has an initialization function.

Flag	Value	Description
IMAGE_LOADER_FLAGS_BREAK_ON_LOAD	0x0001	Halt prior to executing first instruction.
IMAGE_LOADER_FLAGS_DEBUG_ON_LOAD	0x0002	Break prior to executing first instruction; effect is similar to a breakpoint.

3.4.3. Optional Header Data Directories (Image Only)

Each data directory gives the address and size of a table or string used by Windows NT. These are all loaded into memory so that they can be used by the system at run time. A data directory is an eight-byte field that has the following declaration:

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   RVA;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

The first field, RVA, is the relative virtual address of the table. The RVA is the address of the table, when loaded, relative to the base address of the image. The second field gives the size in bytes. The data directories, which form the last part of the Optional Header, are listed below.

Offset	Size	Field	Description
96	8	**Export Table**	Export Table address and size.
104	8	**Import Table**	Import Table address and size
112	8	**Resource Table**	Resource Table address and size.
120	8	**Exception Table**	Exception Table address and size.
128	8	**Security Table**	Security Table address and size.
136	8	**Base Relocation Table**	Base Relocation Table address and size.
144	8	**Debug**	Debug data starting address and size.
152	8	**Copyright**	Copyright string address and length.
160	8	**Machine Value**	Security Table address and size.
168	8	**TLS Table**	Thread Local Storage (TLS) Table address and size.
176	8	**Load Config Table**	Load Configuration Table address and size.
184	40	Reserved	

4. Section Table (Section Headers)

Each row of the Section Table, in effect, is a section header. This table immediately follows the COFF optional header, if any. This positioning is required because the file header does not contain a direct pointer to the section table; the location of the section table is determined by calculating the location of the first byte after the headers.

The number of entries in the Section Table is given by the Number of Sections field in the COFF header. Entries in the Section Table are numbered starting from one. The code and data memory section entries are in the order chosen by the linker.

In an image file, the virtual addresses for sections must be assigned by the linker such that they are in ascending order and adjacent, and they must be a multiple of the Section Align value in the optional header.

Each section header (Section Table entry) has the following format, for a total of 40 bytes per entry:

Offset	Size	Field	Description
0	8	**Section Name**	An eight-byte, null-padded ASCII string. There is no terminating null if the string is exactly eight bytes long. For longer names, this field contains a slash (/) followed by ASCII representation of a decimal number: this number is an offset into the string table.
8	4	**Virtual Size**	Total size of the section when loaded. If this is greater than Physical Size, the section is zero-padded.
12	4	**RVA/Offset**	Address of the first byte of the section, when loaded into memory, relative to the image base. For object files, this field is the address of the first byte before relocation is applied; for simplicity, compilers can set this to zero. Otherwise, it is an arbitrary value that is subtracted from offsets during relocation.

16	4	**Size of Raw Data**	Physical file size of the initialized data. Must be a multiple of File Align in the PE Header, and must be less than or equal to Virtual Size.
20	4	**Pointer to Raw Data**	File pointer to section�s first page within the COFF file. Must be a multiple of File Align in the PE Header.
24	4	**Pointer to Relocs**	File pointer to beginning of relocation entries for the section.
28	4	**Pointer to Linenumbers**	File pointer to beginning of line-number entries for the section.
32	2	**Number of Relocs**	Number of relocation entries for the section.
34	2	**Number of Linenumbers**	Number of line-number entries for the section.
36	4	**Section Flags**	Flags describing section�s characteristics. See Section 4.1, �Section Flags,� for more information.

4.1. Section Flags

The Section Flags field indicates characteristics of the section.

Flag	Value	Description
IMAGE_SCN_TYPE_REGULAR	0x00000000	Reserved for future use.
IMAGE_SCN_TYPE_DUMMY	0x00000001	Reserved for future use.
IMAGE_SCN_TYPE_NO_LOAD	0x00000002	Reserved for future use.
IMAGE_SCN_TYPE_GROUPED	0x00000004	Section is grouped with other sections by name. See Section 4.2, �Grouped Sections.�
IMAGE_SCN_TYPE_NO_PAD	0x00000008	Section should not be padded to next boundary.
IMAGE_SCN_TYPE_COPY	0x00000010	Reserved for future use.
IMAGE_SCN_CNT_CODE	0x00000020	Section contains executable code.
IMAGE_SCN_CNT_INITIALIZED_DATA	0x00000040	Section contains initialized data.
IMAGE_SCN_CNT_UNINITIALIZED_DATA	0x00000080	Section contains uninitialized data.
IMAGE_SCN_LNK_OTHER	0x00000100	Reserved for future use.
IMAGE_SCN_LNK_INFO	0x00000200	Section contains comments or other information. (The **.drectve** section has this type.)
IMAGE_SCN_LNK_OVERLAY	0x00000400	Section contains an overlay.
IMAGE_SCN_LNK_REMOVE	0x00000800	Section will not become part of the image.
IMAGE_SCN_LNK_COMDAT	0x00001000	Section contains COMDAT data. See Section 5.5.6, �COMDAT Sections,� for more information. Object files only.
IMAGE_SCN_MEM_DISCARDABLE	0x02000000	Section can be discarded as needed.
IMAGE_SCN_MEM_NOT_CACHED	0x04000000	Section cannot be cached.
IMAGE_SCN_MEM_NOT_PAGED	0x08000000	Section is not pageable.
IMAGE_SCN_MEM_SHARED	0x10000000	Section can be shared in memory.
IMAGE_SCN_MEM_EXECUTE	0x20000000	Section can be executed as code.
IMAGE_SCN_MEM_READ	0x40000000	Section can be read.
IMAGE_SCN_MEM_WRITE	0x80000000	Section can be written to.

4.2. Grouped Sections (Object Only)

The �$� character (dollar sign) has a special interpretation in section names in object files.

When determining the image section that will contain the contents of an object section, the linker discards the �$� and all characters following it. Thus, an object section named **.text$X** will actually contribute to the **.text** section in the image.

However, the characters following the �$� determine the ordering of the contributions to the image section. All contributions with the same object-section name will be allocated contiguously in the image, and the blocks of contributions will be sorted in lexical order by object-section name. Therefore, everything in object files with section name **.text$X** will end up together, after the **.text$W** contributions and before the **.text$Y** contributions.

The section name in an image file will never contain a �$� character.

5. Other Contents of the File

The data structures described so far, up to and including the Section Table, are all located at a fixed offset from the beginning of the file (or from the PE header if the file is an image containing an MS-DOS stub).

The remainder of a COFF object or image file contains blocks of data that are not necessarily at any specific file offset. Instead the locations are defined by pointers in the Optional Header or a section header.

An exception is for images with a Section Alignment value (see the Optional Header description) of less than 4K, in which case there are constraints on the file offset of the section data, as described in the next section. Another exception is that debug information must be placed at the very end of an image file, becuase the loader does not map the **.debug** section into memory. The rule on debug information does not apply to object files, however.

5.1. Section Data

Initialized data for a section consists of simple blocks of bytes. There is no iterated-data mechanism as there is for 16-bit executable files. However, for sections containing all zeros, the section data need not be included.

The data for each section is located at the file offset given by the Pointer to Raw Data field in the section header, and the size of this data in the file is indicated by the Physical Size field. If the Physical Size is less than the Virtual Size, the remainder is padded with zeros.

In an image file, the section data must be aligned on a boundary as specified by the File Align field in the image optional header. Section data must appear in order of the RVA values for the corresponding sections (as do the individual section headers in the Section Table).

There are additional restrictions on image files for which the Section Align value in the Optional Header is less than 4K. For such files, the location of section data in the file must match its location in memory when the image is loaded, so that the physical offset for section data is the same as the RVA.

5.2. COFF Relocations

Object files contain COFF relocations, which specify how the section data should be modified when placed in the image file and subsequently loaded into memory.

Ordinary image files (those not placed in ROM) do not contain COFF relocations, since all symbols referenced have already been assigned addresses in a flat address space. An image file does contain relocation information in the form of base relocations in the .reloc section (unless the image has the IMAGE_FILE_RELOCS_STRIPPED attribute).

An image file can be placed in ROM, which means it contains two address spaces�code and data�instead of one. Such an image contains COFF relocations, allowing the loader to resolve references between code and data when they are loaded into different places. With the Windows NT SDK linker, you produce such images by using the -rom option.

For each section, there is an array of fixed-length records that are the section�s COFF relocations. The position and length of the array are specified in the section header. Each element of the array has the following format:

Offset	Size	Field	Description
0	4	**Offset**	Address of the item to which relocation is applied: this is the offset from the beginning of the section, plus the value of the section�s RVA/Offset field (see Section 4, �Section Table.�). For example, if the first byte of the section has an address of 0x10, the third byte has an address of 0x12.

4	4	**Symbol Table Index**	A zero-based index into the section�s symbol table. This symbol gives the address to be used for the relocation. If the specified symbol has section storage class, then the symbol�s address is the address with the first section of the same name.
8	2	**Type**	A value indicating what kind of relocation should be performed. Valid relocation types depend on machine type. See Section 5.2.1, �Type Indicators.�

If the symbol referred to (by the Symbol Table Index field) has storage class IMAGE_SYM_CLASS_STATIC, the symbol�s address is the beginning of the secion. The section is usually in the same file, except when the object file is part of an archive (library). In that case, the section may be found in any other object file in the archive that has the same archive-member name as the current object file. (The relationship with the archive-member name is used in the linking of import tables, i.e. the .idata section.)

5.2.1. Type Indicators

The Type field of the relocation record indicates what kind of relocation should be performed. Different relocation types are defined for each type of machine.

Intel386

The following relocation Type indicators are defined for Intel386 and compatible processors:

Constant	Value	Description
IMAGE_REL_I386_ABSOLUTE	0	No location is necessary; reference is to an absolute value.
IMAGE_REL_I386_DIR16	1	Direct 16-bit reference to the symbol�s virtual address.
IMAGE_REL_I386_REL16	2	Program-counter-relative 16-bit reference to the symbol�s virtual address. This is usually a code reference.
IMAGE_REL_I386_DIR32	3	Direct 32-bit reference to the symbol�s virtual address.
IMAGE_REL_I386_DIR32NB	7	Direct 32-bit reference to the symbol�s virtual address, base not included. This is a relative virtual address.
IMAGE_REL_I386_SEG12	9	Direct 32-bit reference to the segment-selector bits of a 32-bit virtual address.
IMAGE_REL_I386_SECTION	0xA	Reference to a section address. A common use is to facilitate references to debugging information, which reside in separate **.debug** sections.
IMAGE_REL_I386_SECREL	0xB	Reference to an offset from a section address. Has same general purpose as IMAGE_REL_I386_SECTION.
IMAGE_REL_I386_REL32	0x14	Program-counter-relative 32-bit reference to the symbol�s virtual address. This is usually a code reference.

MIPS Processors

The following relocation Type indicators are defined for MIPS processors:

Constant	Value	Description
IMAGE_REL_MIPS_ABSOLUTE	0	No location is necessary; reference is to an absolute value.
IMAGE_REL_MIPS_REFHALF	1	Direct reference to a 16-bit address.
IMAGE_REL_MIPS_REFWORD	2	Direct reference to a 32-bit address. Because of RISC architecture, such an address cannot fit into a single instruction, so this reference is likely pointer data.
IMAGE_REL_MIPS_JMPADDR	3	Displacement to a code (jump) address.
IMAGE_REL_MIPS_REFHI	4	Reference to the high portion of a 32-bit address. Used for the first instruction in a two-instruction sequence that loads a full address. REFHI must be followed by PAIR and REFLO relocations.
IMAGE_REL_MIPS_REFLO	5	Reference to the low portion of a 32-bit address.

IMAGE_REL_MIPS_GPREL	6	Reference that is relative to the Global Pointer (GP) register.
IMAGE_REL_MIPS_LITERAL	7	Same as IMAGE_REL_MIPS_GPREL.
IMAGE_REL_MIPS_SECTION	10	Reference to a section address. A common use is to facilitate references to debugging information, which reside in separate **.debug** sections.
IMAGE_REL_MIPS_SECREL	11	Reference to an offset from a section address. Has same general purpose as IMAGE_REL_MIPS_SECTION.
IMAGE_REL_MIPS_REFWORDNB	34	Direct reference to a 32-bit address which is relative to the image base. This address is a linker-generated thunk.
IMAGE_REL_MIPS_PAIR	35	Relocation connecting a REFHI and a REFLO relocation pair. A REFHI relocation must be followed by a relocation of this type.

Alpha Processors

The following relocation Type indicators are defined for Alpha processors:

Constant	Value	Description
IMAGE_REL_ALPHA_ABSOLUTE	0	No location is necessary; reference is to an absolute value.
IMAGE_REL_ALPHA_REFHALF	1	Direct reference to a 16-bit address.
IMAGE_REL_ALPHA_REFQUAD	2	32-bit address reference to a QUAD data element.
IMAGE_REL_ALPHA_GPREL32	3	Reference to a 32-bit address that is relative to the Global Pointer (GP) register.
IMAGE_REL_ALPHA_LITERAL	4	Low 16 bits of a 32-bit displacement from the Global Pointer (GP) register.
IMAGE_REL_ALPHA_LITUSE	5	High 16 bits of a 32-bit displacement from the GP register.
IMAGE_REL_ALPHA_GPDISP	6	Reference that is a displacement off of the Global Pointer (GP) register.
IMAGE_REL_ALPHA_BRADDR	7	Displacement to a code (jump) address.
IMAGE_REL_ALPHA_HINT	8	Hint to the processor indicating which cache line will be loaded into the Instruction cache, for the target of a jump. If correct, the hint eliminates the branch-delay-slot.
IMAGE_REL_ALPHA_INLINE_REFLONG	9	Reference to a 32-bit address spread over two instructions; followed by either an ABSOLUTE or MATCH relocation.
IMAGE_REL_ALPHA_REFHI	10	Reference to the high portion of a 32-bit address. Used for the first instruction in a two-instruction sequence that loads a full address. REFHI must be followed by PAIR and REFLO relocations.
IMAGE_REL_ALPHA_REFLO	11	Reference to the low portion of a 32-bit address.
IMAGE_REL_ALPHA_PAIR	12	Relocation connecting a REFHI and a REFLO relocation pair. The symbol table index contains a value that is the low 16 bits of the HI/PAIR/LO relocation sequence.
IMAGE_REL_ALPHA_MATCH	13	Relocation following an INLINE_REFLONG. The symbol table index does not indicate which symbol the relocation is tied to, but indicates the displacement in bytes of the instruction with the matching low address.
IMAGE_REL_ALPHA_SECTION	14	Reference to an offset from a section address. Has same general purpose as IMAGE_REL_ALPHA_SECTION.
IMAGE_REL_ALPHA_SECREL	15	Direct reference to a 32-bit address that is relative to the image base.
IMAGE_REL_ALPHA_REFLONGNB	16	Reference to a 32-bit address relative to the image base. This is always a linker-generated thunk.

5.3. COFF Line Numbers

COFF line numbers indicate the relationship between code and line-numbers in source files. The Microsoft format for COFF line numbers is similar to standard COFF, but it has been extended to allow a single section to relate to line numbers in multiple source files.

COFF line numbers consist of an array of fixed-length records. The location (file offset) and size of the array are specified in the section header. Each line-number record is of the following format:

Offset	Size	Field	Description
0	4	**Type (*)**	Union of two fields: Symbol Table Index and RVA. Whether Symbol Table Index or RVA is used depends on the value of Linenumber.
4	2	**Linenumber**	When nonzero, this field specifies a one-based line number. When zero, the Type field is interpreted as a Symbol Table Index for a function.

The Type field is a union of two four-byte fields, Symbol Table Index, and RVA:

Offset	Size	Field	Description
0	4	**Symbol Table Index**	Used when Linenumber is 0: index to symbol table entry for a function. This format is used to indicate the function that a group of line-number records refer to.
0	4	**RVA**	Used when Linenumber is greater than 0: relative virtual address of the executable code that corresponds to the source line indicated. (In an object file, the RVA is relative to the section.)

A line-number record, then, can either set the Linenumber field to 0 and point to a function definition in the Symbol Table, or else it can work as a standard line-number entry by giving a positive integer (line number) and the corresponding address in the object code.

A group of line-number entries always begins with the first format: the index of a function symbol. If this is the first line-number record in the section, then it is also the COMDAT symbol name for the function if the section�s COMDAT flag is set. (See Section 5.5.6, �COMDAT Sections.�) The function�s auxiliary record in the Symbol Table has a Pointer to Linenumbers field that points to this same line-number record.

A record identifying a function is followed by any number of line-number entries that give actual line-number information (Linenumber greater than zero). These entries are one-based, relative to the beginning of the function, and represent every source line in the function except for the first one.

For example, the first line-number record for the following example would specify the ReverseSign function (Symbol Table Index of ReverseSign, Linenumber set to 0). Then records with Linenumber values of 1, 2, and 3 would follow, corresponding to source lines as shown:

```
      // some code precedes ReverseSign function

     int   ReverseSign(int i)
 1:  {
 2:     return -1 * i;
 3:  }
```

5.4. COFF Symbol Table

The Symbol Table described in this section is inherited from the traditional COFF format. It is distinct from CodeView� information. A file may contain both a COFF Symbol Table and CodeView debug information, and the two are kept separate. Some Microsoft tools use the Symbol Table for limited but important purposes, such as communicating COMDAT information to the linker. Section names and file names, as well as code and data symbols, are listed in the Symbol Table.

The location of the Symbol Table is indicated in the COFF Header.

The Symbol Table is an array of records, each 18 bytes long. Each record is either a standard or auxiliary symbol-table record. A standard record defines a symbol or name, and has the following format:

Offset	Size	Field	Description
0	8	**Name (*)**	Name of the symbol, represented by union of three structures. An array of eight bytes is used if the name is not more than eight bytes long. See Section 5.4.1, �Symbol Name Representation, � for more information.
8	4	**Value**	Value associated with the symbol. The interpretation of this field depends on Section Number and Storage Class. A typical meaning is the relocatable address.
12	2	**Section Number**	Signed integer identifying the section, using a one-based index into the Section Table. Some values have special meaning defined in �Section Number Values.�
14	2	**Type**	A number representing type. Microsoft tools set this field to 0x20 (function) or 0x0 (not a function). See Section 5.4.3, �Type Representation,� for more information.
16	1	**Storage Class**	Enumerated value representing storage class. See Section 5.4.4, �Storage Class,� for more information.
17	1	**Number of Aux Symbols**	Number of auxiliary symbol table entries that follow this record.

Zero or more auxiliary symbol-table records immediately follow each standard symbol-table record. However, typically not more than one auxiliary symbol-table record follows a standard symbol-table record (except for .file records with long file names). Each auxiliary record is the same size as a standard symbol-table record (18 bytes), but rather than define a new symbol, the auxiliary record gives additional information on the last symbol defined. The choice of which of several formats to use depends on the Storage Class field. Currently-defined formats for auxiliary symbol table records are shown in �Auxiliary Symbol Records.�

Tools that read COFF symbol tables must ignore auxiliary symbol records whose interpretation is unknown. This allows the symbol table format to be extended to add new auxiliary records, without breaking existing tools.

5.4.1. Symbol Name Representation

The Name field in a symbol table consists of eight bytes that contain the name itself, if not too long, or else give an offset into the String Table. To determine whether the name itself or an offset is given, test the first four bytes for equality to zero.

Offset	Size	Field	Description
0	8	**Short Name**	An array of eight bytes. This array is padded with nulls on the right if the name is less than eight bytes long.
0	4	**Zeroes**	Set to all zeros if the name is longer than eight bytes.
4	4	**Offset**	Offset into the String Table.

5.4.2. Section Number Values

Normally, the Section Value field in a symbol table entry is a one-based index into the Section Table. However, this field is a signed integer and may take negative values. The following values, less than one, have special meanings:

Constant	Value	Description
IMAGE_SYM_UNDEFINED	0	Symbol record is not yet assigned a section. In object files, this setting is used for references to external symbols not defined by the symbol record.
IMAGE_SYM_ABSOLUTE	-1	The symbol has a value but is not an address. Examples are automatic variables and registers. Not used by all Microsoft tools.
IMAGE_SYM_DEBUG	-2	The symbol provides general type or debugging information but does not correspond to a section. Microsoft tools use this setting along with **.file** records (storage class FILE).

5.4.3. Type Representation

The Type field of a symbol table entry contains two bytes, each byte representing type information. The least-significant byte represents simple (base) data type, and the most-significant byte represents complex type, if any:

MSB	LSB
Complex type: none, pointer, function, array.	Base type: integer, floating-point, etc.

The following values are defined for base type, although Microsoft tools generally do not use this field, setting the least-significant byte to 0. Instead, CodeView information is used to indicate types. However, the possible COFF values are listed here for completeness.

Constant	Value	Description
IMAGE_SYM_TYPE_NULL	0	No type information, or unknown base type. Microsoft tools use this setting.
IMAGE_SYM_TYPE_VOID	1	No valid type; used with void pointers and functions.
IMAGE_SYM_TYPE_CHAR	2	Character (signed byte).
IMAGE_SYM_TYPE_SHORT	3	Two-byte signed integer.
IMAGE_SYM_TYPE_INT	4	Natural integer type (normally four bytes in Windows NT).
IMAGE_SYM_TYPE_LONG	5	Four-byte signed integer.
IMAGE_SYM_TYPE_FLOAT	6	Four-byte floating-point number.
IMAGE_SYM_TYPE_DOUBLE	7	Eight-byte floating-point number.
IMAGE_SYM_TYPE_STRUCT	8	Structure.
IMAGE_SYM_TYPE_UNION	9	Union.
IMAGE_SYM_TYPE_ENUM	10	Enumerated type.
IMAGE_SYM_TYPE_MOE	11	Member of enumeration (a specific value).
IMAGE_SYM_TYPE_BYTE	12	Byte; unsigned one-byte integer.
IMAGE_SYM_TYPE_WORD	13	Word; unsigned two-byte integer.
IMAGE_SYM_TYPE_UINT	14	Unsigned integer of natural size (normally, four bytes).
IMAGE_SYM_TYPE_DWORD	15	Unsigned four-byte integer.

The most significant byte specifies whether the symbol is a pointer to, function returning, or array of the base type specified in the least significant byte. Microsoft tools use this field only to indicate whether or not the symbol is a function, so that the only two resulting values are 0x0 and 0x20 for the Type field. However, other tools can use this field to communicate more information.

Constant	Value	Description
IMAGE_SYM_DTYPE_NULL	0	No derived type; the symbol is a simple scalar variable.
IMAGE_SYM_DTYPE_POINTER	1	Pointer to base type.
IMAGE_SYM_DTYPE_FUNCTION	2	Function returning base type.
IMAGE_SYM_DTYPE_ARRAY	3	Array of base type.

5.4.4. Storage Class

The Storage Class field of the Symbol Table indicates what kind of definition a symbol represents. The following table shows possible values. Note that the Storage Class field is an unsigned one-byte integer. The special value -1 should therefore be taken to mean its unsigned equivalent, 0xFF.

Although traditional COFF format makes use of many storage-class values, Microsoft tools rely on CodeView format for most symbolic information and generally use only four storage-class values: EXTERNAL (2), STATIC (3), FUNCTION (101), and STATIC (103). Except in the second column heading below, �Value� should be taken to mean the Value field of the symbol record (whose interpretation depends on the number found as the storage class).

Constant	Value	Description / Interpretation of Value Field
IMAGE_SYM_CLASS_END_OF_FUNCTION	-1 (0xFF)	Special symbol representing end of function, for debugging purposes.
IMAGE_SYM_CLASS_NULL	0	No storage class assigned.
IMAGE_SYM_CLASS_AUTOMATIC	1	Automatic (stack) variable. The Value field specifies stack frame offset.
IMAGE_SYM_CLASS_EXTERNAL	2	Used by Microsoft tools for external symbols. When combined with function type and Section Number greater than 0, the record is a function definition. With non-function type, the record is a data declaration, and the Value field indicates the size of the symbol (except in the case of weak externals, which have Value set to 0).
IMAGE_SYM_CLASS_STATIC	3	Used by Microsoft tools for section definitions. Value field is unused.
IMAGE_SYM_CLASS_REGISTER	4	Register variable. The Value field specifies register number.
IMAGE_SYM_CLASS_EXTERNAL_DEF	5	Symbol is defined externally.
IMAGE_SYM_CLASS_LABEL	6	Code label defined within the module. The Value field gives the label�s relocatable address.
IMAGE_SYM_CLASS_UNDEFINED_LABEL	7	Reference to a code label not defined.
IMAGE_SYM_CLASS_MEMBER_OF_STRUCT	8	Structure member. The Value field specifies nth member.
IMAGE_SYM_CLASS_ARGUMENT	9	Formal argument (parameter)of a function. The Value field specifies nth argument.
IMAGE_SYM_CLASS_STRUCT_TAG	10	Structure tag-name entry.
IMAGE_SYM_CLASS_MEMBER_OF_UNION	11	Union member. The Value field specifies nth member.
IMAGE_SYM_CLASS_UNION_TAG	12	Union tag-name entry.
IMAGE_SYM_CLASS_TYPE_DEFINITION	13	Typedef entry.
IMAGE_SYM_CLASS_UNDEFINED_STATIC	14	Static data declaration.
IMAGE_SYM_CLASS_ENUM_TAG	15	Enumerated type tagname entry.
IMAGE_SYM_CLASS_MEMBER_OF_ENUM	16	Member of enumeration. Value specifies nth member.
IMAGE_SYM_CLASS_REGISTER_PARAM	17	Register parameter.
IMAGE_SYM_CLASS_BIT_FIELD	18	Bit-field reference. Value specifies nth bit in the bit field.
IMAGE_SYM_CLASS_BLOCK	100	A **.bb** (beginning of block) or **.eb** (end of block) record. Value is the relocatable address of the code location.
IMAGE_SYM_CLASS_FUNCTION	101	Used by Microsoft tools for symbol records that define the extent of a function: begin function (named **.bf**), end function (**.ef**), and lines in function (**.lf**). For **.lf** records, Value gives the number of source lines in the function. For **.ef** records, Value gives the size of function code.
IMAGE_SYM_CLASS_END_OF_STRUCT	102	End of structure entry.
IMAGE_SYM_CLASS_FILE	103	Used by Microsoft tools, as well as traditional COFF format, for the source-file symbol record. The symbol is followed by an auxiliary record that names the file.
IMAGE_SYM_CLASS_SECTION	104	Definition of a section (Microsoft tools use STATIC storage class instead).
IMAGE_SYM_CLASS_WEAK_EXTERNAL	105	Weak external. Microsoft tools use EXTERNAL storage class instead. See Section 5.5.3, �Auxiliary Format 3: Weak Externals,� for more information.

5.5. Auxiliary Symbol Records

Auxiliary Symbol Table records always follow, and apply to, some standard Symbol Table record. An auxiliary record can have any format that the tools are designed to recognize, but 18 bytes must be allocated for them so that Symbol Table is maintained as an array of regular size. Currently, Microsoft tools recognize auxiliary formats for the following kinds of records: function definitions, function begin and end symbols (**.bf** and **.ef**), weak externals, filenames, and section definitions.

The traditional COFF design also includes auxiliary-record formats for arrays and structures. Microsoft tools do not use these, instead placing that symbolic information in CodeView format in the debug sections.

5.5.1. Auxiliary Format 1: Function Definitions

A symbol table record marks the beginning of a function defintion if all of the following are true: it has storage class EXTERNAL (2), a Type value indicating it is a function (0x20), and a section number greater than zero. Note that a symbol table record that has a section number of UNDEFINED (0) does not define the function and does not have an auxiliary record. Function-definition symbol records are followed by an auxiliary record with the format described below.

Offset	Size	Field	Description
0	4	**Tag Index**	Symbol-table index of the corresponding **.bf** (begin function) symbol record.
4	4	**Total Size**	Size of the executable code for the function itself. If the function is in its own section, the Size of Raw Data in the section header will be greater or equal to this field, depending on alignment considerations.
8	4	**Pointer to Linenumbers**	File offset of the first COFF line-number entry for the function, or zero if none exists. See Section 5.3, �COFF Line Numbers,� for more information.
12	4	**Pointer to Next Function**	Symbol-table index of the record for the next function. If the function is the last in the symbol table, this field is set to zero.
16	2	Unused.	

5.5.2. Auxiliary Format 2: .bf and .ef Symbols

For each function definition in the Symbol Table, there are three contiguous items that describe the beginning, ending, and number of lines. Each of these symbols has storage class FUNCTION (101):

- A symbol record named **.bf** (begin function). The Value field is unused.
- A symbol record named **.lf** (lines in function). The Value field gives the number of lines in the function.
- A symbol record named **.ef** (end of function). The Value field has the same number as the Total Size field in the function-definition symbol record.

The **.bf** and **.ef** symbol records (but not **.lf** records) are followed by an auxiliary record with the following format:

Offset	Size	Field	Description
0	4	Unused.	
4	2	**Line Number**	Actual ordinal line number (1, 2, 3, etc.) within source file, corresponding to the **.bf** or **.ef** record.
6	6	Unused.	
12	4	**Pointer to Next Function (.bf only)**	Symbol-table index of the next **.bf** symbol record. If the function is the last in the symbol table, this field is set to zero. Not used for **.ef** records.
16	2	Unused.	

5.5.3. Auxiliary Format 3: Weak Externals

�Weak externals� are a mechanism for object files allowing flexibility at link time. A module can contain an unresolved external symbol (sym1), but it can also include an auxiliary record indicating that if sym1 is not present at link time, another external symbol (sym2) is used to resolve references instead.

If a definition of sym1 is linked, then an external reference to the symbol is resolved normally. If a definition of sym1 is not linked, then all references to the weak external for sym1 refer to sym2 instead. The external symbol, sym2, must always be linked; typically it is defined in the module containing the weak reference to sym1.

Weak externals are represented by a Symbol Table record with EXTERNAL storage class, UNDEF section number, and a value of 0. The weak-external symbol record is followed by an auxiliary record with the following format:

Offset	Size	Field	Description
0	4	**Tag Index**	Symbol-table index of sym2, the symbol to be linked if sym1 is not found.
4	4	**Characteristics**	A value of 1 indicating that no library search for sym1 should be performed, or a value of 2 indicating that the linker should search all libraries.
8	10	Unused.	

Note that the Characteristics field is not defined in WINNT.H; instead, the Total Size field is used.

5.5.4. Auxiliary Format 4: Files

This format follows a symbol-table record with storage class FILE (103). The symbol name itself should be **.file**, and the auxiliary record that follows it gives the name of a source-code file.

Offset	Size	Field	Description
0	18	**File Name**	ASCII string giving the name of the source file; padded with nulls if less than maximum length.

5.5.5. Auxiliary Format 5: Section Definitions

This format follows a symbol-table record that defines a section: such a record has a symbol name that is the name of a section (such as **.text** or **.drectve**) and has storage class STATIC (3). The auxiliary record provides information on the section referred to. Thus it duplicates some of the information in the section header.

Offset	Size	Field	Description
0	4	**Length**	Size of section data; same as Size of Raw Data in the section header.
4	2	**Number Of Relocs**	Number of relocation entries for the section.
6	2	**Number Of Linenumbers**	Number of line-number entries for the section.
8	4	**Check Sum**	Checksum for communal data. Applicable if the IMAGE_SCN_LNK_COMDAT flag is set in the section header. See �COMDAT Sections� below, for more information.
12	2	**Number**	One-based index into the Section Table for the associated section; used when the COMDAT Selection setting is 5.
14	1	**Selection**	COMDAT selection number. Applicable if the section is a COMDAT section.
15	3	Unused.	

5.5.6. COMDAT Sections (Object Only)

The Selection field of the Section Definition auxiliary format is applicable if the section is a COMDAT section: a section that can be defined by more than one object file. (The flag IMAGE_SCN_LNK_COMDAT is set in the Section Flags field of the section header.) The way that the linker resolves the multiple definitions of COMDAT sections is determined by the Selection field.

There may be other symbols associated with each symbol record that defines a section. These associated symbol records share the same section number (see the Section Number field in the standard symbol-record format). At most, one of these associated symbol records should define an external symbol; this symbol, if present, is considered the �COMDAT symbol.�

Other sections in the same object must not contain relocations to the COMDAT symbol. When the symbol is to be referenced in the same module in which it has been defined, there must be an undefined (storage class UNDEFINED, or 0) external symbol in the COFF Symbol Table with the same name as the COMDAT symbol, and all relocations must point to the undefined one. This is a linker restriction that may be removed eventually.

Constant	Value	Description
IMAGE_COMDAT_SELECT_UNKNOWN	0	COMDAT behavior not defined.
IMAGE_COMDAT_SELECT_NODUPLICATES	1	The linker generates a warning if more than one section defines the same COMDAT symbol, but links in one of the sections anyway.
IMAGE_COMDAT_SELECT_ANY	2	Any section defining the same COMDAT symbol may be linked; the rest are removed.
IMAGE_COMDAT_SELECT_SAME_SIZE	3	The linker chooses an arbitrary section among the duplicate sections (having same COMDAT symbol); however, all must be the same size or the linker generates a warning.
IMAGE_COMDAT_SELECT_EXACT_MATCH	4	All duplicate sections must match exactly. One of them is linked. (This is not currently implemented in the linker).
IMAGE_COMDAT_SELECT_ASSOCIATIVE	5	The section is linked if a certain other COMDAT section is linked. This other section is indicated by the Number field of the auxiliary symbol record for the section definition. Use of this setting is useful for definitions that have components in multiple sections (for example, code in one and data in another), but where all must be linked together.

5.6. COFF String Table

Immediately following the COFF symbol table is the COFF string table. The position of this table is found by taking the symbol table address in the COFF header, and adding the number of symbols multiplied by the size of a symbol.

At the beginning of the COFF string table are 4 bytes containing the total size (in bytes) of the rest of the string table. This size does not include the Size field itself, so that the value in this location would be 0 if no strings were present.

Following the size are null-terminated strings pointed to by symbols in the COFF symbol table.

6. Special Sections

Typical COFF sections contain code or data that linkers and Win32 loaders process without special knowledge of the sections� contents. The contents are relevant only to the application being linked or executed.

However, some COFF sections have special meanings when found in object files and/or image files. Tools and loaders recognize these sections because they have special flags set in the section header, or because they are pointed to from special locations in the image optional header, or because the section name is �magic�: that is, the name indicates a special function of the section. (Even where the section name is not magic, the name is dictated by convention, so we will refer to a name.)

Some of the sections listed here are marked �(object only)� or �(image only)� to indicate that their special semantics are relevant only for object files or image files, respectively. A section that says �(image only)� may still appear in an object file as a way of getting into the image file, but the section has no special meaning to the linker, only to the image file loader.

6.1. The .debug Section

The **.debug** section is used in object files to contain compiler-generated debug information, and in image files to contain the total debug information generated. This section describes the packaging of debug information in object and image files. The actual format of CodeView debug information is not described here. See the document *CV4 Symbolic Debug Information Specification*.

The next section describes the format of the debug directory, which can be anywhere in the image. Subsequent sections describe the �groups� in object files that contain debug information.

6.1.1. Debug Directory (Image Only)

Image files contain an optional �debug directory� indicating what form of debug information is present and where it is. This directory consists of an array of �debug directory entries� whose location and size are indicated in the image optional header.

The debug directory may be in a discardable **.debug** section (if one exists) or it may be included in any other section in the image file, or not in a section at all.

Each debug directory entry identifies the location and size of a block of debug information. The RVA specified may be 0 if the debug information is not covered by a section header (i.e., it resides in the image file and is not mapped into the run-time address space). If it is mapped, the RVA is its address.

Here is the format of a debug directory entry:

Offset	Size	Field	Description
0	4	**Characteristics**	A reserved field intended to be used for flags, set to zero for now.
4	4	**Time/Date Stamp**	Time and date the debug data was created.
8	2	**Major Version**	Major version number of the debug data format.
10	2	**Minor Version**	Minor version number of the debug data format.
12	4	**Debug Type**	Format of debugging information: this field enables support of multiple debuggers. See Section 6.1.2, �Debug Type,� for more information.
16	4	**Size of Data**	Size of the debug data (not including the debug directory itself).
20	4	**RVA of Raw Data**	Address of the debug data when loaded, relative to the image base.
24	4	**Pointer to Raw Data**	File pointer to the debug data.

6.1.2. Debug Type

The following values are defined for the Debug Type field of the debug directory:

Constant	Value	Description
IMAGE_DEBUG_TYPE_UNKOWN	0	Unknown value, ignored by all tools.
IMAGE_DEBUG_TYPE_COFF	1	COFF debug information (line numbers, symbol table, and string table). This type of debug information is also pointed to by fields in the file headers.
IMAGE_DEBUG_TYPE_CODEVIEW	2	CodeView debug information. The format of the data block is described by the CV4 specification.
IMAGE_DEBUG_TYPE_FPO	3	Frame Pointer Omission (FPO) information. This information tells the debugger how to interpret non-standard stack frames, which use the EBP register for a purpose other than as a frame pointer.

If Debug Type is set to IMAGE_DEBUG_TYPE_FPO, the debug raw data is an array in which each member describes the stack frame of a function. Not every function in the image file need have FPO information defined for it, even though debug type is FPO. Those functions that do not have FPO information are assumed to have normal stack frames. The format for FPO information is defined as follows:

```
#define FRAME_FPO   0
#define FRAME_TRAP  1
#define FRAME_TSS   2

typedef struct _FPO_DATA {
    DWORD       ulOffStart;            // offset 1st byte of function code
    DWORD       cbProcSize;            // # bytes in function
    DWORD       cdwLocals;             // # bytes in locals/4
    WORD        cdwParams;             // # bytes in params/4

    WORD        cbProlog : 8;          // # bytes in prolog
    WORD        cbRegs   : 3;          // # regs saved
    WORD        fHasSEH  : 1;          // TRUE if SEH in func
    WORD        fUseBP   : 1;          // TRUE if EBP has been allocated
    WORD        reserved : 1;          // reserved for future use
    WORD        cbFrame  : 2;          // frame type
} FPO_DATA;
```

6.1.3. .debug$F (Object Only)

Object files can contain **.debug$F** sections whose contents are one or more FPO_DATA

records (Frame Pointer Omission information). See �IMAGE_DEBUG_TYPE_FPO� in table above.

The linker recognizes these **.debug$F** records. If debug information is being generated, the linker sorts the FPO_DATA records by procedure address, and generates a debug directory entry for them.

The compiler need not generate FPO records for procedures that have a standard frame format.

6.1.4. .debug$S (Object Only)

This section contains CV4 symbolic information: a stream of CV4 symbol records as described in the CV4 spec.

6.1.5. .debug$T (Object Only)

This section contains CV4 type information: a stream of CV4 type records as described in the CV4 spec.

6.1.6. Linker Support for Microsoft CodeView Debug Information

To support CodeView debug information, the linker:

- Generates the header and �NB05� signature.
- Packages the header with **.debug$S** and **.debug$T** sections from object files and synthetic (linker-generated) CV4 information, and creates a debug directory entry.
- Generates the subsection directory containing a pointer to each known subsection, including subsections that are linker-generated.
- Generates the sstModules subsection, which specifies the address and size of each module�s contribution(s) to the image address space.
- Generates the sstSegMap subsection, which specifies the address and size of each section in the image.
- Generates the sstPublicSym subsection, which contains the name and address of all externally defined symbols. (A symbol may be represented both by **.debug$S** information and by an sstPublicSym entry.)

6.2. The .drectve Section (Object Only)

A section is a �directive� section if it has the IMAGE_SCN_LNK_INFO flag set in the section header. By convention, such a section also has the name **.drectve**. The linker removes a **.drectve** section after processing the information, so the section does not appear in the image file being linked.

A **.drectve** section consists of a string of ASCII text. This string is a series of linker options (each option containing hyphen, option name, and any appropriate attribute) separated by spaces. The **.drectve** section should not have relocations or line numbers.

In a **.drectve** section, if the hyphen preceding an option is followed by a question mark (for example, �-?export�), and the option is not recognized as a valid directive, the linker must ignore it. This allows compilers and linkers to add new directives while maintaining compatibility with existing linkers, as long as the new directives are not required for the correct linking of the application. For example, if the directive enables a link-time optimization, it is acceptable if some linkers cannot recognize it.

6.3. The .edata Section (Image Only)

The export data section, named **.edata**, contains information about symbols that other images can access through dynamic linking. Exports are generally found in DLLs, but DLLs can import symbols as well.

An overview of the general structure of the export section is described below. The tables described are generally contiguous in the file and present in the order shown (though this is not strictly required). Only the Directory Table and Address Table are necessary for exporting symbols as ordinals. (An ordinal is an export accessed directly as an Export Address Table index.) The Name Pointer Table, Ordinal Table, and Export Name Table all exist to support use of export names.

Table Name	Description
Export Directory Table	A table with just one row (unlike the debug directory). This table indicates the locations and sizes of the other export tables.
Export Address Table	An array of RVAs of exported symbols. These are the actual addresses of the exported functions and data within the executable code and data sections. Other image files can import a symbol by using an index to this table (an ordinal) or, optionally, by using the public name that corresponds to the ordinal if one is defined.
Name Pointer Table	Array of pointers to the public export names, sorted in ascending order.
Ordinal Table	Array of the ordinals that correspond to members of the Name Pointer Table. The correspondence is by position; therefore, the Name Pointer Table and the Ordinal Table must have the same number of members. Each ordinal is an index into the Export Address Table.
Export Name Table	A series of null-terminated ASCII strings. Members of the Name Pointer Table point into this area. These names are the public names through which the symbols are imported and exported; they do not necessarily have to be the same as the private names used within the image file.

When another image file imports a symbol by name, the Name Pointer Table is searched for a matching string. If one is found, the associated ordinal is then determined by looking at the corresponding member in the Ordinal Table (that is, the member of the Ordinal Table with the same index as the string pointer found in the Name Pointer Table). The resulting ordinal is an index into the Export Address Table, which gives the actual location of the desired symbol. Every export symbol can be accessed by an ordinal.

Direct use of an ordinal is therefore more efficient, because it avoids the need to search the Name Pointer Table for a matching string. However, use of an export name is more mnemonic and does not require the user to know the table index for the symbol.

6.3.1. Export Directory Table

The export information begins with the Export Directory Table, which describes the remainder of the export information. The Export Directory Table contains address information that is used to resolve fix-up references to the entry points within this image.

Offset	Size	Field	Description
0	4	**Export Flags**	A reserved field, set to zero for now.
4	4	**Time/Date Stamp**	Time and date the export data was created.

8	2	**Major Version**	Major version number. The major/minor version number can be set by the user.
10	2	**Minor Version**	Minor version number.
12	4	**Name RVA**	Address of the ASCII string containing the name of the DLL. Relative to image base.
16	4	**Ordinal Base**	Starting ordinal number for exports in this image. This field specifies the starting ordinal number for the Export Address Table. Usually set to 1.
20	4	**Address Table Entries**	Number of entries in the Export Address Table.
24	4	**Number of Name Pointers**	Number of entries in the Name Pointer Table (also the number of entries in the Ordinal Table).
28	4	**Export Address Table RVA**	Address of the Export Address Table, relative to the image base.
32	4	**Name Pointer RVA**	Address of the Export Name Pointer Table, relative to the image base. The table size is given by Number of Name Pointers.
36	4	**Ordinal Table RVA**	Address of the Ordinal Table, relative to the image base.

6.3.2. Export Address Table

The Export Address Table contains the address of exported entry points and exported data and absolutes. An ordinal number is used to index the Export Address Table, after subtracting the value of the Ordinal Base field to get a true, zero-based index. (Thus, if the Ordinal Base is set to 1, a common value, an ordinal of 6 is the same as a zero-based index of 5.)

Each entry in the Export Address Table is a field that uses one of two formats, as shown in the following table. If the address specified is *not* within the export section (as defined by the address and length indicated in the Optional Header), the field is an Export RVA: an actual address in code or data. Otherwise, the field is a Forwarder RVA, which names a symbol in another DLL.

Offset	Size	Field	Description
0	4	**Export RVA**	Address of the exported symbol when loaded into memory, relative to the image base. For example, the address of an exported function.
0	4	**Forwarder RVA**	Pointer to a null-terminated ASCII string in the export section, giving the DLL name and the name of the export as �*dll.export*�.

A Forwarder RVA exports a definition from some other image, making it appear as if it were being exported by the current image. Thus the symbol is simultaneously imported and exported.

For example, in KERNEL32.DLL in Windows NT, the export named �HeapAlloc� is forwarded to the string �NTDLL.RtlAllocateHeap�. This allows applications to use the NT-specific module �NTDLL.DLL� without actually containing import references to it. The application�s import table references only �KERNEL32.DLL.� Therefore, the application is not specific to Windows NT and can run on any Win32 system.

6.3.3. Export Name Pointer Table

The Export Name Pointer Table is an array of addresses (RVAs) into the Export Name Table. The pointers are 32 bits each and are relative to the Image Base. The pointers are ordered lexically to allow binary searches.

An export name is defined only if the Export Name Pointer Table contains a pointer to it.

6.3.4. Export Ordinal Table

The Export Ordinal Table is an array of 16-bit indexes into the Export Address Table. The ordinals are biased by the Ordinal Base field of the Export Directory Table. In other words, the Ordinal Base must be subtracted from the ordinals to obtain true indexes into the Export Address Table.

The Export Name Pointer Table and the Export Ordinal Table form two parallel arrays, separated to allow natural field alignment. These two tables, in effect, operate as one table, in which the Export Name Pointer �column� points to a public (exported) name, and the Export Ordinal �column� gives the corresponding ordinal for that public name. A member of the Export Name Pointer Table and a member of the Export Ordinal Table are associated by having the same position (index) in their respective arrays.

Thus, when the Export Name Pointer Table is searched and a matching string is found at position i, the algorithm for finding the symbol�s address is:

```
i = Search_ExportNamePointerTable (ExportName);
ordinal = ExportOrdinalTable [i];
SymbolRVA = ExportAddressTable [ordinal - OrdinalBase];
```

6.3.5. Export Name Table

The Export Name Table contains the actual string data pointed to by the Export Name Pointer Table. The strings in this table are public names that can be used by other images to import the symbols; these public export names are not necessarily the same as the (private) symbol names that the symbols have in their own image file and source code, although they can be.

Every exported symbol has an ordinal value, which is just the index into the Export Address Table (plus the Ordinal Base value). Use of export names, however, is optional. Some, all, or none of the exported symbols can have export names. For those exported symbols that do have export names, corresponding entries in the Export Name Pointer Table and Export Ordinal Table work together to associate each name with an ordinal.

The structure of the Export Name Table is a series of ASCII strings, of variable length, each null terminated.

6.4. The .idata Section

All image files that import symbols, including virtually all .EXE files, have an **.idata** section. A typical file layout for the import information follows:

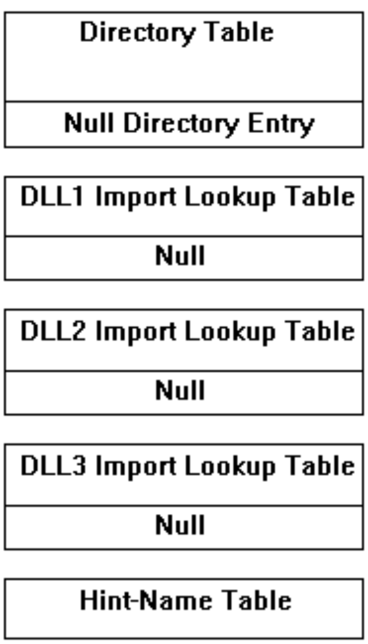

Figure 3. Typical Import Section Layout

6.4.1. Import Directory Table

The import information begins with the Import Directory Table, which describes the remainder of the import information. The Import Directory Table contains address information that is used to resolve fix-up references to the entry points within a DLL image. The Import Directory Table consists of an array of Import Directory Entries, one entry for each DLL the image references. The last directory entry is empty (filled with null values), which indicates the end of the directory table.

Each Import Directory entry has the following format:

Offset	Size	Field	Description
0	4	**Import Lookup Table RVA (Characteristics)**	Relative virtual address of the Import Lookup Table; this table contains a name or ordinal for each import. (The name �Characteristics� is used in WINNT.H but is no longer descriptive of this field.)
4	4	**Time/Date Stamp**	Set to zero until bound; then this field is set to the time/data stamp of the DLL.
8	4	**Fowarder Chain**	Index of first forwarder reference.
12	4	**Name RVA**	Address of ASCII string containing the DLL name. This address is relative to the image base.
16	4	**Import Address Table RVA (Thunk Table)**	Relative virtual address of the Import Address Table: this table is identical in contents to the Import Lookup Table until the image is bound.

6.4.2. Import Lookup Table

An Import Lookup Table is an array of 32-bit numbers, each using the bit-field format described below, in which bit 31 is the most significant bit. The collection of these entries describes all imports from the image to a given DLL. The last entry is set to zero (NULL) to indicate end of the table.

Bit(s)	Size	Bit Field	Description
31	1	**Ordinal/Name Flag**	If bit is set, import by ordinal. Otherwise, import by name. Bit is masked as 0x80000000.
30 - 0	31	**Ordinal Number**	Ordinal/Name Flag is 1: import by ordinal. This field is a 31-bit ordinal number.
30 - 0	31	**Hint/Name Table RVA**	Ordinal/Name Flag is 0: import by name. This field is a 31-bit address of a Hint/Name Table entry, relative to image base.

The lower 31 bits can be masked as 0x7FFFFFFF. In either case, the resulting number is a 32-bit integer or pointer in which the high bit is always zero (zero extension to 32 bits).

6.4.3. Hint/Name Table

One Hint/Name Table suffices for the entire import section. Each entry in the Hint/Name Table has the following format:

Offset	Size	Field	Description
0	4	**Hint**	Index into the Export Name Pointer Table. A match is attempted first with this value. If it fails, a binary search is performed on the DLL�s Export Name Pointer Table.
4	variable	**Name**	ASCII string containing name to import. This is the string that must be matched to the public name in the DLL. This string is case sensitive and terminated by a null byte.
*	0 or 1	**Pad**	A trailing zero pad byte appears after the trailing null byte, if necessary, to align the next entry on an even boundary.

6.4.4. Import Address Table

The structure and content of the Import Address Table are identifical to that of the Import Lookup Table, until the file is bound. During binding, the entries in the Import Address Table are overwritten with the 32-bit addresses of the symbols being imported: these addresses are the actual memory addresses of the symbols themselves (although technically, they are still called �virtual addresses�). The processing of binding is typically performed by the loader.

6.5. The .reloc Section (Image Only)

The Fix-Up Table contains entries for all fixups in the image. The Total Fix-Up Data Size in the Optional Header is the number of bytes in the fixup table. The fixup table is broken into blocks of fixups. Each block represents the fixups for a 4K page. Each block must start on a 32-bit boundary.

Fixups that are resolved by the linker do not need to be processed by the loader, unless the load image can�t be loaded at the Image Base specified in the PE Header.

6.5.1. Fixup Block

Each fixup block starts with the following structure:

Offset	Size	Field	Description
0	4	**Page RVA**	The image base plus the page RVA is added to each offset to create the virtual address of where the fixup needs to be applied.
4	4	**Block Size**	Total number of bytes in the fixup block, including the Page RVA and Block Size fields, as well as the Type/Offset fields that follow.

The Block Size field is then followed by any number of Type/Offset entries. Each entry has the following structure:

Offset	Size	Field	Description
0	2	**Type**	Value indicating which type of fixup is to be applied. These fixups are described in �Fixup Types.�
2	2	**Offset**	Offset from starting address specified in the Page RVA field for the block. This offset specifies where the fixup is to be applied.

To apply a fixup, a delta is calculated as the difference between the preferred base address, and the base where the image is actually loaded. If the image is loaded at its preferred base, the delta would be zero, and thus the fixups would not have to be applied.

6.5.2. Fixup Types

Constant	Value	Description
IMAGE_REL_BASED_ABSOLUTE	0	The fixup is skipped. This type can be used to pad a block.
IMAGE_REL_BASED_HIGH	1	The fixup adds the high 16 bits of the delta to the 16-bit field at Offset. The 16-bit field represents the high value of a 32-bit word.
IMAGE_REL_BASED_LOW	2	The fixup adds the low 16 bits of the delta to the 16-bit field at Offset. The 16-bit field represents the low half of a 32-bit word. This fixup is emitted for a RISC machine only when the image Object Align value is not the default of 64K.
IMAGE_REL_BASED_HIGHLOW	3	The fixup applies the 32-bit delta to the 32-bit field at Offset. The high 16 bits are located at Offset, and the low 16 bits are located in the next Offset record. The two need to be combined into a signed variable.
IMAGE_REL_BASED_HIGHADJ	4	This fixup requires a full 32-bit value.

IMAGE_REL_BASED_MIPS_JMPADDR	5	Fixup applies to a MIPS jump instruction.

6.6. The .tls Section

The **.tls** section provides direct PE/COFF support for static Thread Local Storage (TLS). TLS is a special storage class supported by Windows NT, in which a data object is not an automatic (stack) variable, yet it is local to each individual thread that runs the code. Thus, each thread can maintain a different value for a variable declared using TLS.

Note that any amount of TLS data can be supported by using the API calls **TlsAlloc**, **TlsFree**, **TlsSetValue**, and **TlsGetValue**. The PE/COFF implementation is an alternative approach to using the API, and it has the advantage of being simpler from the high-level-language programmer�s point of view. This implementation enables TLS data to be defined and initialized in a manner similar to ordinary static variables in a program. For example, in Microsoft Visual C++, a static TLS variable can be defined as follows, without using the Windows API:

```
__declspec (thread) int tlsFlag = 1;
```

To support this programming construct, the PE/COFF **.tls** section specifies the following information: initialization data, callback routines for per-thread initialization and termination, and the TLS index, explained in the following discussion.

Note: Statically declared TLS data objects can be used only in statically loaded image files. This fact makes it unreliable to use static TLS data in a DLL unless you know that the DLL, or anything statically linked with it, will never be loaded dynamically with the **LoadLibrary** API function.

Executable code accesses a static TLS data object through the following steps:

1. At link time, the linker sets the Address of Index field of the TLS Directory. This field points to a location where the program will expect to receive the TLS index.

   The Microsoft run-time library facilitates this process by defining a memory image of the TLS Directory and giving it the special name �_tls_used�. The linker looks for �_tls_used� and uses the data there to create the TLS Directory. Other compilers that support TLS and work with the Microsoft linker must use this same technique.
2. When a thread is created, the loader communicates the address of the thread�s TLS array by placing the address of the Thread Environment Block (TEB) in the FS register. A pointer to the TLS array is at the offset of 0x2C from the beginning of TEB.
3. The loader assigns the value of the TLS index to the place indicated by the Address of Index field.
4. The executable code retrieves the TLS index and also the location of the TLS array.
5. The code uses the TLS index and the TLS array location (multiplying the index by four and using it as an offset to the array) to get the address of the TLS data area for the given program and module. Each thread has its own TLS data area, but this is transparent to the program, which doesn�t need to know how data is allocated for individual threads.
6. An individual TLS data object is accessed as some fixed offset into the TLS data area.

The TLS array is an array of addresses that the system maintains for each thread. Each address in this array gives the location of TLS data for a given module (.EXE or DLL) within the program. The TLS index indicates which member of the array to use. (The index is a number, meaningful only to the system, that identifies the module).

6.6.1. The TLS Directory

The TLS Directory has the following format:

Offset	Size	Field	Description
0	4	**Raw Data Start VA (Virtual Address)**	Starting address of the TLS template. The template is a block of data used to initialize TLS data. The system copies all this data each time a thread is created, so it must not be corrupted. Note that this address is not an RVA; it is an address for which there should be a base relocation in the **.reloc**

			section.
4	4	**Raw Data End VA**	Address of the last byte of the TLS, except for the zero fill. As with the Raw Data Start VA, this is a virtual address, not an RVA.
8	4	**Address of Index**	Location to receive the TLS index, which the loader assigns. This location is in an ordinary data section, so it can be given a symbolic name accessible to the program.
12	4	**Address of Callbacks**	Pointer to an array of TLS callback functions. The array is null-terminated, so if there is no callback function supported, this field points to four bytes set to zero. The prototype for these functions is given below, in �TLS Callback Functions.�
16	4	**Size of Zero Fill**	The size in bytes of the template, beyond the initialized data delimited by Raw Data Start VA and Raw Data End VA. The total template size should be the same as the total size of TLS data in the image file. The zero fill is the amount of data that comes after the initialized nonzero data.
20	4	**Characteristics**	Reserved for possible future use by TLS flags.

6.6.2. TLS Callback Functions

The program can provide one or more TLS callback functions (though Microsoft compilers do not currently use this feature) to support additional initialization and termination for TLS data objects. A typical reason to use such a callback function would be to call constructors and destructors for objects.

Although there is typically no more than one callback function, a callback is implemented as an array to make it possible to add additional callback functions if desired. If there is more than one callback function, each function is called in the order its address appears in the array. A null pointer terminates the array. It is perfectly valid to have an empty list (no callback supported), in which case the callback array has exactly one member�a null pointer.

The prototype for a callback function (pointed to by a pointer of type PIMAGE_TLS_CALLBACK) has the same parameters as a DLL entry-point function:

```
typedef VOID
(NTAPI *PIMAGE_TLS_CALLBACK) (
    PVOID DllHandle,
    DWORD Reason,
    PVOID Reserved
    );
```

The Reserved parameter should be left set to 0. The Reason parameter can take the following values:

Setting	Value	Description
DLL_PROCESS_ATTACH	1	New process has started, including the first thread.
DLL_THREAD_ATTACH	2	New thread has been created (this notification sent for all but the first thread).
DLL_THREAD_DETACH	3	Thread is about to be terminated (this notification sent for all but the first thread).
DLL_PROCESS_DETACH	0	Process is about to terminate, including the original thread.

6.7. The .rsrc Section

Resources are indexed by a multiple level binary-sorted tree structure. The general design can incorporate 2**31 levels. By convention, however, Windows NT uses three levels:

- Type

- Name
- Language

A series of Resource Directory Tables relate all the levels in the following way: each directory table is followed by a series of directory entries, which give the name or ID for that level (Type, Name, or Language level) and an address of either a data description or another directory table. If a data description is pointed to, then the data is a leaf in the tree. If another directory table is pointed to, then that table lists directory entries at the next level down.

A leaf�s Type, Name, and Language IDs are determined by the path taken, through directory tables, to reach the leaf. The first table determines Type ID, the second table (pointed to by the directory entry in the first table) determines Name ID, and the third table determines Language ID.

The general structure of the **.rsrc** section is:

Data	Description
Resource Directory Tables (and Resource Directory Entries)	A series of tables, one for each group of nodes in the tree. All top-level (Type) nodes are listed in the first table. Entries in this table point to second-level tables. Each second-level tree has the same Type identifier but different Name identifiers. Third-level trees have the same Type and Name identifiers but different Language identifiers.Each individual table is immediately followed by directory entries, in which each entry has: 1) a name or numeric identifier, and 2) a pointer to a data description or a table at the next lower level.
Resource Directory Strings	Two-byte-aligned Unicode� strings, which serve as string data pointed to by directory entries.
Resource Data Description	An array of records, pointed to by tables, which describe the actual size and location of the resource data. These records are the leaves in the resource-description tree.
Resource Data	Raw data of the resource section. The size and location information in the Resource Data Descriptions delimit the individual regions of resource data.

6.7.1. Resource Directory Table

Each Resource Directory Table has the following format. This data structure should be considered the heading of a table, because the table actually consists of directory entries (see next section) as well as this structure:

Offset	Size	Field	Description
0	4	**Characteristics**	Resource flags, reserved for future use; currently set to zero.
4	4	**Time/Date Stamp**	Time the resource data was created by the resource compiler.
8	2	**Major Version**	Major version number, set by the user.
10	2	**Minor Version**	Minor version number.
12	2	**Number of Name Entries**	Number of directory entries, immediately following the table, that use strings to identify Type, Name, or Language (depending on the level of the table).
14	2	**Number of ID Entries**	Number of directory entries, immediately following the Name entries, that use numeric identifiers for Type, Name, or Language.

6.7.2. Resource Directory Entries

The directory entries make up the rows of a table. Each Resource Directory Entry has the following format. Note that whether the entry is a Name or ID entry is indicated by the Resource Directory Table, which indicates how many Name and ID entries follow it (remember that all the Name entries precede all the ID entries for the table). All entries for the table are sorted in ascending order: the Name entries by case-insensitive string, and the ID entries by numeric value.

Offset	Size	Field	Description
0	4	**Name RVA**	Address of string that gives the Type, Name, or Language identifier, depending on level of table.
0	4	**Integer ID**	32-bit integer that identifies Type, Name, or Language.
4	4	**Data Entry RVA**	High bit 0. Address of a Resource Data Entry (a leaf).
4	4	**Subdirectory RVA**	High bit 1. Lower 31 bits are the address of another Resource Directory Table (the next level down).

6.7.3. Resource Directory String

The Resource Directory String area consists of Unicode strings, which are word aligned. These strings are stored together after the last Resource Directory Entry and before the first Resource Data Entry. This minimizes the impact of these variable length strings on the alignment of the fixed-size directory entries. Each Resource Directory String has the following format:

Offset	Size	Field	Description
0	2	**Length**	Size of string, not including length field itself.
2	variable	**Unicode String**	Variable-length Unicode string data, word aligned.

6.7.4. Resource Data Entry

Each Resource Data Entry describes an actual unit of raw data in the Resource Data area, and has the following format:

Offset	Size	Field	Description
0	4	**Data RVA**	Address of a unit of resource data in the Resource Data area.
4	4	**Size**	Size, in bytes, of the resource data pointed to by the Data RVA field.
8	4	**Codepage**	Code page used to decode code point values within the resource data. Typically, the code page would be the Unicode code page.
12	4	Reserved (must be set to 0)	

6.7.5. Resource Example

The resource example shows the PE/COFF representation of the following resource data:

TypeId#	NameId#	Language ID	Resource Data
1	1	0	00010001
1	1	1	10010001
1	2	0	00010002
1	3	0	00010003
2	1	0	00020001
2	2	0	00020002
2	3	0	00020003
2	4	0	00020004

9	1	0	00090001
9	9	0	00090009
9	9	1	10090009
9	9	2	20090009

When this data is encoded, a dump of the PE/COFF Resource Directory results in the following output:

```
Offset  Data
0000:   00000000 00000000 00000000 00030000 (3 entries in this directory)
0010:   00000001 80000028   (TypeId #1, Subdirectory at offset 0x28)
0018:   00000002 80000050   (TypeId #2, Subdirectory at offset 0x50)
0020:   00000009 80000080   (TypeId #9, Subdirectory at offset 0x80)
0028:   00000000 00000000 00000000 00030000 (3 entries in this directory)
0038:   00000001 800000A0   (NameId #1, Subdirectory at offset 0xA0)
0040:   00000002 00000108   (NameId #2, data desc at offset 0x108)
0048:   00000003 00000118   (NameId #3, data desc at offset 0x118)
0050:   00000000 00000000 00000000 00040000 (4 entries in this directory)
0060:   00000001 00000128   (NameId #1, data desc at offset 0x128)
0068:   00000002 00000138   (NameId #2, data desc at offset 0x138)
0070:   00000003 00000148   (NameId #3, data desc at offset 0x148)
0078:   00000004 00000158   (NameId #4, data desc at offset 0x158)
0080:   00000000 00000000 00000000 00020000 (2 entries in this directory)
0090:   00000001 00000168   (NameId #1, data desc at offset 0x168)
0098:   00000009 800000C0   (NameId #9, Subdirectory at offset 0xC0)
00A0:   00000000 00000000 00000000 00020000 (2 entries in this directory)
00B0:   00000000 000000E8   (Language ID 0, data desc at offset 0xE8
00B8:   00000001 000000F8   (Language ID 1, data desc at offset 0xF8
00C0:   00000000 00000000 00000000 00030000 (3 entries in this directory)
00D0:   00000001 00000178   (Language ID 0, data desc at offset 0x178
00D8:   00000001 00000188   (Language ID 1, data desc at offset 0x188
00E0:   00000001 00000198   (Language ID 2, data desc at offset 0x198
00E8:   000001A8    (At offset 0x1A8, for TypeId #1, NameId #1, Language id #0
    00000004    (4 bytes of data)
    00000000    (codepage)
    00000000    (reserved)
00F8:   000001AC    (At offset 0x1AC, for TypeId #1, NameId #1, Language id #1
    00000004    (4 bytes of data)
    00000000    (codepage)
    00000000    (reserved)
0108:   000001B0    (At offset 0x1B0, for TypeId #1, NameId #2,
00000004    (4 bytes of data)
    00000000    (codepage)
    00000000    (reserved)
0118:   000001B4    (At offset 0x1B4, for TypeId #1, NameId #3,
    00000004    (4 bytes of data)
    00000000    (codepage)
    00000000    (reserved)
0128:   000001B8    (At offset 0x1B8, for TypeId #2, NameId #1,
    00000004    (4 bytes of data)
    00000000    (codepage)
    00000000    (reserved)
0138:   000001BC    (At offset 0x1BC, for TypeId #2, NameId #2,
    00000004    (4 bytes of data)
    00000000    (codepage)
00000000    (reserved)
0148:   000001C0    (At offset 0x1C0, for TypeId #2, NameId #3,
    00000004    (4 bytes of data)
    00000000    (codepage)
    00000000    (reserved)
0158:   000001C4    (At offset 0x1C4, for TypeId #2, NameId #4,
    00000004    (4 bytes of data)
    00000000    (codepage)
    00000000    (reserved)
0168:   000001C8    (At offset 0x1C8, for TypeId #9, NameId #1,
    00000004    (4 bytes of data)
    00000000    (codepage)
    00000000    (reserved)
0178:   000001CC    (At offset 0x1CC, for TypeId #9, NameId #9, Language id #0
00000004    (4 bytes of data)
    00000000    (codepage)
```

```
    00000000    (reserved)
0188:   000001D0    (At offset 0x1D0, for TypeId #9, NameId #9, Language id #1
    00000004    (4 bytes of data)
    00000000    (codepage)
    00000000    (reserved)
0198:   000001D4    (At offset 0x1D4, for TypeId #9, NameId #9, Language id #2
    00000004    (4 bytes of data)
    00000000    (codepage)
    00000000    (reserved)
```

The raw data for the resources follows:

```
01A8:   00010001
01AC:   10010001
01B0:   00010002
01B4:   00010003
01B8:   00020001
01BC:   00020002
01C0:   00020003
01C4:   00020004
01C8:   00090001
01CC:   00090009
01D0:   10090009
01D4:   20090009
```

Appendix: Example Object File

This section describes the PE/COFF object file produced by compiling the file HELLO2.C, which contains the following small C program:

```
main()
{
foo();
}

foo()
{
}
```

The commands used to compile HELLO.C (with debug information) and generate this example were the following (the -Gy option to the compiler is used, which causes each procedure to be generated as a separate COMDAT section):

```
cl -c -Zi -Gy hello2.c
link -dump -all hello2.obj >hello2.dmp
```

Here is the resulting file HELLO2.DMP: (The reader is encouraged to experiment with various other examples, in order to clarify the concepts described in this specification.)

```
Dump of file hello2.obj

File Type: COFF OBJECT

FILE HEADER VALUES
     14C machine (i386)
       7 number of sections
2BA23B9A time date stamp Sat Mar 13 11:52:58 1993
     26F file pointer to symbol table
      20 number of symbols
       0 size of optional header
       0 characteristics

SECTION HEADER #1
.drectve name
       0 physical address
```

```
       0 virtual address
      11 size of raw data
     12C file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
     A00 flags
         Info
         Remove
         (no align specified)

RAW DATA #1
00000000  2D 64 65 66 61 75 6C 74 | 6C 69 62 3A 4C 49 42 43  -default|lib:LIBC
00000010  20                                                           |

SECTION HEADER #2
.debug$S name
      11 physical address
      11 virtual address
      5B size of raw data
     13D file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42000048 flags
         No Pad
         Initialized Data
         Discardable
         (no align specified)
         Read Only

RAW DATA #2
00000000  01 00 00 00 11 00 09 00 | 00 00 00 00 0A 68 65 6C  ........|.....hel
00000010  6C 6F 32 2E 6F 62 6A 42 | 00 01 00 04 00 00 00 3B  lo2.objB|.......;
00000020  40 28 23 29 20 4D 69 63 | 72 6F 73 6F 66 74 20 43  @(#) Mic|rosoft C
00000030  2F 43 2B 2B 20 33 32 20 | 62 69 74 73 20 78 38 36  /C++ 32 |bits x86
00000040  20 43 6F 6D 70 69 6C 65 | 72 20 56 65 72 73 69 6F   Compile|r Versio
00000050  6E 20 38 2E 30 30 2E 58 | 58 58 58                 n 8.00.X|XXX

SECTION HEADER #3
   .text name
      6C physical address
      6C virtual address
      10 size of raw data
     198 file pointer to raw data
     1A8 file pointer to relocation table
     1B2 file pointer to line numbers
       1 number of relocations
       3 number of line numbers
60001020 flags
         Code
         Communal; sym= _main
         (no align specified)
         Execute Read

RAW DATA #3
00000000  55 8B EC 53 56 57 E8 00 | 00 00 00 5F 5E 5B C9 C3  U..SVW..|..._^[..

RELOCATIONS #3
      73 virtual address,        B symbol table index, REL32

LINENUMBERS #3
       9    0  sym=  _main
      72    1        77    2

SECTION HEADER #4
   .text name
      7C physical address
      7C virtual address
      10 size of raw data
```

```
     1C4 file pointer to raw data
       0 file pointer to relocation table
     1D4 file pointer to line numbers
       0 number of relocations
       2 number of line numbers
60001020 flags
         Code
         Communal; sym= _foo
         (no align specified)
         Execute Read

RAW DATA #4
00000000  55 8B EC 53 56 57 5F 5E | 5B C9 C3 00 00 00 00 00  U..SVW_^|[.......

LINENUMBERS #4
      15    0  sym=  _foo
      82    1
SECTION HEADER #5
.debug$S name
      8C physical address
      8C virtual address
      2E size of raw data
     1E0 file pointer to raw data
     20E file pointer to relocation table
       0 file pointer to line numbers
       1 number of relocations
       0 number of line numbers
42001048 flags
         No Pad
         Initialized Data
         Communal (no symbol)
         Discardable
         (no align specified)
         Read Only

RAW DATA #5
00000000  28 00 05 02 00 00 00 00 | 00 00 00 00 00 00 00 00  (.......|........
00000010  10 00 00 00 06 00 00 00 | 0B 00 00 00 00 00 00 00  ........|........
00000020  00 00 01 10 00 04 6D 61 | 69 6E 02 00 06 00        ......ma|in....

RELOCATIONS #5
      A8 virtual address,        6 symbol table index, DIR32

SECTION HEADER #6
.debug$S name
      BA physical address
      BA virtual address
      2D size of raw data
     218 file pointer to raw data
     245 file pointer to relocation table
       0 file pointer to line numbers
       1 number of relocations
       0 number of line numbers
42001048 flags
         No Pad
         Initialized Data
         Communal (no symbol)
         Discardable
         (no align specified)
         Read Only

RAW DATA #6
00000000  27 00 05 02 00 00 00 00 | 00 00 00 00 00 00 00 00  '.......|........
00000010  0B 00 00 00 06 00 00 00 | 06 00 00 00 00 00 00 00  ........|........
00000020  00 00 01 10 00 03 66 6F | 6F 02 00 06 00           ......fo|o....
RELOCATIONS #6
      D6 virtual address,        B symbol table index, DIR32

SECTION HEADER #7
.debug$T name
      E7 physical address
      E7 virtual address
```

```
      20 size of raw data
     24F file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42000048 flags
         No Pad
         Initialized Data
         Discardable
         (no align specified)
         Read Only

RAW DATA #7
00000000  01 00 00 00 1A 00 16 00 | 98 58 42 2B 25 00 00 00  ........|.XB+%...
00000010  0F 43 3A 5C 74 6D 70 5C | 6D 73 76 63 2E 70 64 62  .C:\tmp\|msvc.pdb
SYMBOL TABLE
000 00000000 .file                            DEBUG notype       Filename
    hello2.c
002 00000000 .drectve                         SECT1 notype       Static
    Section length   11, #relocs    0, #linenums    0
004 00000000 .debug$S                         SECT2 notype       Static
    Section length   5B, #relocs    0, #linenums    0
006 00000000 _main                            UNDEF notype ()    External
007 00000000 .text                            SECT3 notype       Static
    Section length   10, #relocs    1, #linenums    3, checksum        0, selection
    1 (pick no duplicates)
009 00000000 _main                            SECT3 notype ()    External
    tag index 0000000e size 00000010 lines 000001b2 next function 00000015
00B 00000000 _foo                             UNDEF notype ()    External
00C 00000000 .text                            SECT4 notype       Static
    Section length   10, #relocs    0, #linenums    2, checksum        0, selection
    1 (pick no duplicates)
00E 00000000 .bf                              SECT3 notype       BeginFunction
    line# 0002 end 00000017
010 00000003 .lf                              SECT3 notype       .bf or.ef
011 00000010 .ef                              SECT3 notype       EndFunction
    line# 0004
013 00000000 .debug$S                         SECT5 notype       Static
    Section length   2E, #relocs    1, #linenums    0, checksum        0, selection
    5 (pick associative Section 3)
015 00000000 _foo                             SECT4 notype ()    External
    tag index 00000017 size 0000000b lines 000001d4 next function 00000000
017 00000000 .bf                              SECT4 notype       BeginFunction
    line# 0007 end 00000000
019 00000002 .lf                              SECT4 notype       .bf or.ef
01A 0000000B .ef                              SECT4 notype       EndFunction
    line# 0008
01C 00000000 .debug$S                         SECT6 notype       Static
    Section length   2D, #relocs    1, #linenums    0, checksum        0, selection
    5 (pick associative Section 4)
01E 00000000 .debug$T                         SECT7 notype       Static
    Section length   20, #relocs    0, #linenums    0

     Summary
    .debug$T       20
       .text       20
    .debug$S       B6
    .drectve       11
```

Here is a hexadecimal dump of HELLO2.OBJ:

```
hello2.obj:
00000000   4c 01 07 00 9a 3b a2 2b 6f 02 00 00 20 00 00 00  L....;.+o... ...
00000010   00 00 00 00 2e 64 72 65 63 74 76 65 00 00 00 00  .....drectve....
00000020   00 00 00 00 11 00 00 00 2c 01 00 00 00 00 00 00  ........,.......
00000030   00 00 00 00 00 00 00 00 00 0a 00 00 2e 64 65 62  .............deb
00000040   75 67 24 53 11 00 00 00 11 00 00 00 5b 00 00 00  ug$S........[...
00000050   3d 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00  =...............
00000060   48 00 00 42 2e 74 65 78 74 00 00 00 6c 00 00 00  H..B.text...l...
00000070   6c 00 00 00 10 00 00 00 98 01 00 00 a8 01 00 00  l...............
```

```
00000080   b2 01 00 00 01 00 03 00 20 10 00 60 2e 74 65 78   ........ ..`.tex
00000090   74 00 00 00 7c 00 00 00 7c 00 00 00 10 00 00 00   t...|...|.......
000000a0   c4 01 00 00 00 00 00 00 d4 01 00 00 00 00 02 00   ................
000000b0   20 10 00 60 2e 64 65 62 75 67 24 53 8c 00 00 00    ..`.debug$S....
000000c0   8c 00 00 00 2e 00 00 00 e0 01 00 00 0e 02 00 00   ................
000000d0   00 00 00 00 01 00 00 00 48 10 00 42 2e 64 65 62   ........H..B.deb
000000e0   75 67 24 53 ba 00 00 00 ba 00 00 00 2d 00 00 00   ug$S........-...
000000f0   18 02 00 00 45 02 00 00 00 00 00 00 01 00 00 00   ....E...........
00000100   48 10 00 42 2e 64 65 62 75 67 24 54 e7 00 00 00   H..B.debug$T....
00000110   e7 00 00 00 20 00 00 00 4f 02 00 00 00 00 00 00   .... ...O.......
00000120   00 00 00 00 00 00 00 00 48 00 00 42 2d 64 65 66   ........H..B-def
00000130   61 75 6c 74 6c 69 62 3a 4c 49 42 43 20 01 00 00   aultlib:LIBC ...
00000140   00 11 00 09 00 00 00 00 00 0a 68 65 6c 6c 6f 32   ..........hello2
00000150   2e 6f 62 6a 42 00 01 00 04 00 00 00 3b 40 28 23   .objB.......;@(#
00000160   29 20 4d 69 63 72 6f 73 6f 66 74 20 43 2f 43 2b   ) Microsoft C/C+
00000170   2b 20 33 32 20 62 69 74 73 20 78 38 36 20 43 6f   + 32 bits x86 Co
00000180   6d 70 69 6c 65 72 20 56 65 72 73 69 6f 6e 20 38   mpiler Version 8
00000190   2e 30 30 2e 58 58 58 58 55 8b ec 53 56 57 e8 00   .00.XXXXU..SVW..
000001a0   00 00 00 5f 5e 5b c9 c3 73 00 00 00 0b 00 00 00   ..._^[..s.......
000001b0   14 00 09 00 00 00 00 00 72 00 00 00 01 00 77 00   ........r.....w.
000001c0   00 00 02 00 55 8b ec 53 56 57 5f 5e 5b c9 c3 00   ....U..SVW_^[...
000001d0   00 00 00 00 15 00 00 00 00 00 82 00 00 00 01 00   ................
000001e0   28 00 05 02 00 00 00 00 00 00 00 00 00 00 00 00   (...............
000001f0   10 00 00 00 06 00 00 00 0b 00 00 00 00 00 00 00   ................
00000200   00 00 01 10 00 04 6d 61 69 6e 02 00 06 00 a8 00   ......main......
00000210   00 00 06 00 00 00 06 00 27 00 05 02 00 00 00 00   ........'.......
00000220   00 00 00 00 00 00 00 00 0b 00 00 00 06 00 00 00   ................
00000230   06 00 00 00 00 00 00 00 00 00 01 10 00 03 66 6f   ..............fo
00000240   6f 02 00 06 00 d6 00 00 00 0b 00 00 00 06 00 01   o...............
00000250   00 00 00 1a 00 16 00 98 58 42 2b 25 00 00 00 0f   ........XB+%....
00000260   43 3a 5c 74 6d 70 5c 6d 73 76 63 2e 70 64 62 2e   C:\tmp\msvc.pdb.
00000270   66 69 6c 65 00 00 00 00 00 00 00 fe ff 00 00 67   file...........g
00000280   01 68 65 6c 6c 6f 32 2e 63 00 00 00 00 00 00 00   .hello2.c.......
00000290   00 00 00 2e 64 72 65 63 74 76 65 00 00 00 00 01   ....drectve.....
000002a0   00 00 00 03 01 11 00 00 00 00 00 00 00 00 00 00   ................
000002b0   00 00 00 00 00 00 00 2e 64 65 62 75 67 24 53 00   ........debug$S.
000002c0   00 00 00 02 00 00 00 03 01 5b 00 00 00 00 00 00   .........[......
000002d0   00 00 00 00 00 00 00 00 00 00 00 5f 6d 61 69 6e   ..........._main
000002e0   00 00 00 00 00 00 00 00 00 20 00 02 00 2e 74 65   ......... ....te
000002f0   78 74 00 00 00 00 00 00 00 03 00 00 00 03 01 10   xt..............
00000300   00 00 00 01 00 03 00 00 00 00 00 00 00 01 00 00   ................
00000310   00 5f 6d 61 69 6e 00 00 00 00 00 00 00 03 00 20   ._main.........
00000320   00 02 01 0e 00 00 00 10 00 00 00 b2 01 00 00 15   ................
00000330   00 00 00 00 00 5f 66 6f 6f 00 00 00 00 00 00 00   ....._foo.......
00000340   00 00 00 20 00 02 00 2e 74 65 78 74 00 00 00 00   ... ....text....
00000350   00 00 00 04 00 00 00 03 01 10 00 00 00 00 00 02   ................
00000360   00 00 00 00 00 00 00 01 00 00 00 2e 62 66 00 00   ...........bf..
00000370   00 00 00 00 00 00 00 03 00 00 00 65 01 00 00 00   ...........e....
00000380   00 02 00 00 00 00 00 00 00 17 00 00 00 00 00 2e   ................
00000390   6c 66 00 00 00 00 00 03 00 00 00 03 00 00 00 65   lf.............e
000003a0   00 2e 65 66 00 00 00 00 00 10 00 00 00 03 00 00   ..ef............
000003b0   00 65 01 00 00 00 00 04 00 00 00 00 00 00 00 00   .e..............
000003c0   00 00 00 00 00 2e 64 65 62 75 67 24 53 00 00 00   ......debug$S...
000003d0   00 05 00 00 00 03 01 2e 00 00 00 01 00 00 00 00   ................
000003e0   00 00 00 03 00 05 00 00 00 5f 66 6f 6f 00 00 00   ........._foo...
000003f0   00 00 00 00 00 04 00 20 00 02 01 17 00 00 00 0b   ....... ........
00000400   00 00 00 d4 01 00 00 00 00 00 00 00 00 2e 62 66   ..............bf
00000410   00 00 00 00 00 00 00 00 00 04 00 00 00 65 01 00   .............e..
00000420   00 00 00 07 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000430   00 2e 6c 66 00 00 00 00 00 02 00 00 00 04 00 00   ..lf............
00000440   00 65 00 2e 65 66 00 00 00 00 00 0b 00 00 00 04   .e..ef..........
00000450   00 00 00 65 01 00 00 00 00 08 00 00 00 00 00 00   ...e............
00000460   00 00 00 00 00 00 00 2e 64 65 62 75 67 24 53 00   ........debug$S.
00000470   00 00 00 06 00 00 00 03 01 2d 00 00 00 01 00 00   .........-......
00000480   00 00 00 00 00 04 00 05 00 00 00 2e 64 65 62 75   ...........debu
00000490   67 24 54 00 00 00 00 07 00 00 00 03 01 20 00 00   g$T.......... ..
000004a0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04   ................
000004b0   00 00 00                                          ...
```