Windows

Microsoft Portable Executable and Common Object File Format Specification

Revision 11 – June 20, 2017

Abstract

This specification describes the structure of executable (image) files and object files under the Windows® family of operating systems. These files are referred to as Portable Executable (PE) and Common Object File Format (COFF) files, respectively.

Note

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

This revision of the Microsoft Portable Executable and Common Object File Format Specification replaces all previous revisions of this specification.

For the latest information, see:
http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

Legal Notice

Microsoft Portable Executable and Common Object File Format Specification
Microsoft Corporation
Revision 10

Note: *This specification is provided to aid in the development of certain development tools for the Microsoft Windows platform. However, Microsoft does not guarantee that it is a complete specification in all respects, and cannot guarantee the accuracy of any information presented after the date of publication. Microsoft reserves the right to alter this specification without notice.*

Microsoft will grant a royalty-free license, under reasonable and non-discriminatory terms and conditions, to any Microsoft patent claims (if any exist) that Microsoft deems necessary for the limited purpose of implementing and complying with the required portions of this specification only in the software development tools known as compilers, linkers, and assemblers targeting Microsoft Windows.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this specification may be reproduced, stored in or introduced into a retrieval system, modified or used in a derivative work, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft.

Microsoft may have intellectual property rights covering subject matter in this specification. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this specification does not give you any license to any intellectual property rights, and no other rights are granted by implication, estoppel, or otherwise.

This specification is provided "AS IS." Microsoft makes no representations or warranties, express, implied, or statutory, as (1) to the information in this specification, including any warranties of merchantability, fitness for a particular purpose, non-infringement, or title; (2) that the contents of this specification are suitable for any purpose; nor (3) that the implementation of such contents will not infringe any third party patents, copyrights, trademarks, or other rights.

Microsoft will not be liable for any direct, indirect, special, incidental, or consequential damages arising out of or relating to any use or distribution of this specification.

Microsoft, MS-DOS, MSDN, Visual Studio, Visual C++, Win32, Windows, Windows NT, Windows Server, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The foregoing names and trademarks may not be used in any manner, including advertising or publicity pertaining to this specification or its contents without specific, written prior permission from the respective owners.

Contents

1. General Concepts

This document specifies the structure of executable (image) files and object files under the Microsoft® Windows® family of operating systems. These files are referred to as Portable Executable (PE) and Common Object File Format (COFF) files, respectively. The name "Portable Executable" refers to the fact that the format is not architecture specific.

Certain concepts that appear throughout this specification are described in the following table:

Name	Description
attribute certificate	A certificate that is used to associate verifiable statements with an image. A number of different verifiable statements can be associated with a file; one of the most useful ones is a statement by a software manufacturer that indicates what the message digest of the image is expected to be. A message digest is similar to a checksum except that it is extremely difficult to forge. Therefore, it is very difficult to modify a file to have the same message digest as the original file. The statement can be verified as being made by the manufacturer by using public or private key cryptography schemes. This document describes details about attribute certificates other than to allow for their insertion into image files.
date/time stamp	A stamp that is used for different purposes in several places in a PE or COFF file. In most cases, the format of each stamp is the same as that used by the time functions in the C run-time library. For exceptions, see the descripton of **IMAGE_DEBUG_TYPE_REPRO** (section 6.1.2). If the stamp value is 0 or 0xFFFFFFFF, it does not represent a real or meaningful date/time stamp.
file pointer	The 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.
linker	A reference to the linker that is provided with Microsoft Visual Studio®.
object file	A file that is 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.
reserved, must be 0	A description of a field that indicates that the value of the field must be zero for generators and consumers must ignore the field.
RVA	Relative virtual address. In an image file, the address of an item after it is loaded into memory, with the base address of the image file subtracted from it. The RVA of an item almost always differs 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 (described later in this table), to which a relocation is later applied during linking. For simplicity, a compiler should just set the first RVA in each section to zero.
section	The basic unit of code or data within a PE or COFF file. For example, all code in an object file 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 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 **.reloc**, which have special purposes.

Name	Description
VA	virtual address. Same as RVA, except that the base address of the image file is not subtracted. The address is called a "VA" because Windows creates a distinct VA space for each process, independent of physical memory. For almost all purposes, a VA should be considered just an address. A VA is not as predictable as an RVA because the loader might not load the image at its preferred location.

2. Overview

Figure 1 illustrates the Microsoft PE executable format.

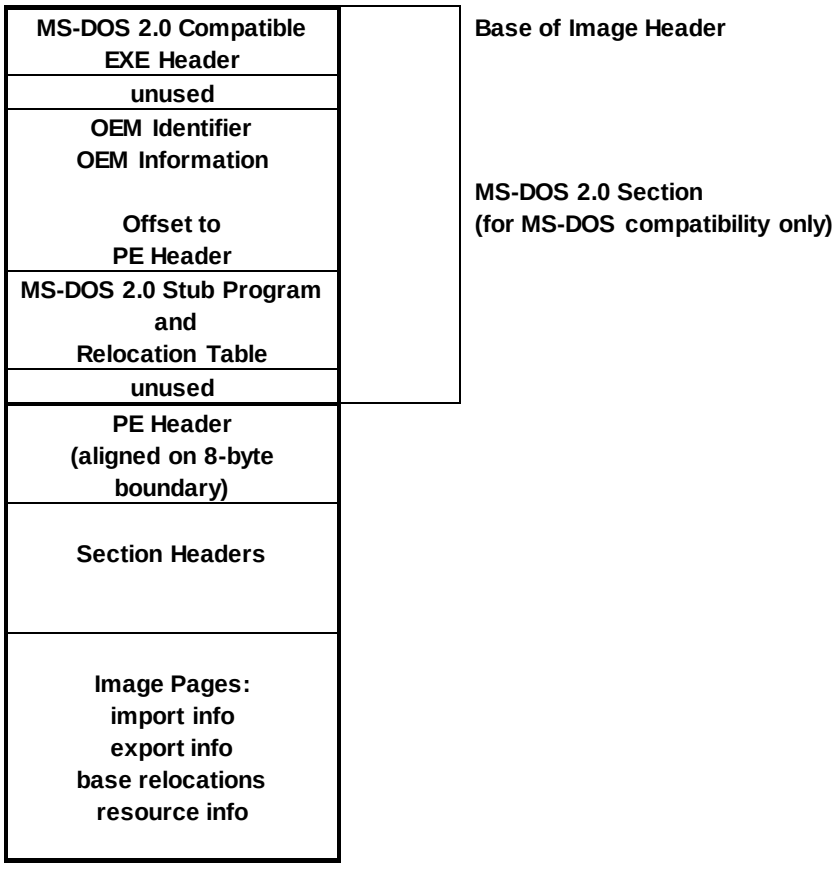

Figure 1. Typical Portable EXE File Layout

Figure 2 illustrates the Microsoft COFF object-module format:

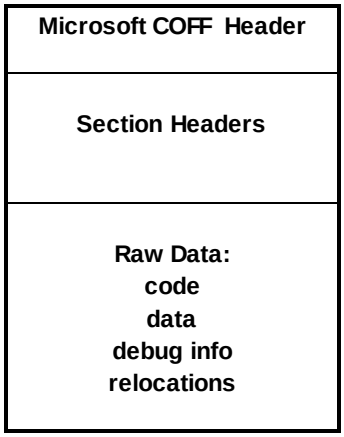

Figure 2. Typical COFF Object Module Layout

3. File Headers

The PE file header consists of a Microsoft MS-DOS® stub, the PE signature, the COFF file header, and an optional header. A COFF object file header consists of a COFF file header and an optional header. 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. It 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 a different stub by using the /STUB linker option.

At location 0x3c, the stub has the file offset to the PE signature. This information enables Windows to properly execute the image file, even though it has an MS-DOS stub. This file offset is placed at location 0x3c during linking.

3.2. Signature (Image Only)

After the MS-DOS stub, at the file offset specified at offset 0x3c, is a 4-byte signature that identifies the file as a PE format image file. This signature is "PE\0\0" (the letters "P" and "E" followed by two null bytes).

3.3. COFF File Header (Object and Image)

At the beginning of an object file, or immediately after the signature of an image file, is a standard COFF file header in the following format. Note that the Windows loader limits the number of sections to 96.

Offset	Size	Field	Description
0	2	Machine	The number that identifies the type of target machine. For more information, see section 3.3.1, "Machine Types."
2	2	NumberOfSections	The number of sections. This indicates the size of the section table, which immediately follows the headers.
4	4	TimeDateStamp	The low 32 bits of the number of seconds since 00:00 January 1, 1970 (a C run-time time_t value), that indicates when the file was created.

Offset	Size	Field	Description
8	4	PointerToSymbolTable	The file offset of the COFF symbol table, or zero if no COFF symbol table is present. This value should be zero for an image because COFF debugging information is deprecated.
12	4	NumberOfSymbols	The number of entries in the symbol table. This data can be used to locate the string table, which immediately follows the symbol table. This value should be zero for an image because COFF debugging information is deprecated.
16	2	SizeOfOptionalHeader	The size of the optional header, which is required for executable files but not for object files. This value should be zero for an object file. For a description of the header format, see section 3.4, “Optional Header (Image Only).”
18	2	Characteristics	The flags that indicate the attributes of the file. For specific flag values, see section 3.3.2, “Characteristics.”

3.3.1. Machine Types

The Machine field has one of the following values that specifies its CPU type. An image file can be run only on the specified machine or on a system that emulates the specified machine.

Constant	Value	Description
IMAGE_FILE_MACHINE_UNKNOWN	0x0	The contents of this field are assumed to be applicable to any machine type
IMAGE_FILE_MACHINE_AM33	0x1d3	Matsushita AM33
IMAGE_FILE_MACHINE_AMD64	0x8664	x64
IMAGE_FILE_MACHINE_ARM	0x1c0	ARM little endian
IMAGE_FILE_MACHINE_ARM64	0xaa64	ARM64 little endian
IMAGE_FILE_MACHINE_ARMNT	0x1c4	ARM Thumb-2 little endian
IMAGE_FILE_MACHINE_EBC	0xebc	EFI byte code
IMAGE_FILE_MACHINE_I386	0x14c	Intel 386 or later processors and compatible processors
IMAGE_FILE_MACHINE_IA64	0x200	Intel Itanium processor family
IMAGE_FILE_MACHINE_M32R	0x9041	Mitsubishi M32R little endian
IMAGE_FILE_MACHINE_MIPS16	0x266	MIPS16
IMAGE_FILE_MACHINE_MIPSFPU	0x366	MIPS with FPU
IMAGE_FILE_MACHINE_MIPSFPU16	0x466	MIPS16 with FPU
IMAGE_FILE_MACHINE_POWERPC	0x1f0	Power PC little endian
IMAGE_FILE_MACHINE_POWERPCFP	0x1f1	Power PC with floating point support
IMAGE_FILE_MACHINE_R4000	0x166	MIPS little endian
IMAGE_FILE_MACHINE_RISCV32	0x5032	RISC-V 32-bit address space
IMAGE_FILE_MACHINE_RISCV64	0x5064	RISC-V 64-bit address space
IMAGE_FILE_MACHINE_RISCV128	0x5128	RISC-V 128-bit address space
IMAGE_FILE_MACHINE_SH3	0x1a2	Hitachi SH3
IMAGE_FILE_MACHINE_SH3DSP	0x1a3	Hitachi SH3 DSP
IMAGE_FILE_MACHINE_SH4	0x1a6	Hitachi SH4
IMAGE_FILE_MACHINE_SH5	0x1a8	Hitachi SH5
IMAGE_FILE_MACHINE_THUMB	0x1c2	Thumb
IMAGE_FILE_MACHINE_WCEMIPSV2	0x169	MIPS little-endian WCE v2

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, Windows CE, and Microsoft Windows NT® and later. This 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. The default behavior of the linker is to strip base relocations from executable (EXE) files.
IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	Image only. This indicates that the image file is valid and can be run. If this flag is not set, it indicates a linker error.
IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004	COFF line numbers have been removed. This flag is deprecated and should be zero.
IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008	COFF symbol table entries for local symbols have been removed. This flag is deprecated and should be zero.
IMAGE_FILE_AGGRESSIVE_WS_TRIM	0x0010	Obsolete. Aggressively trim working set. This flag is deprecated for Windows 2000 and later and must be zero.
IMAGE_FILE_LARGE_ADDRESS_ AWARE	0x0020	Application can handle > 2-GB addresses.
	0x0040	This flag is reserved for future use.
IMAGE_FILE_BYTES_REVERSED_LO	0x0080	Little endian: the least significant bit (LSB) precedes the most significant bit (MSB) in memory. This flag is deprecated and should be zero.
IMAGE_FILE_32BIT_MACHINE	0x0100	Machine is based on a 32-bit-word architecture.
IMAGE_FILE_DEBUG_STRIPPED	0x0200	Debugging information is removed from the image file.
IMAGE_FILE_REMOVABLE_RUN_ FROM_SWAP	0x0400	If the image is on removable media, fully load it and copy it to the swap file.
IMAGE_FILE_NET_RUN_FROM_SWAP	0x0800	If the image is on network media, fully load it and copy it to the swap file.
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_UP_SYSTEM_ONLY	0x4000	The file should be run only on a uniprocessor machine.
IMAGE_FILE_BYTES_REVERSED_HI	0x8000	Big endian: the MSB precedes the LSB in memory. This flag is deprecated and should be zero.

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 required. An object file can have an optional header, but generally this header has no function in an object file except to increase its size.

Note that the size of the optional header is not fixed. The SizeOfOptionalHeader field in the COFF header must be used to validate that a probe into the file for a particular data directory does not go beyond SizeOfOptionalHeader. For more information, see section 3.3, "COFF File Header (Object and Image)."

The NumberOfRvaAndSizes field of the optional header should also be used to ensure that no probe for a particular data directory entry goes beyond the optional header. In addition, it is important to validate the optional header magic number for format compatibility.

The optional header magic number determines whether an image is a PE32 or PE32+ executable.

Magic number	PE format
0x10b	PE32
0x20b	PE32+

PE32+ images allow for a 64-bit address space while limiting the image size to 2 gigabytes. Other PE32+ modifications are addressed in their respective sections.

The optional header itself has three major parts.

Offset (PE32/PE32+)	Size (PE32/PE32+)	Header part	Description
0	28/24	Standard fields	Fields that are defined for all implementations of COFF, including UNIX.
28/24	68/88	Windows-specific fields	Additional fields to support specific features of Windows (for example, subsystems).
96/112	Variable	Data directories	Address/size pairs for special tables that are found in the image file and are used by the operating system (for example, the import table and the export table).

3.4.1. Optional Header Standard Fields (Image Only)

The first eight fields of the optional header are standard fields that are defined for every implementation of COFF. These fields contain general information that is useful for loading and running an executable file. They are unchanged for the PE32+ format.

Offset	Size	Field	Description
0	2	Magic	The unsigned integer that identifies the state of the image file. The most common number is 0x10B, which identifies it as a normal executable file. 0x107 identifies it as a ROM image, and 0x20B identifies it as a PE32+ executable.
2	1	MajorLinkerVersion	The linker major version number.
3	1	MinorLinkerVersion	The linker minor version number.

Offset	Size	Field	Description
4	4	SizeOfCode	The size of the code (text) section, or the sum of all code sections if there are multiple sections.
8	4	SizeOfInitializedData	The size of the initialized data section, or the sum of all such sections if there are multiple data sections.
12	4	SizeOfUninitializedData	The size of the uninitialized data section (BSS), or the sum of all such sections if there are multiple BSS sections.
16	4	AddressOfEntryPoint	The address of the entry point relative to the image base when the executable file is loaded into memory. For program images, this is the starting address. For device drivers, this is the address of the initialization function. An entry point is optional for DLLs. When no entry point is present, this field must be zero.
20	4	BaseOfCode	The address that is relative to the image base of the beginning-of-code section when it is loaded into memory.

PE32 contains this additional field, which is absent in PE32+, following BaseOfCode.

Offset	Size	Field	Description
24	4	BaseOfData	The address that is relative to the image base of the beginning-of-data section when it is loaded into memory.

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

The next 21 fields are an extension to the COFF optional header format. They contain additional information that is required by the linker and loader in Windows.

Offset (PE32/ PE32+)	Size (PE32/ PE32+)	Field	Description
28/24	4/8	ImageBase	The preferred address of the first byte of image when loaded into memory; must be a multiple of 64 K. The default for DLLs is 0x10000000. The default for Windows CE EXEs is 0x00010000. The default for Windows NT, Windows 2000, Windows XP, Windows 95, Windows 98, and Windows Me is 0x00400000.
32/32	4	SectionAlignment	The alignment (in bytes) of sections when they are loaded into memory. It must be greater than or equal to FileAlignment. The default is the page size for the architecture.

Offset (PE32/ PE32+)	Size (PE32/ PE32+)	Field	Description
36/36	4	FileAlignment	The alignment factor (in bytes) that is used to align the raw data of sections in the image file. The value should be a power of 2 between 512 and 64 K, inclusive. The default is 512. If the SectionAlignment is less than the architecture's page size, then FileAlignment must match SectionAlignment.
40/40	2	MajorOperatingSystemVersion	The major version number of the required operating system.
42/42	2	MinorOperatingSystemVersion	The minor version number of the required operating system.
44/44	2	MajorImageVersion	The major version number of the image.
46/46	2	MinorImageVersion	The minor version number of the image.
48/48	2	MajorSubsystemVersion	The major version number of the subsystem.
50/50	2	MinorSubsystemVersion	The minor version number of the subsystem.
52/52	4	Win32VersionValue	Reserved, must be zero.
56/56	4	SizeOfImage	The size (in bytes) of the image, including all headers, as the image is loaded in memory. It must be a multiple of SectionAlignment.
60/60	4	SizeOfHeaders	The combined size of an MS-DOS stub, PE header, and section headers rounded up to a multiple of FileAlignment.
64/64	4	CheckSum	The image file checksum. The algorithm for computing the checksum 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 is loaded into a critical Windows process.
68/68	2	Subsystem	The subsystem that is required to run this image. For more information, see "Windows Subsystem" later in this specification.
70/70	2	DllCharacteristics	For more information, see "DLL Characteristics" later in this specification.
72/72	4/8	SizeOfStackReserve	The size of the stack to reserve. Only SizeOfStackCommit is committed; the rest is made available one page at a time until the reserve size is reached.
76/80	4/8	SizeOfStackCommit	The size of the stack to commit.

Offset (PE32/ PE32+)	Size (PE32/ PE32+)	Field	Description
80/88	4/8	SizeOfHeapReserve	The size of the local heap space to reserve. Only SizeOfHeapCommit is committed; the rest is made available one page at a time until the reserve size is reached.
84/96	4/8	SizeOfHeapCommit	The size of the local heap space to commit.
88/104	4	LoaderFlags	Reserved, must be zero.
92/108	4	NumberOfRvaAndSizes	The number of data-directory entries in the remainder of the optional header. Each describes a location and size.

Windows Subsystem

The following values defined for the Subsystem field of the optional header determine which Windows subsystem (if any) is required to run the image.

Constant	Value	Description
IMAGE_SUBSYSTEM_UNKNOWN	0	An unknown subsystem
IMAGE_SUBSYSTEM_NATIVE	1	Device drivers and native Windows processes
IMAGE_SUBSYSTEM_WINDOWS_GUI	2	The Windows graphical user interface (GUI) subsystem
IMAGE_SUBSYSTEM_WINDOWS_CUI	3	The Windows character subsystem
IMAGE_SUBSYSTEM_OS2_CUI	5	The OS/2 character subsystem
IMAGE_SUBSYSTEM_POSIX_CUI	7	The Posix character subsystem
IMAGE_SUBSYSTEM_NATIVE_WINDOWS	8	Native Win9x driver
IMAGE_SUBSYSTEM_WINDOWS_CE_GUI	9	Windows CE
IMAGE_SUBSYSTEM_EFI_APPLICATION	10	An Extensible Firmware Interface (EFI) application
IMAGE_SUBSYSTEM_EFI_BOOT_ SERVICE_DRIVER	11	An EFI driver with boot services
IMAGE_SUBSYSTEM_EFI_RUNTIME_ DRIVER	12	An EFI driver with run-time services
IMAGE_SUBSYSTEM_EFI_ROM	13	An EFI ROM image
IMAGE_SUBSYSTEM_XBOX	14	XBOX
IMAGE_SUBSYSTEM_WINDOWS_BOOT_APPLICATION	16	Windows boot application.

DLL Characteristics

The following values are defined for the DllCharacteristics field of the optional header.

Constant	Value	Description
	0x0001	Reserved, must be zero.

Constant	Value	Description
	0x0002	Reserved, must be zero.
	0x0004	Reserved, must be zero.
	0x0008	Reserved, must be zero.
IMAGE_DLLCHARACTERISTICS_HIGH_ENTROPY_VA	0x0020	Image can handle a high entropy 64-bit virtual address space.
IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE	0x0040	DLL can be relocated at load time.
IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY	0x0080	Code Integrity checks are enforced.
IMAGE_DLLCHARACTERISTICS_NX_COMPAT	0x0100	Image is NX compatible.
IMAGE_DLLCHARACTERISTICS_NO_ISOLATION	0x0200	Isolation aware, but do not isolate the image.
IMAGE_DLLCHARACTERISTICS_ NO_SEH	0x0400	Does not use structured exception (SE) handling. No SE handler may be called in this image.
IMAGE_DLLCHARACTERISTICS_ NO_BIND	0x0800	Do not bind the image.
IMAGE_DLLCHARACTERISTICS_APPCONTAINER	0x1000	Image must execute in an AppContainer.
IMAGE_DLLCHARACTERISTICS_WDM_DRIVER	0x2000	A WDM driver.
IMAGE_DLLCHARACTERISTICS_GUARD_CF	0x4000	Image supports Control Flow Guard.
IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE	0x8000	Terminal Server aware.

3.4.3. Optional Header Data Directories (Image Only)

Each data directory gives the address and size of a table or string that Windows uses. These data directory entries are all loaded into memory so that the system can use them at run time. A data directory is an 8-byte field that has the following declaration:

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

The first field, VirtualAddress, is actually the RVA of the table. The RVA is the address of the table relative to the base address of the image when the table is loaded. The second field gives the size in bytes. The data directories, which form the last part of the optional header, are listed in the following table.

Note that the number of directories is not fixed. Before looking for a specific directory, check the NumberOfRvaAndSizes field in the optional header.

Also, do not assume that the RVAs in this table point to the beginning of a section or that the sections that contain specific tables have specific names.

Offset (PE/PE32+)	Size	Field	Description
96/112	8	Export Table	The export table address and size. For more information see section 6.3, “The .edata Section (Image Only).”
104/120	8	Import Table	The import table address and size. For more information, see section 6.4, “The .idata Section.”
112/128	8	Resource Table	The resource table address and size. For more information, see section 6.9, “The .rsrc Section.”
120/136	8	Exception Table	The exception table address and size. For more information, see section 6.5, “The .pdata Section.”
128/144	8	Certificate Table	The attribute certificate table address and size. For more information, see section 5.7, “The Attribute Certificate Table (Image Only).”
136/152	8	Base Relocation Table	The base relocation table address and size. For more information, see section 6.6, "The **.reloc** Section (Image Only)."
144/160	8	Debug	The debug data starting address and size. For more information, see section 6.1, “The .debug Section.”
152/168	8	Architecture	Reserved, must be 0
160/176	8	Global Ptr	The RVA of the value to be stored in the global pointer register. The size member of this structure must be set to zero.
168/184	8	TLS Table	The thread local storage (TLS) table address and size. For more information, see section 6.7, “The .tls Section.”
176/192	8	Load Config Table	The load configuration table address and size. For more information, see section 6.8, “The Load Configuration Structure (Image Only).”
184/200	8	Bound Import	The bound import table address and size.
192/208	8	IAT	The import address table address and size. For more information, see section 6.4.4, “Import Address Table.”
200/216	8	Delay Import Descriptor	The delay import descriptor address and size. For more information, see section 5.8, “Delay-Load Import Tables (Image Only).”
208/224	8	CLR Runtime Header	The CLR runtime header address and size. For more information, see section 6.10, “The .cormeta Section (Object Only).”
216/232	8	Reserved, must be zero	

The Certificate Table entry points to a table of attribute certificates. These certificates are not loaded into memory as part of the image. As such, the first field of this entry, which is normally an RVA, is a file pointer instead.

4. Section Table (Section Headers)

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

headers. Make sure to use the size of the optional header as specified in the file header.

The number of entries in the section table is given by the NumberOfSections field in the file header. Entries in the section table are numbered starting from one (1). The code and data memory section entries are in the order chosen by the linker.

In an image file, the VAs for sections must be assigned by the linker so that they are in ascending order and adjacent, and they must be a multiple of the SectionAlignment 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	Name	An 8-byte, null-padded UTF-8 encoded string. If the string is exactly 8 characters long, there is no terminating null. For longer names, this field contains a slash (/) that is followed by an ASCII representation of a decimal number that is an offset into the string table. Executable images do not use a string table and do not support section names longer than 8 characters. Long names in object files are truncated if they are emitted to an executable file.
8	4	VirtualSize	The total size of the section when loaded into memory. If this value is greater than SizeOfRawData, the section is zero-padded. This field is valid only for executable images and should be set to zero for object files.
12	4	VirtualAddress	For executable images, the address of the first byte of the section relative to the image base when the section is loaded into memory. For object files, this field is the address of the first byte before relocation is applied; for simplicity, compilers should set this to zero. Otherwise, it is an arbitrary value that is subtracted from offsets during relocation.
16	4	SizeOfRawData	The size of the section (for object files) or the size of the initialized data on disk (for image files). For executable images, this must be a multiple of FileAlignment from the optional header. If this is less than VirtualSize, the remainder of the section is zero-filled. Because the SizeOfRawData field is rounded but the VirtualSize field is not, it is possible for SizeOfRawData to be greater than VirtualSize as well. When a section contains only uninitialized data, this field should be zero.
20	4	PointerToRawData	The file pointer to the first page of the section within the COFF file. For executable images, this must be a multiple of FileAlignment from the optional header. For object files, the value should be aligned on a 4-byte boundary for best performance. When a section contains only uninitialized data, this field should be zero.
24	4	PointerToRelocations	The file pointer to the beginning of relocation entries for the section. This is set to zero for executable images or if there are no relocations.

Offset	Size	Field	Description
28	4	PointerToLinenumbers	The file pointer to the beginning of line-number entries for the section. This is set to zero if there are no COFF line numbers. This value should be zero for an image because COFF debugging information is deprecated.
32	2	NumberOfRelocations	The number of relocation entries for the section. This is set to zero for executable images.
34	2	NumberOfLinenumbers	The number of line-number entries for the section. This value should be zero for an image because COFF debugging information is deprecated.
36	4	Characteristics	The flags that describe the characteristics of the section. For more information, see section 4.1, “Section Flags.”

4.1. Section Flags

The section flags in the Characteristics field of the section header indicate characteristics of the section.

Flag	Value	Description
	0x00000000	Reserved for future use.
	0x00000001	Reserved for future use.
	0x00000002	Reserved for future use.
	0x00000004	Reserved for future use.
IMAGE_SCN_TYPE_NO_PAD	0x00000008	The section should not be padded to the next boundary. This flag is obsolete and is replaced by IMAGE_SCN_ALIGN_1BYTES. This is valid only for object files.
	0x00000010	Reserved for future use.
IMAGE_SCN_CNT_CODE	0x00000020	The section contains executable code.
IMAGE_SCN_CNT_INITIALIZED_DATA	0x00000040	The section contains initialized data.
IMAGE_SCN_CNT_UNINITIALIZED_ DATA	0x00000080	The section contains uninitialized data.
IMAGE_SCN_LNK_OTHER	0x00000100	Reserved for future use.
IMAGE_SCN_LNK_INFO	0x00000200	The section contains comments or other information. The **.drectve** section has this type. This is valid for object files only.
	0x00000400	Reserved for future use.
IMAGE_SCN_LNK_REMOVE	0x00000800	The section will not become part of the image. This is valid only for object files.
IMAGE_SCN_LNK_COMDAT	0x00001000	The section contains COMDAT data. For more information, see section 5.5.6, “COMDAT Sections (Object Only).” This is valid only for object files.
IMAGE_SCN_GPREL	0x00008000	The section contains data referenced through the global pointer (GP).
IMAGE_SCN_MEM_PURGEABLE	0x00020000	Reserved for future use.
IMAGE_SCN_MEM_16BIT	0x00020000	Reserved for future use.

Flag	Value	Description
IMAGE_SCN_MEM_LOCKED	0x00040000	Reserved for future use.
IMAGE_SCN_MEM_PRELOAD	0x00080000	Reserved for future use.
IMAGE_SCN_ALIGN_1BYTES	0x00100000	Align data on a 1-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_2BYTES	0x00200000	Align data on a 2-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_4BYTES	0x00300000	Align data on a 4-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_8BYTES	0x00400000	Align data on an 8-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_16BYTES	0x00500000	Align data on a 16-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_32BYTES	0x00600000	Align data on a 32-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_64BYTES	0x00700000	Align data on a 64-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_128BYTES	0x00800000	Align data on a 128-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_256BYTES	0x00900000	Align data on a 256-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_512BYTES	0x00A00000	Align data on a 512-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_1024BYTES	0x00B00000	Align data on a 1024-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_2048BYTES	0x00C00000	Align data on a 2048-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_4096BYTES	0x00D00000	Align data on a 4096-byte boundary. Valid only for object files.
IMAGE_SCN_ALIGN_8192BYTES	0x00E00000	Align data on an 8192-byte boundary. Valid only for object files.
IMAGE_SCN_LNK_NRELOC_OVFL	0x01000000	The section contains extended relocations.
IMAGE_SCN_MEM_DISCARDABLE	0x02000000	The section can be discarded as needed.
IMAGE_SCN_MEM_NOT_CACHED	0x04000000	The section cannot be cached.
IMAGE_SCN_MEM_NOT_PAGED	0x08000000	The section is not pageable.
IMAGE_SCN_MEM_SHARED	0x10000000	The section can be shared in memory.
IMAGE_SCN_MEM_EXECUTE	0x20000000	The section can be executed as code.
IMAGE_SCN_MEM_READ	0x40000000	The section can be read.
IMAGE_SCN_MEM_WRITE	0x80000000	The section can be written to.

IMAGE_SCN_LNK_NRELOC_OVFL indicates that the count of relocations for the section exceeds the 16 bits that are reserved for it in the section header. If the bit is set and the NumberOfRelocations field in the section header is 0xffff, the actual relocation count is stored in the 32-bit VirtualAddress field of the first relocation. It is an error if IMAGE_SCN_LNK_NRELOC_OVFL is set and there are fewer than 0xffff relocations in the section.

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 "$" and all characters that follow it. Thus, an object section named **.text$X** actually contributes to the **.text** section in the image.

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

The section name in an image file never contains a "$" character.

5. Other Contents of the File

The data structures that were described so far, up to and including the optional header, are all located at a fixed offset from the beginning of the file (or from the PE header if the file is an image that contains 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 SectionAlignment value of less than the page size of the architecture (4 K for Intel x86 and for MIPS, and 8 K for Itanium). For a description of SectionAlignment, see section 3.4, "Optional Header (Image Only)." In this case, there are constraints on the file offset of the section data, as described in section 5.1, "Section Data." Another exception is that attribute certificate and debug information must be placed at the very end of an image file, with the attribute certificate table immediately preceding the debug section, because the loader does not map these into memory. The rule about attribute certificate and debug information does not apply to object files, however.

5.1. Section Data

Initialized data for a section consists of simple blocks of bytes. However, for sections that contain all zeros, the section data need not be included.

The data for each section is located at the file offset that was given by the PointerToRawData field in the section header. The size of this data in the file is indicated by the SizeOfRawData field. If SizeOfRawData is less than VirtualSize, the remainder is padded with zeros.

In an image file, the section data must be aligned on a boundary as specified by the FileAlignment field in the 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 if the SectionAlignment value in the optional header is less than the page size of the architecture. 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 Only)

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.

Image files do not contain COFF relocations, because all referenced symbols have already been assigned addresses in a flat address space. An image contains relocation information in the form of base relocations in the **.reloc** section (unless the image has the IMAGE_FILE_RELOCS_STRIPPED attribute). For more information, see section 6.6, "The **.reloc** Section (Image Only)."

For each section in an object file, an array of fixed-length records holds 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	VirtualAddress	The 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 (Section Headers)." For example, if the first byte of the section has an address of 0x10, the third byte has an address of 0x12.
4	4	SymbolTableIndex	A zero-based index into the symbol table. This symbol gives the address that is 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 that indicates the kind of relocation that should be performed. Valid relocation types depend on machine type. See section 5.2.1, "Type Indicators."

If the symbol referred to by the SymbolTableIndex field has the storage class IMAGE_SYM_CLASS_SECTION, the symbol's address is the beginning of the section. The section is usually in the same file, except when the object file is part of an archive (library). In that case, the section can 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, that is, 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.

x64 Processors

The following relocation type indicators are defined for x64 and compatible processors.

Constant	Value	Description
IMAGE_REL_AMD64_ABSOLUTE	0x0000	The relocation is ignored.
IMAGE_REL_AMD64_ADDR64	0x0001	The 64-bit VA of the relocation target.
IMAGE_REL_AMD64_ADDR32	0x0002	The 32-bit VA of the relocation target.
IMAGE_REL_AMD64_ADDR32NB	0x0003	The 32-bit address without an image base (RVA).
IMAGE_REL_AMD64_REL32	0x0004	The 32-bit relative address from the byte following the relocation.
IMAGE_REL_AMD64_REL32_1	0x0005	The 32-bit address relative to byte distance 1 from the relocation.

Constant	Value	Description
IMAGE_REL_AMD64_REL32_2	0x0006	The 32-bit address relative to byte distance 2 from the relocation.
IMAGE_REL_AMD64_REL32_3	0x0007	The 32-bit address relative to byte distance 3 from the relocation.
IMAGE_REL_AMD64_REL32_4	0x0008	The 32-bit address relative to byte distance 4 from the relocation.
IMAGE_REL_AMD64_REL32_5	0x0009	The 32-bit address relative to byte distance 5 from the relocation.
IMAGE_REL_AMD64_SECTION	0x000A	The 16-bit section index of the section that contains the target. This is used to support debugging information.
IMAGE_REL_AMD64_SECREL	0x000B	The 32-bit offset of the target from the beginning of its section. This is used to support debugging information and static thread local storage.
IMAGE_REL_AMD64_SECREL7	0x000C	A 7-bit unsigned offset from the base of the section that contains the target.
IMAGE_REL_AMD64_TOKEN	0x000D	CLR tokens.
IMAGE_REL_AMD64_SREL32	0x000E	A 32-bit signed span-dependent value emitted into the object.
IMAGE_REL_AMD64_PAIR	0x000F	A pair that must immediately follow every span-dependent value.
IMAGE_REL_AMD64_SSPAN32	0x0010	A 32-bit signed span-dependent value that is applied at link time.

ARM Processors

The following relocation type indicators are defined for ARM processors.

Constant	Value	Description
IMAGE_REL_ARM_ABSOLUTE	0x0000	The relocation is ignored.
IMAGE_REL_ARM_ADDR32	0x0001	The 32-bit VA of the target.
IMAGE_REL_ARM_ADDR32NB	0x0002	The 32-bit RVA of the target.
IMAGE_REL_ARM_BRANCH24	0x0003	The 24-bit relative displacement to the target.
IMAGE_REL_ARM_BRANCH11	0x0004	The reference to a subroutine call. The reference consists of two 16-bit instructions with 11-bit offsets.
IMAGE_REL_ARM_SECTION	0x000E	The 16-bit section index of the section that contains the target. This is used to support debugging information.
IMAGE_REL_ARM_SECREL	0x000F	The 32-bit offset of the target from the beginning of its section. This is used to support debugging information and static thread local storage.
IMAGE_REL_ARM_MOV32	0x0010	The 32-bit VA of the target. This relocation is applied using a MOVW instruction for the low 16 bits followed by a MOVT for the high 16 bits.
IMAGE_REL_THUMB_MOV32	0x0011	The 32-bit VA of the target. This relocation is applied using a MOVW instruction for the low 16 bits followed by a MOVT for the high 16 bits.

Constant	Value	Description
IMAGE_REL_THUMB_BRANCH20	0x0012	The instruction is fixed up with the 21-bit relative displacement to the 2-byte aligned target. The least significant bit of the displacement is always zero and is not stored. This relocation corresponds to a Thumb-2 32-bit conditional B instruction.
Unused	0x0013	
IMAGE_REL_THUMB_BRANCH24	0x0014	The instruction is fixed up with the 25-bit relative displacement to the 2-byte aligned target. The least significant bit of the displacement is zero and is not stored. This relocation corresponds to a Thumb-2 B instruction.
IMAGE_REL_THUMB_BLX23	0x0015	The instruction is fixed up with the 25-bit relative displacement to the 4-byte aligned target. The low 2 bits of the displacement are zero and are not stored. This relocation corresponds to a Thumb-2 BLX instruction.
IMAGE_REL_ARM_PAIR	0x0016	The relocation is valid only when it immediately follows a ARM_REFHI or THUMB_REFHI. Its SymbolTableIndex contains a displacement and not an index into the symbol table.

ARM64 Processors

The following relocation type indicators are defined for ARM64 processors.

Constant	Value	Description
IMAGE_REL_ARM64_ABSOLUTE	0x0000	The relocation is ignored.
IMAGE_REL_ARM64_ADDR32	0x0001	The 32-bit VA of the target.
IMAGE_REL_ARM64_ADDR32NB	0x0002	The 32-bit RVA of the target.
IMAGE_REL_ARM64_BRANCH26	0x0003	The 26-bit relative displacement to the target, for B and BL instructions.
IMAGE_REL_ARM64_PAGEBASE_REL21	0x0004	The page base of the target, for ADRP instruction.
IMAGE_REL_ARM64_REL21	0x0005	The 12-bit relative displacement to the target, for instruction ADR
IMAGE_REL_ARM64_PAGEOFFSET_12A	0x0006	The 12-bit page offset of the target, for instructions ADD/ADDS (immediate) with zero shift.
IMAGE_REL_ARM64_PAGEOFFSET_12L	0x0007	The 12-bit page offset of the target, for instruction LDR (indexed, unsigned immediate).
IMAGE_REL_ARM64_SECREL	0x0008	The 32-bit offset of the target from the beginning of its section. This is used to support debugging information and static thread local storage.
IMAGE_REL_ARM64_SECREL_LOW12A	0x0009	Bit 0:11 of section offset of the target, for instructions ADD/ADDS (immediate) with zero shift.
IMAGE_REL_ARM64_SECREL_HIGH12A	0x000A	Bit 12:23 of section offset of the target, for instructions ADD/ADDS (immediate) with zero shift.

Constant	Value	Description
IMAGE_REL_ARM64_SECREL_LOW12L	0x000B	Bit 0:11 of section offset of the target, for instruction LDR (indexed, unsigned immediate).
IMAGE_REL_ARM64_TOKEN	0x000C	CLR token.
IMAGE_REL_ARM64_SECTION	0x000D	The 16-bit section index of the section that contains the target. This is used to support debugging information.
IMAGE_REL_ARM64_ADDR64	0x000E	The 64-bit VA of the relocation target.
IMAGE_REL_ARM64_BRANCH19	0x000F	The 19-bit offset to the relocation target, for conditional B instruction.
IMAGE_REL_ARM64_BRANCH14	0x0010	The 14-bit offset to the relocation target, for instructions TBZ and TBNZ.

Hitachi SuperH Processors

The following relocation type indicators are defined for SH3 and SH4 processors. SH5-specific relocations are noted as SHM (SH Media).

Constant	Value	Description
IMAGE_REL_SH3_ABSOLUTE	0x0000	The relocation is ignored.
IMAGE_REL_SH3_DIRECT16	0x0001	A reference to the 16-bit location that contains the VA of the target symbol.
IMAGE_REL_SH3_DIRECT32	0x0002	The 32-bit VA of the target symbol.
IMAGE_REL_SH3_DIRECT8	0x0003	A reference to the 8-bit location that contains the VA of the target symbol.
IMAGE_REL_SH3_DIRECT8_WORD	0x0004	A reference to the 8-bit instruction that contains the effective 16-bit VA of the target symbol.
IMAGE_REL_SH3_DIRECT8_LONG	0x0005	A reference to the 8-bit instruction that contains the effective 32-bit VA of the target symbol.
IMAGE_REL_SH3_DIRECT4	0x0006	A reference to the 8-bit location whose low 4 bits contain the VA of the target symbol.
IMAGE_REL_SH3_DIRECT4_WORD	0x0007	A reference to the 8-bit instruction whose low 4 bits contain the effective 16-bit VA of the target symbol.
IMAGE_REL_SH3_DIRECT4_LONG	0x0008	A reference to the 8-bit instruction whose low 4 bits contain the effective 32-bit VA of the target symbol.
IMAGE_REL_SH3_PCREL8_WORD	0x0009	A reference to the 8-bit instruction that contains the effective 16-bit relative offset of the target symbol.
IMAGE_REL_SH3_PCREL8_LONG	0x000A	A reference to the 8-bit instruction that contains the effective 32-bit relative offset of the target symbol.
IMAGE_REL_SH3_PCREL12_WORD	0x000B	A reference to the 16-bit instruction whose low 12 bits contain the effective 16-bit relative offset of the target symbol.
IMAGE_REL_SH3_STARTOF_SECTION	0x000C	A reference to a 32-bit location that is the VA of the section that contains the target symbol.

Constant	Value	Description
IMAGE_REL_SH3_SIZEOF_SECTION	0x000D	A reference to the 32-bit location that is the size of the section that contains the target symbol.
IMAGE_REL_SH3_SECTION	0x000E	The 16-bit section index of the section that contains the target. This is used to support debugging information.
IMAGE_REL_SH3_SECREL	0x000F	The 32-bit offset of the target from the beginning of its section. This is used to support debugging information and static thread local storage.
IMAGE_REL_SH3_DIRECT32_NB	0x0010	The 32-bit RVA of the target symbol.
IMAGE_REL_SH3_GPREL4_LONG	0x0011	GP relative.
IMAGE_REL_SH3_TOKEN	0x0012	CLR token.
IMAGE_REL_SHM_PCRELPT	0x0013	The offset from the current instruction in longwords. If the NOMODE bit is not set, insert the inverse of the low bit at bit 32 to select PTA or PTB.
IMAGE_REL_SHM_REFLO	0x0014	The low 16 bits of the 32-bit address.
IMAGE_REL_SHM_REFHALF	0x0015	The high 16 bits of the 32-bit address.
IMAGE_REL_SHM_RELLO	0x0016	The low 16 bits of the relative address.
IMAGE_REL_SHM_RELHALF	0x0017	The high 16 bits of the relative address.
IMAGE_REL_SHM_PAIR	0x0018	The relocation is valid only when it immediately follows a REFHALF, RELHALF, or RELLO relocation. The SymbolTableIndex field of the relocation contains a displacement and not an index into the symbol table.
IMAGE_REL_SHM_NOMODE	0x8000	The relocation ignores section mode.

IBM PowerPC Processors

The following relocation type indicators are defined for PowerPC processors.

Constant	Value	Description
IMAGE_REL_PPC_ABSOLUTE	0x0000	The relocation is ignored.
IMAGE_REL_PPC_ADDR64	0x0001	The 64-bit VA of the target.
IMAGE_REL_PPC_ADDR32	0x0002	The 32-bit VA of the target.
IMAGE_REL_PPC_ADDR24	0x0003	The low 24 bits of the VA of the target. This is valid only when the target symbol is absolute and can be sign-extended to its original value.
IMAGE_REL_PPC_ADDR16	0x0004	The low 16 bits of the target's VA.
IMAGE_REL_PPC_ADDR14	0x0005	The low 14 bits of the target's VA. This is valid only when the target symbol is absolute and can be sign-extended to its original value.
IMAGE_REL_PPC_REL24	0x0006	A 24-bit PC-relative offset to the symbol's location.
IMAGE_REL_PPC_REL14	0x0007	A 14-bit PC-relative offset to the symbol's location.
IMAGE_REL_PPC_ADDR32NB	0x000A	The 32-bit RVA of the target.
IMAGE_REL_PPC_SECREL	0x000B	The 32-bit offset of the target from the beginning of its section. This is used to support debugging information and static thread local storage.

Constant	Value	Description
IMAGE_REL_PPC_SECTION	0x000C	The 16-bit section index of the section that contains the target. This is used to support debugging information.
IMAGE_REL_PPC_SECREL16	0x000F	The 16-bit offset of the target from the beginning of its section. This is used to support debugging information and static thread local storage.
IMAGE_REL_PPC_REFHI	0x0010	The high 16 bits of the target's 32-bit VA. This is used for the first instruction in a two-instruction sequence that loads a full address. This relocation must be immediately followed by a PAIR relocation whose SymbolTableIndex contains a signed 16-bit displacement that is added to the upper 16 bits that was taken from the location that is being relocated.
IMAGE_REL_PPC_REFLO	0x0011	The low 16 bits of the target's VA.
IMAGE_REL_PPC_PAIR	0x0012	A relocation that is valid only when it immediately follows a REFHI or SECRELHI relocation. Its SymbolTableIndex contains a displacement and not an index into the symbol table.
IMAGE_REL_PPC_SECRELLO	0x0013	The low 16 bits of the 32-bit offset of the target from the beginning of its section.
IMAGE_REL_PPC_GPREL	0x0015	The 16-bit signed displacement of the target relative to the GP register.
IMAGE_REL_PPC_TOKEN	0x0016	The CLR token.

Intel 386 Processors

The following relocation type indicators are defined for Intel 386 and compatible processors.

Constant	Value	Description
IMAGE_REL_I386_ABSOLUTE	0x0000	The relocation is ignored.
IMAGE_REL_I386_DIR16	0x0001	Not supported.
IMAGE_REL_I386_REL16	0x0002	Not supported.
IMAGE_REL_I386_DIR32	0x0006	The target's 32-bit VA.
IMAGE_REL_I386_DIR32NB	0x0007	The target's 32-bit RVA.
IMAGE_REL_I386_SEG12	0x0009	Not supported.
IMAGE_REL_I386_SECTION	0x000A	The 16-bit section index of the section that contains the target. This is used to support debugging information.
IMAGE_REL_I386_SECREL	0x000B	The 32-bit offset of the target from the beginning of its section. This is used to support debugging information and static thread local storage.
IMAGE_REL_I386_TOKEN	0x000C	The CLR token.
IMAGE_REL_I386_SECREL7	0x000D	A 7-bit offset from the base of the section that contains the target.
IMAGE_REL_I386_REL32	0x0014	The 32-bit relative displacement to the target. This supports the x86 relative branch and call instructions.

Intel Itanium Processor Family (IPF)

The following relocation type indicators are defined for the Intel Itanium processor family and compatible processors. Note that relocations on *instructions* use the bundle's offset and slot number for the relocation offset.

Constant	Value	Description
IMAGE_REL_IA64_ABSOLUTE	0x0000	The relocation is ignored.
IMAGE_REL_IA64_IMM14	0x0001	The instruction relocation can be followed by an ADDEND relocation whose value is added to the target address before it is inserted into the specified slot in the IMM14 bundle. The relocation target must be absolute or the image must be fixed.
IMAGE_REL_IA64_IMM22	0x0002	The instruction relocation can be followed by an ADDEND relocation whose value is added to the target address before it is inserted into the specified slot in the IMM22 bundle. The relocation target must be absolute or the image must be fixed.
IMAGE_REL_IA64_IMM64	0x0003	The slot number of this relocation must be one (1). The relocation can be followed by an ADDEND relocation whose value is added to the target address before it is stored in all three slots of the IMM64 bundle.
IMAGE_REL_IA64_DIR32	0x0004	The target's 32-bit VA. This is supported only for /LARGEADDRESSAWARE:NO images.
IMAGE_REL_IA64_DIR64	0x0005	The target's 64-bit VA.
IMAGE_REL_IA64_PCREL21B	0x0006	The instruction is fixed up with the 25-bit relative displacement to the 16-bit aligned target. The low 4 bits of the displacement are zero and are not stored.
IMAGE_REL_IA64_PCREL21M	0x0007	The instruction is fixed up with the 25-bit relative displacement to the 16-bit aligned target. The low 4 bits of the displacement, which are zero, are not stored.
IMAGE_REL_IA64_PCREL21F	0x0008	The LSBs of this relocation's offset must contain the slot number whereas the rest is the bundle address. The bundle is fixed up with the 25-bit relative displacement to the 16-bit aligned target. The low 4 bits of the displacement are zero and are not stored.
IMAGE_REL_IA64_GPREL22	0x0009	The instruction relocation can be followed by an ADDEND relocation whose value is added to the target address and then a 22-bit GP-relative offset that is calculated and applied to the GPREL22 bundle.
IMAGE_REL_IA64_LTOFF22	0x000A	The instruction is fixed up with the 22-bit GP-relative offset to the target symbol's literal table entry. The linker creates this literal table entry based on this relocation and the ADDEND relocation that might follow.
IMAGE_REL_IA64_SECTION	0x000B	The 16-bit section index of the section contains the target. This is used to support debugging information.

Constant	Value	Description
IMAGE_REL_IA64_SECREL22	0x000C	The instruction is fixed up with the 22-bit offset of the target from the beginning of its section. This relocation can be followed immediately by an ADDEND relocation, whose Value field contains the 32-bit unsigned offset of the target from the beginning of the section.
IMAGE_REL_IA64_SECREL64I	0x000D	The slot number for this relocation must be one (1). The instruction is fixed up with the 64-bit offset of the target from the beginning of its section. This relocation can be followed immediately by an ADDEND relocation whose Value field contains the 32-bit unsigned offset of the target from the beginning of the section.
IMAGE_REL_IA64_SECREL32	0x000E	The address of data to be fixed up with the 32-bit offset of the target from the beginning of its section.
IMAGE_REL_IA64_DIR32NB	0x0010	The target's 32-bit RVA.
IMAGE_REL_IA64_SREL14	0x0011	This is applied to a signed 14-bit immediate that contains the difference between two relocatable targets. This is a declarative field for the linker that indicates that the compiler has already emitted this value.
IMAGE_REL_IA64_SREL22	0x0012	This is applied to a signed 22-bit immediate that contains the difference between two relocatable targets. This is a declarative field for the linker that indicates that the compiler has already emitted this value.
IMAGE_REL_IA64_SREL32	0x0013	This is applied to a signed 32-bit immediate that contains the difference between two relocatable values. This is a declarative field for the linker that indicates that the compiler has already emitted this value.
IMAGE_REL_IA64_UREL32	0x0014	This is applied to an unsigned 32-bit immediate that contains the difference between two relocatable values. This is a declarative field for the linker that indicates that the compiler has already emitted this value.
IMAGE_REL_IA64_PCREL60X	0x0015	A 60-bit PC-relative fixup that always stays as a BRL instruction of an MLX bundle.
IMAGE_REL_IA64_PCREL60B	0x0016	A 60-bit PC-relative fixup. If the target displacement fits in a signed 25-bit field, convert the entire bundle to an MBB bundle with NOP.B in slot 1 and a 25-bit BR instruction (with the 4 lowest bits all zero and dropped) in slot 2.
IMAGE_REL_IA64_PCREL60F	0x0017	A 60-bit PC-relative fixup. If the target displacement fits in a signed 25-bit field, convert the entire bundle to an MFB bundle with NOP.F in slot 1 and a 25-bit (4 lowest bits all zero and dropped) BR instruction in slot 2.

Constant	Value	Description
IMAGE_REL_IA64_PCREL60I	0x0018	A 60-bit PC-relative fixup. If the target displacement fits in a signed 25-bit field, convert the entire bundle to an MIB bundle with NOP.I in slot 1 and a 25-bit (4 lowest bits all zero and dropped) BR instruction in slot 2.
IMAGE_REL_IA64_PCREL60M	0x0019	A 60-bit PC-relative fixup. If the target displacement fits in a signed 25-bit field, convert the entire bundle to an MMB bundle with NOP.M in slot 1 and a 25-bit (4 lowest bits all zero and dropped) BR instruction in slot 2.
IMAGE_REL_IA64_IMMGPREL64	0x001a	A 64-bit GP-relative fixup.
IMAGE_REL_IA64_TOKEN	0x001b	A CLR token.
IMAGE_REL_IA64_GPREL32	0x001c	A 32-bit GP-relative fixup.
IMAGE_REL_IA64_ADDEND	0x001F	The relocation is valid only when it immediately follows one of the following relocations: IMM14, IMM22, IMM64, GPREL22, LTOFF22, LTOFF64, SECREL22, SECREL64I, or SECREL32. Its value contains the addend to apply to instructions within a bundle, not for data.

MIPS Processors

The following relocation type indicators are defined for MIPS processors.

Constant	Value	Description
IMAGE_REL_MIPS_ABSOLUTE	0x0000	The relocation is ignored.
IMAGE_REL_MIPS_REFHALF	0x0001	The high 16 bits of the target's 32-bit VA.
IMAGE_REL_MIPS_REFWORD	0x0002	The target's 32-bit VA.
IMAGE_REL_MIPS_JMPADDR	0x0003	The low 26 bits of the target's VA. This supports the MIPS J and JAL instructions.
IMAGE_REL_MIPS_REFHI	0x0004	The high 16 bits of the target's 32-bit VA. This is used for the first instruction in a two-instruction sequence that loads a full address. This relocation must be immediately followed by a PAIR relocation whose SymbolTableIndex contains a signed 16-bit displacement that is added to the upper 16 bits that are taken from the location that is being relocated.
IMAGE_REL_MIPS_REFLO	0x0005	The low 16 bits of the target's VA.
IMAGE_REL_MIPS_GPREL	0x0006	A 16-bit signed displacement of the target relative to the GP register.
IMAGE_REL_MIPS_LITERAL	0x0007	The same as IMAGE_REL_MIPS_GPREL.
IMAGE_REL_MIPS_SECTION	0x000A	The 16-bit section index of the section contains the target. This is used to support debugging information.
IMAGE_REL_MIPS_SECREL	0x000B	The 32-bit offset of the target from the beginning of its section. This is used to support debugging information and static thread local storage.
IMAGE_REL_MIPS_SECRELLO	0x000C	The low 16 bits of the 32-bit offset of the target from the beginning of its section.

Constant	Value	Description
IMAGE_REL_MIPS_SECRELHI	0x000D	The high 16 bits of the 32-bit offset of the target from the beginning of its section. An IMAGE_REL_MIPS_PAIR relocation must immediately follow this one. The SymbolTableIndex of the PAIR relocation contains a signed 16-bit displacement that is added to the upper 16 bits that are taken from the location that is being relocated.
IMAGE_REL_MIPS_JMPADDR16	0x0010	The low 26 bits of the target's VA. This supports the MIPS16 JAL instruction.
IMAGE_REL_MIPS_REFWORDNB	0x0022	The target's 32-bit RVA.
IMAGE_REL_MIPS_PAIR	0x0025	The relocation is valid only when it immediately follows a REFHI or SECRELHI relocation. Its SymbolTableIndex contains a displacement and not an index into the symbol table.

Mitsubishi M32R

The following relocation type indicators are defined for the Mitsubishi M32R processors.

Constant	Value	Description
IMAGE_REL_M32R_ABSOLUTE	0x0000	The relocation is ignored.
IMAGE_REL_M32R_ADDR32	0x0001	The target's 32-bit VA.
IMAGE_REL_M32R_ADDR32NB	0x0002	The target's 32-bit RVA.
IMAGE_REL_M32R_ADDR24	0x0003	The target's 24-bit VA.
IMAGE_REL_M32R_GPREL16	0x0004	The target's 16-bit offset from the GP register.
IMAGE_REL_M32R_PCREL24	0x0005	The target's 24-bit offset from the program counter (PC), shifted left by 2 bits and sign-extended
IMAGE_REL_M32R_PCREL16	0x0006	The target's 16-bit offset from the PC, shifted left by 2 bits and sign-extended
IMAGE_REL_M32R_PCREL8	0x0007	The target's 8-bit offset from the PC, shifted left by 2 bits and sign-extended
IMAGE_REL_M32R_REFHALF	0x0008	The 16 MSBs of the target VA.
IMAGE_REL_M32R_REFHI	0x0009	The 16 MSBs of the target VA, adjusted for LSB sign extension. This is used for the first instruction in a two-instruction sequence that loads a full 32-bit address. This relocation must be immediately followed by a PAIR relocation whose SymbolTableIndex contains a signed 16-bit displacement that is added to the upper 16 bits that are taken from the location that is being relocated.
IMAGE_REL_M32R_REFLO	0x000A	The 16 LSBs of the target VA.
IMAGE_REL_M32R_PAIR	0x000B	The relocation must follow the REFHI relocation. Its SymbolTableIndex contains a displacement and not an index into the symbol table.
IMAGE_REL_M32R_SECTION	0x000C	The 16-bit section index of the section that contains the target. This is used to support debugging information.
IMAGE_REL_M32R_SECREL	0x000D	The 32-bit offset of the target from the beginning of its section. This is used to support debugging information and static thread local storage.
IMAGE_REL_M32R_TOKEN	0x000E	The CLR token.

5.3. COFF Line Numbers (Deprecated)

COFF line numbers are no longer produced and, in the future, will not be consumed.

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 (*)	This is a union of two fields: SymbolTableIndex and VirtualAddress. Whether SymbolTableIndex 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 4-byte fields: SymbolTableIndex and VirtualAddress.

Offset	Size	Field	Description
0	4	SymbolTableIndex	Used when Linenumber is zero: index to symbol table entry for a function. This format is used to indicate the function to which a group of line-number records refers.
0	4	VirtualAddress	Used when Linenumber is non-zero: the RVA of the executable code that corresponds to the source line indicated. In an object file, this contains the VA within the section.

A line-number record can either set the Linenumber field to zero and point to a function definition in the symbol table or 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 (Object Only)." The function's auxiliary record in the symbol table has a pointer to the Linenumber field that points to this same line-number record.

A record that identifies a function is followed by any number of line-number entries that give actual line-number information (that is, entries with 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 line.

For example, the first line-number record for the following example would specify the ReverseSign function (SymbolTableIndex of ReverseSign and Linenumber set to zero). 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 in this section is inherited from the traditional COFF format. It is distinct from Microsoft Visual C++® debug information. A file can contain both a COFF symbol table and Visual C++ 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 (*)	The name of the symbol, represented by a union of three structures. An array of 8 bytes is used if the name is not more than 8 bytes long. For more information, see section 5.4.1, “Symbol Name Representation.”
8	4	Value	The value that is associated with the symbol. The interpretation of this field depends on SectionNumber and StorageClass. A typical meaning is the relocatable address.
12	2	SectionNumber	The signed integer that identifies the section, using a one-based index into the section table. Some values have special meaning, as defined in section 5.4.2, “Section Number Values.”
14	2	Type	A number that represents type. Microsoft tools set this field to 0x20 (function) or 0x0 (not a function). For more information, see section 5.4.3, “Type Representation.”
16	1	StorageClass	An enumerated value that represents storage class. For more information, see section 5.4.4, “Storage Class.”
17	1	NumberOfAuxSymbols	The 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 StorageClass field. Currently-defined formats for auxiliary symbol table records are shown in section 5.5, “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 ShortName field in a symbol table consists of 8 bytes that contain the name itself, if it is not more than 8 bytes long, or the ShortName field gives an offset into

the string table. To determine whether the name itself or an offset is given, test the first 4 bytes for equality to zero.

By convention, the names are treated as zero-terminated UTF-8 encoded strings.

Offset	Size	Field	Description
0	8	ShortName	An array of 8 bytes. This array is padded with nulls on the right if the name is less than 8 bytes long.
0	4	Zeroes	A field that is set to all zeros if the name is longer than 8 bytes.
4	4	Offset	An 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 can take negative values. The following values, less than one, have special meanings.

Constant	Value	Description
IMAGE_SYM_UNDEFINED	0	The symbol record is not yet assigned a section. A value of zero indicates that a reference to an external symbol is defined elsewhere. A value of non-zero is a common symbol with a size that is specified by the value.
IMAGE_SYM_ABSOLUTE	-1	The symbol has an absolute (non-relocatable) value and is not an address.
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 2 bytes, where each byte represents type information. The LSB represents the simple (base) data type, and the MSB represents the complex type, if any:

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

The following values are defined for base type, although Microsoft tools generally do not use this field and set the LSB to 0. Instead, Visual C++ debug 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	A character (signed byte)
IMAGE_SYM_TYPE_SHORT	3	A 2-byte signed integer
IMAGE_SYM_TYPE_INT	4	A natural integer type (normally 4 bytes in Windows)
IMAGE_SYM_TYPE_LONG	5	A 4-byte signed integer
IMAGE_SYM_TYPE_FLOAT	6	A 4-byte floating-point number
IMAGE_SYM_TYPE_DOUBLE	7	An 8-byte floating-point number
IMAGE_SYM_TYPE_STRUCT	8	A structure
IMAGE_SYM_TYPE_UNION	9	A union
IMAGE_SYM_TYPE_ENUM	10	An enumerated type

Constant	Value	Description
IMAGE_SYM_TYPE_MOE	11	A member of enumeration (a specific value)
IMAGE_SYM_TYPE_BYTE	12	A byte; unsigned 1-byte integer
IMAGE_SYM_TYPE_WORD	13	A word; unsigned 2-byte integer
IMAGE_SYM_TYPE_UINT	14	An unsigned integer of natural size (normally, 4 bytes)
IMAGE_SYM_TYPE_DWORD	15	An unsigned 4-byte integer

The most significant byte specifies whether the symbol is a pointer to, function returning, or array of the base type that is specified in the LSB. Microsoft tools use this field only to indicate whether 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.

It is very important to specify the function attribute correctly. This information is required for incremental linking to work correctly. For some architectures, the information may be required for other purposes.

Constant	Value	Description
IMAGE_SYM_DTYPE_NULL	0	No derived type; the symbol is a simple scalar variable.
IMAGE_SYM_DTYPE_POINTER	1	The symbol is a pointer to base type.
IMAGE_SYM_DTYPE_FUNCTION	2	The symbol is a function that returns a base type.
IMAGE_SYM_DTYPE_ARRAY	3	The symbol is an array of base type.

5.4.4. Storage Class

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

Although the traditional COFF format uses many storage-class values, Microsoft tools rely on Visual C++ debug 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 the Value field
IMAGE_SYM_CLASS_END_OF_FUNCTION	-1 (0xFF)	A special symbol that represents the end of function, for debugging purposes.
IMAGE_SYM_CLASS_NULL	0	No assigned storage class.
IMAGE_SYM_CLASS_AUTOMATIC	1	The automatic (stack) variable. The Value field specifies the stack frame offset.
IMAGE_SYM_CLASS_EXTERNAL	2	A value that Microsoft tools use for external symbols. The Value field indicates the size if the section number is IMAGE_SYM_UNDEFINED (0). If the section number is not zero, then the Value field specifies the offset within the section.

Constant	Value	Description/interpretation of the Value field
IMAGE_SYM_CLASS_STATIC	3	The offset of the symbol within the section. If the Value field is zero, then the symbol represents a section name.
IMAGE_SYM_CLASS_REGISTER	4	A register variable. The Value field specifies the register number.
IMAGE_SYM_CLASS_EXTERNAL_DEF	5	A symbol that is defined externally.
IMAGE_SYM_CLASS_LABEL	6	A code label that is defined within the module. The Value field specifies the offset of the symbol within the section.
IMAGE_SYM_CLASS_UNDEFINED_LABEL	7	A reference to a code label that is not defined.
IMAGE_SYM_CLASS_MEMBER_OF_STRUCT	8	The structure member. The Value field specifies the *n*th member.
IMAGE_SYM_CLASS_ARGUMENT	9	A formal argument (parameter) of a function. The Value field specifies the *n*th argument.
IMAGE_SYM_CLASS_STRUCT_TAG	10	The structure tag-name entry.
IMAGE_SYM_CLASS_MEMBER_OF_UNION	11	A union member. The Value field specifies the *n*th member.
IMAGE_SYM_CLASS_UNION_TAG	12	The Union tag-name entry.
IMAGE_SYM_CLASS_TYPE_DEFINITION	13	A Typedef entry.
IMAGE_SYM_CLASS_UNDEFINED_STATIC	14	A static data declaration.
IMAGE_SYM_CLASS_ENUM_TAG	15	An enumerated type tagname entry.
IMAGE_SYM_CLASS_MEMBER_OF_ENUM	16	A member of an enumeration. The Value field specifies the *n*th member.
IMAGE_SYM_CLASS_REGISTER_PARAM	17	A register parameter.
IMAGE_SYM_CLASS_BIT_FIELD	18	A bit-field reference. The Value field specifies the *n*th bit in the bit field.
IMAGE_SYM_CLASS_BLOCK	100	A **.bb** (beginning of block) or **.eb** (end of block) record. The Value field is the relocatable address of the code location.
IMAGE_SYM_CLASS_FUNCTION	101	A value that Microsoft tools use for symbol records that define the extent of a function: begin function **(.bf**), end function (**.ef**), and lines in function (**.lf**). For **.lf** records, the Value field gives the number of source lines in the function. For **.ef** records, the Value field gives the size of the function code.
IMAGE_SYM_CLASS_END_OF_STRUCT	102	An end-of-structure entry.

Constant	Value	Description/interpretation of the Value field
IMAGE_SYM_CLASS_FILE	103	A value that Microsoft tools, as well as traditional COFF format, use for the source-file symbol record. The symbol is followed by auxiliary records that name the file.
IMAGE_SYM_CLASS_SECTION	104	A definition of a section (Microsoft tools use STATIC storage class instead).
IMAGE_SYM_CLASS_WEAK_EXTERNAL	105	A weak external. For more information, see section 5.5.3, "Auxiliary Format 3: Weak Externals."
IMAGE_SYM_CLASS_CLR_TOKEN	107	A CLR token symbol. The name is an ASCII string that consists of the hexadecimal value of the token. For more information, see section 5.5.7, "CLR Token Definition (Object Only)."

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 can 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, file names, and section definitions.

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

5.5.1. Auxiliary Format 1: Function Definitions

A symbol table record marks the beginning of a function definition if it has all of the following: a storage class of EXTERNAL (2), a Type value that indicates it is a function (0x20), and a section number that is 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 in the format described below:

Offset	Size	Field	Description
0	4	TagIndex	The symbol-table index of the corresponding **.bf** (begin function) symbol record.
4	4	TotalSize	The size of the executable code for the function itself. If the function is in its own section, the SizeOfRawData in the section header is greater or equal to this field, depending on alignment considerations.
8	4	PointerToLinenumber	The file offset of the first COFF line-number entry for the function, or zero if none exists. For more information, see section 5.3, "COFF Line Numbers (Deprecated)."
12	4	PointerToNextFunction	The 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.

Offset	Size	Field	Description
16	2	Unused	

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

For each function definition in the symbol table, three items 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	Linenumber	The actual ordinal line number (1, 2, 3, and so on) within the source file, corresponding to the **.bf** or **.ef** record.
6	6	Unused	
12	4	PointerToNextFunction (**.bf** only)	The 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. It is not used for **.ef** records.
16	2	Unused	

5.5.3. Auxiliary Format 3: Weak Externals

"Weak externals" are a mechanism for object files that allows flexibility at link time. A module can contain an unresolved external symbol (sym1), but it can also include an auxiliary record that indicates 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 that contains 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 zero. The weak-external symbol record is followed by an auxiliary record with the following format:

Offset	Size	Field	Description
0	4	TagIndex	The symbol-table index of sym2, the symbol to be linked if sym1 is not found.
4	4	Characteristics	A value of IMAGE_WEAK_EXTERN_SEARCH_NOLIBRARY indicates that no library search for sym1 should be performed. A value of IMAGE_WEAK_EXTERN_SEARCH_LIBRARY indicates that a library search for sym1 should be performed. A value of IMAGE_WEAK_EXTERN_SEARCH_ALIAS indicates that sym1 is an alias for sym2.

Offset	Size	Field	Description
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	An ANSI string that gives the name of the source file. This is padded with nulls if it is less than the 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 about the section to which it refers. Thus, it duplicates some of the information in the section header.

Offset	Size	Field	Description
0	4	Length	The size of section data; the same as SizeOfRawData in the section header.
4	2	NumberOfRelocations	The number of relocation entries for the section.
6	2	NumberOfLinenumbers	The number of line-number entries for the section.
8	4	CheckSum	The checksum for communal data. It is applicable if the IMAGE_SCN_LNK_COMDAT flag is set in the section header. For more information, see section 5.5.6, “COMDAT Sections (Object Only).”
12	2	Number	One-based index into the section table for the associated section. This is used when the COMDAT selection setting is 5.
14	1	Selection	The COMDAT selection number. This is 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 COMDAT section is 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 Selection field determines the way in which the linker resolves the multiple definitions of COMDAT sections.

The first symbol that has the section value of the COMDAT section must be the section symbol. This symbol has the name of the section, the Value field equal to zero, the section number of the COMDAT section in question, the Type field equal to IMAGE_SYM_TYPE_NULL, the Class field equal to IMAGE_SYM_CLASS_STATIC, and one auxiliary record. The second symbol is called “the COMDAT symbol” and is used by the linker in conjunction with the Selection field.

The values for the Selection field are shown below.

Constant	Value	Description
IMAGE_COMDAT_SELECT_NODUPLICATES	1	If this symbol is already defined, the linker issues a "multiply defined symbol" error.
IMAGE_COMDAT_SELECT_ANY	2	Any section that defines the same COMDAT symbol can be linked; the rest are removed.
IMAGE_COMDAT_SELECT_SAME_SIZE	3	The linker chooses an arbitrary section among the definitions for this symbol. If all definitions are not the same size, a "multiply defined symbol" error is issued.
IMAGE_COMDAT_SELECT_EXACT_MATCH	4	The linker chooses an arbitrary section among the definitions for this symbol. If all definitions do not match exactly, a "multiply defined symbol" error is issued.
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. 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 or discarded as a set. The other section with which this section is associated must be a COMDAT section; it cannot be another associative COMDAT section (that is, the other section cannot have IMAGE_COMDAT_SELECT_ASSOCIATIVE set).
IMAGE_COMDAT_SELECT_LARGEST	6	The linker chooses the largest definition from among all of the definitions for this symbol. If multiple definitions have this size, the choice between them is arbitrary.

5.5.7. CLR Token Definition (Object Only)

This auxiliary symbol generally follows the IMAGE_SYM_CLASS_CLR_TOKEN. It is used to associate a token with the COFF symbol table's namespace.

Offset	Size	Field	Description
0	1	bAuxType	Must be IMAGE_AUX_SYMBOL_TYPE_TOKEN_DEF (1).
1	1	bReserved	Reserved, must be zero.
2	4	SymbolTableIndex	The symbol index of the COFF symbol to which this CLR token definition refers.
6	12		Reserved, must be zero.

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 that contain the total size (in bytes) of the rest of the string table. This size includes the size field itself, so that the value in this location would be 4 if no strings were present.

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

5.7. The Attribute Certificate Table (Image Only)

Attribute certificates can be associated with an image by adding an attribute certificate table. The attribute certificate table is composed of a set of contiguous, quadword-aligned attribute certificate entries. Zero padding is inserted between the original end of the file and the beginning of the attribute certificate table to achieve this alignment. Each attribute certificate entry contains the following fields.

Offset	Size	Field	Description
0	4	dwLength	Specifies the length of the attribute certificate entry.
4	2	wRevision	Contains the certificate version number. For details, see the following text.
6	2	wCertificateType	Specifies the type of content in **bCertificate**. For details, see the following text.
8	See the following	bCertificate	Contains a certificate, such as an Authenticode signature. For details, see the following text.

The virtual address value from the Certificate Table entry in the Optional Header Data Directory is a file offset to the first attribute certificate entry. Subsequent entries are accessed by advancing that entry's dwLength bytes, rounded up to an 8-byte multiple, from the start of the current attribute certificate entry. This continues until the sum of the rounded dwLength values equals the Size value from the Certificates Table entry in the Optional Header Data Directory. If the sum of the rounded dwLength values does not equal the Size value, then either the attribute certificate table or the Size field is corrupted.

For example, if the Optional Header Data Directory's Certificate Table Entry contains:

```
virtual address = 0x5000
size = 0x1000
```

The first certificate starts at offset 0x5000 from the start of the file on disk. To advance through all the attribute certificate entries:

1. Add the first attribute certificate's dwLength value to the starting offset.
2. Round the value from step 1 up to the nearest 8-byte multiple to find the offset of the second attribute certificate entry.
3. Add the offset value from step 2 to the second attribute certificate entry's dwLength value and round up to the nearest 8-byte multiple to determine the offset of the third attribute certificate entry.

4. Repeat step 3 for each successive certificate until the calculated offset equals 0x6000 (0x5000 start + 0x1000 total size), which indicates that you've walked the entire table.

Alternatively, you can enumerate the certificate entries by calling the Win32® **ImageEnumerateCertificates** function in a loop. For a link to the function's reference page, see "References."

Attribute certificate table entries can contain any certificate type, as long as the entry has the correct dwLength value, a unique wRevision value, and a unique wCertificateType value. The most common type of certificate table entry is a WIN_CERTIFICATE structure, which is documented in Wintrust.h and discussed in the remainder of this section.

The options for the WIN_CERTIFICATE **wRevision** member include (but are not limited to) the following.

Value	Name	Notes
0x0100	WIN_CERT_REVISION_1_0	Version 1, legacy version of the Win_Certificate structure. It is supported only for purposes of verifying legacy Authenticode signatures
0x0200	WIN_CERT_REVISION_2_0	Version 2 is the current version of the Win_Certificate structure.

The options for the WIN_CERTIFICATE **wCertificateType** member include (but are not limited to) the items in the following table. Note that some values are not currently supported.

Value	Name	Notes
0x0001	WIN_CERT_TYPE_X509	**bCertificate** contains an X.509 Certificate Not Supported
0x0002	WIN_CERT_TYPE_PKCS_SIGNED_DATA	**bCertificate** contains a PKCS#7 SignedData structure
0x0003	WIN_CERT_TYPE_RESERVED_1	Reserved
0x0004	WIN_CERT_TYPE_TS_STACK_SIGNED	Terminal Server Protocol Stack Certificate signing Not Supported

The WIN_CERTIFICATE structure's **bCertificate** member contains a variable-length byte array with the content type specified by **wCertificateType**. The type supported by Authenticode is WIN_CERT_TYPE_PKCS_SIGNED_DATA, a PKCS#7 **SignedData** structure. For details on the Authenticode digital signature format, see "Windows Authenticode Portable Executable Signature Format."

If the **bCertificate** content does not end on a quadword boundary, the attribute certificate entry is padded with zeros, from the end of **bCertificate** to the next quadword boundary.

The **dwLength** value is the length of the finalized WIN_CERTIFICATE structure and is computed as:

```
dwLength = offsetof(WIN_CERTIFICATE, bCertificate) + (size of the
variable-length binary array contained within bCertificate)
```

This length should include the size of any padding that is used to satisfy the requirement that each WIN_CERTIFICATE structure is quadword aligned:

```
dwLength += (8 - (dwLength & 7)) & 7;
```

The **Certificate Table size**—specified in the **Certificates Table** entry in the **Optional Header Data Directory** (section 3.4.3)—includes the padding.

For more information on using the ImageHlp API to enumerate, add, and remove certificates from PE Files, see “ImageHlp Functions.”

5.7.1. Certificate Data

As stated in the preceding section, the certificates in the attribute certificate table can contain any certificate type. Certificates that ensure a PE file's integrity may include a PE image hash.

A PE image hash (or file hash) is similar to a file checksum in that the hash algorithm produces a message digest that is related to the integrity of a file. However, a checksum is produced by a simple algorithm and is used primarily to detect whether a block of memory on disk has gone bad and the values stored there have become corrupted. A file hash is similar to a checksum in that it also detects file corruption. However, unlike most checksum algorithms, it is very difficult to modify a file without changing the file hash from its original unmodified value. A file hash can thus be used to detect intentional and even subtle modifications to a file, such as those introduced by viruses, hackers, or Trojan horse programs.

When included in a certificate, the image digest must exclude certain fields in the PE Image, such as the Checksum and Certificate Table entry in Optional Header Data Directories. This is because the act of adding a Certificate changes these fields and would cause a different hash value to be calculated.

The Win32 **ImageGetDigestStream** function provides a data stream from a target PE file with which to hash functions. This data stream remains consistent when certificates are added to or removed from a PE file. Based on the parameters that are passed to **ImageGetDigestStream**, other data from the PE image can be omitted from the hash computation. For a link to the function's reference page, see ”References.”

5.8. Delay-Load Import Tables (Image Only)

These tables were added to the image to support a uniform mechanism for applications to delay the loading of a DLL until the first call into that DLL. The layout of the tables matches that of the traditional import tables that are described in section 6.4, "The **.idata** Section." Only a few details are discussed here.

5.8.1. The Delay-Load Directory Table

The delay-load directory table is the counterpart to the import directory table. It can be retrieved through the Delay Import Descriptor entry in the optional header data directories list (offset 200). The table is arranged as follows:

Offset	Size	Field	Description
0	4	Attributes	Must be zero.
4	4	Name	The RVA of the name of the DLL to be loaded. The name resides in the read-only data section of the image.

Offset	Size	Field	Description
8	4	Module Handle	The RVA of the module handle (in the data section of the image) of the DLL to be delay-loaded. It is used for storage by the routine that is supplied to manage delay-loading.
12	4	Delay Import Address Table	The RVA of the delay-load import address table. For more information, see section 5.8.5, "Delay Import Address Table (IAT)."
16	4	Delay Import Name Table	The RVA of the delay-load name table, which contains the names of the imports that might need to be loaded. This matches the layout of the import name table. For more information, see section 6.4.3, "Hint/Name Table."
20	4	Bound Delay Import Table	The RVA of the bound delay-load address table, if it exists.
24	4	Unload Delay Import Table	The RVA of the unload delay-load address table, if it exists. This is an exact copy of the delay import address table. If the caller unloads the DLL, this table should be copied back over the delay import address table so that subsequent calls to the DLL continue to use the thunking mechanism correctly.
28	4	Time Stamp	The timestamp of the DLL to which this image has been bound.

The tables that are referenced in this data structure are organized and sorted just as their counterparts are for traditional imports. For details, see section 6.4, "The **.idata** Section."

5.8.2. Attributes

As yet, no attribute flags are defined. The linker sets this field to zero in the image. This field can be used to extend the record by indicating the presence of new fields, or it can be used to indicate behaviors to the delay or unload helper functions.

5.8.3. Name

The name of the DLL to be delay-loaded resides in the read-only data section of the image. It is referenced through the `szName` field.

5.8.4. Module Handle

The handle of the DLL to be delay-loaded is in the data section of the image. The `phmod` field points to the handle. The supplied delay-load helper uses this location to store the handle to the loaded DLL.

5.8.5. Delay Import Address Table

The delay import address table (IAT) is referenced by the delay import descriptor through the `pIAT` field. The delay-load helper updates these pointers with the real entry points so that the thunks are no longer in the calling loop. The function pointers are accessed by using the expression `pINT->u1.Function`.

5.8.6. Delay Import Name Table

The delay import name table (INT) contains the names of the imports that might require loading. They are ordered in the same fashion as the function pointers in the IAT. They consist of the same structures as the standard INT and are accessed by using the expression `pINT->u1.AddressOfData->Name[0]`.

5.8.7. Delay Bound Import Address Table and Time Stamp

The delay bound import address table (BIAT) is an optional table of IMAGE_THUNK_DATA items that is used along with the timestamp field of the delay-load directory table by a post-process binding phase.

5.8.8. Delay Unload Import Address Table

The delay unload import address table (UIAT) is an optional table of IMAGE_THUNK_DATA items that the unload code uses to handle an explicit unload request. It consists of initialized data in the read-only section that is an exact copy of the original IAT that referred the code to the delay-load thunks. On the unload request, the library can be freed, the `*phmod` cleared, and the UIAT written over the IAT to restore everything to its preload state.

6. Special Sections

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

However, some COFF sections have special meanings when found in object files or image files. Tools and loaders recognize these sections because they have special flags set in the section header, because special locations in the image optional header point to them, or because the section name itself indicates a special function of the section. (Even if the section name itself does not indicate a special function of the section, the section name is dictated by convention, so the authors of this specification can refer to a section name in all cases.)

The reserved sections and their attributes are described in the table below, followed by detailed descriptions for the section types that are persisted into executables and the section types that contain metadata for extensions.

Section Name	Content	Characteristics
.bss	Uninitialized data (free format)	IMAGE_SCN_CNT_UNINITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE
.cormeta	CLR metadata that indicates that the object file contains managed code	IMAGE_SCN_LNK_INFO
.data	Initialized data (free format)	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE
.debug$F	Generated FPO debug information (object only, x86 architecture only, and now obsolete)	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_DISCARDABLE
.debug$P	Precompiled debug types (object only)	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_DISCARDABLE
.debug$S	Debug symbols (object only)	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_DISCARDABLE
.debug$T	Debug types (object only)	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_DISCARDABLE
.drective	Linker options	IMAGE_SCN_LNK_INFO

Section Name	Content	Characteristics
.edata	Export tables	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ
.idata	Import tables	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE
.idlsym	Includes registered SEH (image only) to support IDL attributes. For information, see "IDL Attributes" in "References" at the end of this specification.	IMAGE_SCN_LNK_INFO
.pdata	Exception information	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ
.rdata	Read-only initialized data	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ
.reloc	Image relocations	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_DISCARDABLE
.rsrc	Resource directory	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ
.sbss	GP-relative uninitialized data (free format)	IMAGE_SCN_CNT_UNINITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE | IMAGE _SCN_GPREL The IMAGE_SCN_GPREL flag should be set for IA64 architectures only; this flag is not valid for other architectures. The IMAGE_SCN_GPREL flag is for object files only; when this section type appears in an image file, the IMAGE_SCN_GPREL flag must not be set.
.sdata	GP-relative initialized data (free format)	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE | IMAGE _SCN_GPREL The IMAGE_SCN_GPREL flag should be set for IA64 architectures only; this flag is not valid for other architectures. The IMAGE_SCN_GPREL flag is for object files only; when this section type appears in an image file, the IMAGE_SCN_GPREL flag must not be set.
.srdata	GP-relative read-only data (free format)	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE _SCN_GPREL The IMAGE_SCN_GPREL flag should be set for IA64 architectures only; this flag is not valid for other architectures. The IMAGE_SCN_GPREL flag is for object files only; when this section type appears in an image file, the IMAGE_SCN_GPREL flag must not be set.

Section Name	Content	Characteristics
.sxdata	Registered exception handler data (free format and x86/object only)	IMAGE_SCN_LNK_INFO Contains the symbol index of each of the exception handlers being referred to by the code in that object file. The symbol can be for an UNDEF symbol or one that is defined in that module.
.text	Executable code (free format)	IMAGE_SCN_CNT_CODE | IMAGE_SCN_MEM_EXECUTE | IIMAGE_SCN_MEM_READ
.tls	Thread-local storage (object only)	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE
.tls$	Thread-local storage (object only)	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE
.vsdata	GP-relative initialized data (free format and for ARM, SH4, and Thumb architectures only)	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE
.xdata	Exception information (free format)	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ

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 is marked “image only” might 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 all of the debug information that is generated. This section describes the packaging of debug information in object and image files.

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.

The default for the linker is that debug information is not mapped into the address space of the image. A .**debug** section exists only when debug information is mapped in the address space.

6.1.1. Debug Directory (Image Only)

Image files contain an optional debug directory that indicates 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 can be in a discardable **.debug** section (if one exists), or it can be included in any other section in the image file, or not be in a section at all.

Each debug directory entry identifies the location and size of a block of debug information. The specified RVA can be zero if the debug information is not covered by a section header (that is, 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.

A debug directory entry has the following format:

Offset	Size	Field	Description
0	4	Characteristics	Reserved, must be zero.
4	4	TimeDateStamp	The time and date that the debug data was created.
8	2	MajorVersion	The major version number of the debug data format.
10	2	MinorVersion	The minor version number of the debug data format.
12	4	Type	The format of debugging information. This field enables support of multiple debuggers. For more information, see section 6.1.2, "Debug Type."
16	4	SizeOfData	The size of the debug data (not including the debug directory itself).
20	4	AddressOfRawData	The address of the debug data when loaded, relative to the image base.
24	4	PointerToRawData	The file pointer to the debug data.

6.1.2. Debug Type

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

Constant	Value	Description
IMAGE_DEBUG_TYPE_UNKNOWN	0	An unknown value that is ignored by all tools.
IMAGE_DEBUG_TYPE_COFF	1	The 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	The Visual C++ debug information.
IMAGE_DEBUG_TYPE_FPO	3	The frame pointer omission (FPO) information. This information tells the debugger how to interpret nonstandard stack frames, which use the EBP register for a purpose other than as a frame pointer.
IMAGE_DEBUG_TYPE_MISC	4	The location of DBG file.
IMAGE_DEBUG_TYPE_EXCEPTION	5	A copy of **.pdata** section.
IMAGE_DEBUG_TYPE_FIXUP	6	Reserved.
IMAGE_DEBUG_TYPE_OMAP_TO_SRC	7	The mapping from an RVA in image to an RVA in source image.
IMAGE_DEBUG_TYPE_OMAP_FROM_SRC	8	The mapping from an RVA in source image to an RVA in image.
IMAGE_DEBUG_TYPE_BORLAND	9	Reserved for Borland.
IMAGE_DEBUG_TYPE_RESERVED10	10	Reserved.
IMAGE_DEBUG_TYPE_CLSID	11	Reserved.
IMAGE_DEBUG_TYPE_REPRO	16	PE determinism or reproducibility.

If the Type field 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 must 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 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;
```

The presence of an entry of type IMAGE_DEBUG_TYPE_REPRO indicates the PE file is built in a way to achieve determinism or reproducibility. If the input does not change, the output PE file is guaranteed to be bit-for-bit identical no matter when or where the PE is produced. Various date/time stamp fields in the PE file are filled with part or all the bits from a calculated hash value that uses PE file content as input, and therefore no longer represent the actual date and time when a PE file or related specific data within the PE is produced. The raw data of this debug entry may be empty, or may contain a calculated hash value preceded by a four-byte value that represents the hash value length.

6.1.3. .debug$F (Object Only)

The data in this section has been superseded in Visual C++ version 7.0 and later by a more extensive set of data that is emitted into a **.debug$S** subsection.

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 section 6.1.2, "Debug Type."

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

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

6.1.4. .debug$S (Object Only)

This section contains Visual C++ debug information (symbolic information).

6.1.5. .debug$P (Object Only)

This section contains Visual C++ debug information (precompiled information). These are shared types among all of the objects that were compiled by using the precompiled header that was generated with this object.

6.1.6. .debug$T (Object Only)

This section contains Visual C++ debug information (type information).

6.1.7. Linker Support for Microsoft Debug Information

To support debug information, the linker:

- Gathers all relevant debug data from the **.debug$F, debug$S, .debug$P,** and **.debug$T** sections.
- Processes that data along with the linker-generated debugging information into the PDB file, and creates a debug directory entry to refer to it.

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 and has the **.drectve** section name. The linker removes a **.drectve** section after processing the information, so the section does not appear in the image file that is being linked.

A **.drectve** section consists of a string of text that can be encoded as ANSI or UTF-8. If the UTF-8 byte order marker (BOM, a three-byte prefix that consists of 0xEF, 0xBB, and 0xBF) is not present, the directive string is interpreted as ANSI. The directive string is a series of linker options that are separated by spaces. Each option contains a hyphen, the option name, and any appropriate attribute. If an option contains spaces, the option must be enclosed in quotes. The **.drectve** section must not have relocations or line numbers.

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. Exported symbols are generally found in DLLs, but DLLs can also import symbols.

An overview of the general structure of the export section is described below. The tables described are usually contiguous in the file in the order shown (though this is not required). Only the export directory table and export address table are required to export symbols as ordinals. (An ordinal is an export that is accessed directly by its 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 a public name is defined.
Name pointer table	An array of pointers to the public export names, sorted in ascending order.
Ordinal table	An 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 are not necessarily the same as the private names that are used within the image file.

When another image file imports a symbol by name, the Win32 loader searches the name pointer table for a matching string. If a matching string is found, the associated ordinal is identified by looking up 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.

When another image file imports a symbol by ordinal, it is unnecessary to search the name pointer table for a matching string. Direct use of an ordinal is therefore

more efficient. However, an export name is easier to remember and does not require the user to know the table index for the symbol.

6.3.1. Export Directory Table

The export symbol information begins with the export directory table, which describes the remainder of the export symbol information. The export directory table contains address information that is used to resolve imports to the entry points within this image.

Offset	Size	Field	Description
0	4	Export Flags	Reserved, must be 0.
4	4	Time/Date Stamp	The time and date that the export data was created.
8	2	Major Version	The major version number. The major and minor version numbers can be set by the user.
10	2	Minor Version	The minor version number.
12	4	Name RVA	The address of the ASCII string that contains the name of the DLL. This address is relative to the image base.
16	4	Ordinal Base	The starting ordinal number for exports in this image. This field specifies the starting ordinal number for the export address table. It is usually set to 1.
20	4	Address Table Entries	The number of entries in the export address table.
24	4	Number of Name Pointers	The number of entries in the name pointer table. This is also the number of entries in the ordinal table.
28	4	Export Address Table RVA	The address of the export address table, relative to the image base.
32	4	Name Pointer RVA	The address of the export name pointer table, relative to the image base. The table size is given by the Number of Name Pointers field.
36	4	Ordinal Table RVA	The 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 as an index into the export address table.

Each entry in the export address table is a field that uses one of two formats in the following table. If the address specified is *not* within the export section (as defined by the address and length that are indicated in the optional header), the field is an export RVA, which is 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	The 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	The pointer to a null-terminated ASCII string in the export section. This string must be within the range that is given by the export table data directory entry. See section 3.4.3, "Optional Header Data Directories (Image Only)." This string gives the DLL name and the name of the export (for example, “MYDLL.expfunc”) or the DLL name and the ordinal number of the export (for example, “MYDLL.#27”).

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 XP, the export named "HeapAlloc" is forwarded to the string "NTDLL.RtlAllocateHeap." This allows applications to use the Windows XP–specific module Ntdll.dll without actually containing import references to it. The application's import table refers only to Kernel32.dll. Therefore, the application is not specific to Windows XP 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 that are 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 that was pointed to by the export name pointer table. The strings in this table are public names that other images can use 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 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 null-terminated ASCII strings of variable length.

6.4. The .idata Section

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

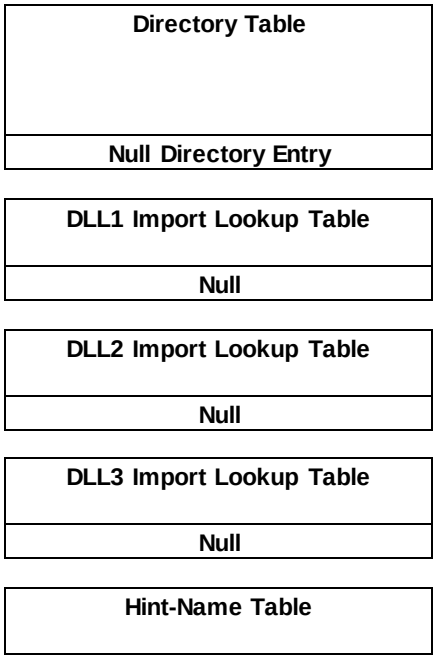

Figure 3. Typical Import Section Layout

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 fixup 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 to which the image refers. 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)	The RVA of the import lookup table. This table contains a name or ordinal for each import. (The name "Characteristics" is used in Winnt.h, but no longer describes this field.)
4	4	Time/Date Stamp	The stamp that is set to zero until the image is bound. After the image is bound, this field is set to the time/data stamp of the DLL.
8	4	Forwarder Chain	The index of the first forwarder reference.
12	4	Name RVA	The address of an ASCII string that contains the name of the DLL. This address is relative to the image base.
16	4	Import Address Table RVA (Thunk Table)	The RVA of the import address table. The contents of this table are identical to the contents of 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 for PE32 or an array of 64-bit numbers for PE32+. Each entry uses the bit-field format that is described in the following table. In this format, bit 31 is the most significant bit for PE32 and bit 63 is the most significant bit for PE32+. The collection of these entries describes all imports from a given DLL. The last entry is set to zero (NULL) to indicate the end of the table.

Bit(s)	Size	Bit field	Description
31/63	1	Ordinal/Name Flag	If this bit is set, import by ordinal. Otherwise, import by name. Bit is masked as 0x80000000 for PE32, 0x8000000000000000 for PE32+.
15-0	16	Ordinal Number	A 16-bit ordinal number. This field is used only if the Ordinal/Name Flag bit field is 1 (import by ordinal). Bits 30-15 or 62-15 must be 0.
30–0	31	Hint/Name Table RVA	A 31-bit RVA of a hint/name table entry. This field is used only if the Ordinal/Name Flag bit field is 0 (import by name). For PE32+ bits 62-31 must be zero.

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	2	Hint	An 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.
2	variable	Name	An ASCII string that contains the 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 that 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 identical to those 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 (for PE32) or 64-bit (for PE32+) addresses of the symbols that are being imported. These addresses are the actual memory addresses of the symbols, although technically they are still called "virtual addresses." The loader typically processes the binding.

6.5. The .pdata Section

The **.pdata** section contains an array of function table entries that are used for exception handling. It is pointed to by the exception table entry in the image data directory. The entries must be sorted according to the function addresses (the first field in each structure) before being emitted into the final image. The target platform determines which of the three function table entry format variations described below is used.

For 32-bit MIPS images, function table entries have the following format:

Offset	Size	Field	Description
0	4	Begin Address	The VA of the corresponding function.
4	4	End Address	The VA of the end of the function.
8	4	Exception Handler	The pointer to the exception handler to be executed.
12	4	Handler Data	The pointer to additional information to be passed to the handler.
16	4	Prolog End Address	The VA of the end of the function's prolog.

For the ARM, PowerPC, SH3 and SH4 Windows CE platforms, function table entries have the following format:

Offset	Size	Field	Description
0	4	Begin Address	The VA of the corresponding function.
4	8 bits	Prolog Length	The number of instructions in the function's prolog.
4	22 bits	Function Length	The number of instructions in the function.
4	1 bit	32-bit Flag	If set, the function consists of 32-bit instructions. If clear, the function consists of 16-bit instructions.
4	1 bit	Exception Flag	If set, an exception handler exists for the function. Otherwise, no exception handler exists.

For x64 and Itanium platforms, function table entries have the following format:

Offset	Size	Field	Description
0	4	Begin Address	The RVA of the corresponding function.
4	4	End Address	The RVA of the end of the function.
8	4	Unwind Information	The RVA of the unwind information.

6.6. The .reloc Section (Image Only)

The base relocation table contains entries for all base relocations in the image. The Base Relocation Table field in the optional header data directories gives the number of bytes in the base relocation table. For more information, see section 3.4.3, "Optional Header Data Directories (Image Only)." The base relocation table is divided into blocks. Each block represents the base relocations for a 4K page. Each block must start on a 32-bit boundary.

The loader is not required to process base relocations that are resolved by the linker, unless the load image cannot be loaded at the image base that is specified in the PE header.

6.6.1. Base Relocation Block

Each base relocation 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 VA where the base relocation must be applied.
4	4	Block Size	The total number of bytes in the base relocation block, including the Page RVA and Block Size fields and the Type/Offset fields that follow.

The Block Size field is then followed by any number of Type or Offset field entries. Each entry is a WORD (2 bytes) and has the following structure:

Offset	Size	Field	Description
0	4 bits	Type	Stored in the high 4 bits of the WORD, a value that indicates the type of base relocation to be applied. For more information, see section 6.6.2, “Base Relocation Types.”
0	12 bits	Offset	Stored in the remaining 12 bits of the WORD, an offset from the starting address that was specified in the Page RVA field for the block. This offset specifies where the base relocation is to be applied.

To apply a base relocation, the difference is calculated between the preferred base address and the base where the image is actually loaded. If the image is loaded at its preferred base, the difference is zero and thus the base relocations do not have to be applied.

6.6.2. Base Relocation Types

Constant	Value	Description
IMAGE_REL_BASED_ABSOLUTE	0	The base relocation is skipped. This type can be used to pad a block.
IMAGE_REL_BASED_HIGH	1	The base relocation adds the high 16 bits of the difference 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 base relocation adds the low 16 bits of the difference to the 16-bit field at offset. The 16-bit field represents the low half of a 32-bit word.
IMAGE_REL_BASED_HIGHLOW	3	The base relocation applies all 32 bits of the difference to the 32-bit field at offset.

Constant	Value	Description
IMAGE_REL_BASED_HIGHADJ	4	The base relocation adds the high 16 bits of the difference to the 16-bit field at offset. The 16-bit field represents the high value of a 32-bit word. The low 16 bits of the 32-bit value are stored in the 16-bit word that follows this base relocation. This means that this base relocation occupies two slots.
IMAGE_REL_BASED_MIPS_JMPADDR	5	The relocation interpretation is dependent on the machine type. When the machine type is MIPS, the base relocation applies to a MIPS jump instruction.
IMAGE_REL_BASED_ARM_MOV32	5	This relocation is meaningful only when the machine type is ARM or Thumb. The base relocation applies the 32-bit address of a symbol across a consecutive MOVW/MOVT instruction pair.
IMAGE_REL_BASED_RISCV_HIGH20	5	This relocation is only meaningful when the machine type is RISC-V. The base relocation applies to the high 20 bits of a 32-bit absolute address.
	6	Reserved, must be zero.

Constant	Value	Description
IMAGE_REL_BASED_THUMB_MOV32	7	This relocation is meaningful only when the machine type is Thumb. The base relocation applies the 32-bit address of a symbol to a consecutive MOVW/MOVT instruction pair.
IMAGE_REL_BASED_RISCV_LOW12I	7	This relocation is only meaningful when the machine type is RISC-V. The base relocation applies to the low 12 bits of a 32-bit absolute address formed in RISC-V I-type instruction format.
IMAGE_REL_BASED_RISCV_LOW12S	8	This relocation is only meaningful when the machine type is RISC-V. The base relocation applies to the low 12 bits of a 32-bit absolute address formed in RISC-V S-type instruction format.
IMAGE_REL_BASED_MIPS_JMPADDR16	9	The relocation is only meaningful when the machine type is MIPS. The base relocation applies to a MIPS16 jump instruction.
IMAGE_REL_BASED_DIR64	10	The base relocation applies the difference to the 64-bit field at offset.

6.7. The .tls Section

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

Note that any amount of TLS data can be supported by using the API calls **TlsAlloc**, **TlsFree**, **TlsSetValue**, and **TlsGetValue**. The PE or COFF implementation is an alternative approach to using the API and has the advantage of being simpler from the high-level-language programmer's viewpoint. This implementation enables TLS data to be defined and initialized similarly to ordinary static variables in a program. For example, in 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 and COFF **.tls** section specifies the following information: initialization data, callback routines for per-thread initialization and termination, and the TLS index, which are 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 expects 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" (Intel x86 platforms) or "_tls_used" (other platforms). The linker looks for this memory image 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. This behavior is Intel x86-specific.
3. The loader assigns the value of the TLS index to the place that was 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 4 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 does not 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.

6.7.1. The TLS Directory

The TLS directory has the following format:

Offset (PE32/ PE32+)	Size (PE32/ PE32+)	Field	Description
0	4/8	Raw Data Start VA	The starting address of the TLS template. The template is a block of data that is used to initialize TLS data. The system copies all of 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.
4/8	4/8	Raw Data End VA	The address of the last byte of the TLS, except for the zero fill. As with the Raw Data Start VA field, this is a VA, not an RVA.
8/16	4/8	Address of Index	The 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 that is accessible to the program.
12/24	4/8	Address of Callbacks	The pointer to an array of TLS callback functions. The array is null-terminated, so if no callback function is supported, this field points to 4 bytes set to zero. For information about the prototype for these functions, see section 6.7.2, "TLS Callback Functions."
16/32	4	Size of Zero Fill	The size in bytes of the template, beyond the initialized data delimited by the Raw Data Start VA and Raw Data End VA fields. 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.
20/36	4	Characteristics	The four bits [23:20] describe alignment info. Possible values are those defined as IMAGE_SCN_ALIGN_*, which are also used to describe alignment of section in object files. The other 28 bits are reserved for future use.

6.7.2. TLS Callback Functions

The program can provide one or more TLS callback functions to support additional initialization and termination for TLS data objects. A typical use for 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 in which 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 set to zero. The Reason parameter can take the following values:

Setting	Value	Description
DLL_PROCESS_ATTACH	1	A new process has started, including the first thread.
DLL_THREAD_ATTACH	2	A new thread has been created. This notification sent for all but the first thread.
DLL_THREAD_DETACH	3	A thread is about to be terminated. This notification sent for all but the first thread.
DLL_PROCESS_DETACH	0	A process is about to terminate, including the original thread.

6.8. The Load Configuration Structure (Image Only)

The load configuration structure (IMAGE_LOAD_CONFIG_DIRECTORY) was formerly used in very limited cases in the Windows NT operating system itself to describe various features too difficult or too large to describe in the file header or optional header of the image. Current versions of the Microsoft linker and Windows XP and later versions of Windows use a new version of this structure for 32-bit x86-based systems that include reserved SEH technology. This provides a list of safe structured exception handlers that the operating system uses during exception dispatching. If the handler address resides in an image's VA range and is marked as reserved SEH-aware (that is, IMAGE_DLLCHARACTERISTICS_NO_SEH is clear in the DllCharacteristics field of the optional header, as described earlier), then the handler must be in the list of known safe handlers for that image. Otherwise, the operating system terminates the application. This helps prevent the "x86 exception handler hijacking" exploit that has been used in the past to take control of the operating system.

The Microsoft linker automatically provides a default load configuration structure to include the reserved SEH data. If the user code already provides a load configuration structure, it must include the new reserved SEH fields. Otherwise, the linker cannot include the reserved SEH data and the image is not marked as containing reserved SEH.

6.8.1. Load Configuration Directory

The data directory entry for a pre-reserved SEH load configuration structure must specify a particular size of the load configuration structure because the operating system loader always expects it to be a certain value. In that regard, the size is really only a version check. For compatibility with Windows XP and earlier versions of Windows, the size must be 64 for x86 images.

6.8.2. Load Configuration Layout

The load configuration structure has the following layout for 32-bit and 64-bit PE files:

Offset	Size	Field	Description
0	4	Characteristics	Flags that indicate attributes of the file, currently unused.
4	4	TimeDateStamp	Date and time stamp value. The value is represented in the number of seconds that have elapsed since midnight (00:00:00), January 1, 1970, Universal Coordinated Time, according to the system clock. The time stamp can be printed by using the C runtime (CRT) **time** function.
8	2	MajorVersion	Major version number.
10	2	MinorVersion	Minor version number.
12	4	GlobalFlagsClear	The global loader flags to clear for this process as the loader starts the process.
16	4	GlobalFlagsSet	The global loader flags to set for this process as the loader starts the process.
20	4	CriticalSectionDefaultTimeout	The default timeout value to use for this process's critical sections that are abandoned.
24	4/8	DeCommitFreeBlockThreshold	Memory that must be freed before it is returned to the system, in bytes.
28/32	4/8	DeCommitTotalFreeThreshold	Total amount of free memory, in bytes.
32/40	4/8	LockPrefixTable	[x86 only] The VA of a list of addresses where the LOCK prefix is used so that they can be replaced with NOP on single processor machines.
36/48	4/8	MaximumAllocationSize	Maximum allocation size, in bytes.
40/56	4/8	VirtualMemoryThreshold	Maximum virtual memory size, in bytes.
44/64	4/8	ProcessAffinityMask	Setting this field to a non-zero value is equivalent to calling **SetProcessAffinityMask** with this value during process startup (.exe only)
48/72	4	ProcessHeapFlags	Process heap flags that correspond to the first argument of the **HeapCreate** function. These flags apply to the process heap that is created during process startup.

Offset	Size	Field	Description
52/76	2	CSDVersion	The service pack version identifier.
54/78	2	Reserved	Must be zero.
56/80	4/8	EditList	Reserved for use by the system.
60/88	4/8	SecurityCookie	A pointer to a cookie that is used by Visual C++ or GS implementation.
64/96	4/8	SEHandlerTable	[x86 only] The VA of the sorted table of RVAs of each valid, unique SE handler in the image.
68/104	4/8	SEHandlerCount	[x86 only] The count of unique handlers in the table.
72/112	4/8	GuardCFCheckFunctionPointer	The VA where Control Flow Guard check-function pointer is stored.
76/120	4/8	GuardCFDispatchFunctionPointer	The VA where Control Flow Guard dispatch-function pointer is stored.
80/128	4/8	GuardCFFunctionTable	The VA of the sorted table of RVAs of each Control Flow Guard function in the image.
84/136	4/8	GuardCFFunctionCount	The count of unique RVAs in the above table.
88/144	4	GuardFlags	Control Flow Guard related flags.
92/148	12	CodeIntegrity	Code integrity information.
104/160	4/8	GuardAddressTakenIatEntryTable	The VA where Control Flow Guard address taken IAT table is stored.
108/168	4/8	GuardAddressTakenIatEntryCount	The count of unique RVAs in the above table.
112/176	4/8	GuardLongJumpTargetTable	The VA where Control Flow Guard long jump target table is stored.
116/184	4/8	GuardLongJumpTargetCount	The count of unique RVAs in the above table.

The GuardFlags field contains a combination of one or more of the following flags and subfields:

- Module performs control flow integrity checks using system-supplied support.

  #define IMAGE_GUARD_CF_INSTRUMENTED 0x00000100

- Module performs control flow and write integrity checks.

  #define IMAGE_GUARD_CFW_INSTRUMENTED 0x00000200

- Module contains valid control flow target metadata.

  #define IMAGE_GUARD_CF_FUNCTION_TABLE_PRESENT 0x00000400

- Module does not make use of the /GS security cookie.

  #define IMAGE_GUARD_SECURITY_COOKIE_UNUSED 0x00000800

- Module supports read only delay load IAT.

  #define IMAGE_GUARD_PROTECT_DELAYLOAD_IAT 0x00001000

- Delayload import table in its own .didat section (with nothing else in it) that can be freely reprotected.

```
#define IMAGE_GUARD_DELAYLOAD_IAT_IN_ITS_OWN_SECTION    0x00002000
```

- Module contains suppressed export information. This also infers that the address taken IAT table is also present in the load config.

  ```
  #define IMAGE_GUARD_CF_EXPORT_SUPPRESSION_INFO_PRESENT    0x00004000
  ```

- Module enables suppression of exports.

  ```
  #define IMAGE_GUARD_CF_ENABLE_EXPORT_SUPPRESSION    0x00008000
  ```

- Module contains longjmp target information.

  ```
  #define IMAGE_GUARD_CF_LONGJUMP_TABLE_PRESENT    0x00010000
  ```

- Mask for the subfield that contains the stride of Control Flow Guard function table entries (that is, the additional count of bytes per table entry).

  ```
  #define IMAGE_GUARD_CF_FUNCTION_TABLE_SIZE_MASK    0xF0000000
  ```

Additionally, the Windows SDK winnt.h header defines this macro for the amount of bits to right-shift the GuardFlags value to right-justify the Control Flow Guard function table stride:

```
#define IMAGE_GUARD_CF_FUNCTION_TABLE_SIZE_SHIFT    28
```

6.9. 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 uses three levels:

Type
Name
Language

A series of resource directory tables relates all of the levels in the following way: Each directory table is followed by a series of directory entries that give the name or identifier (ID) for that level (Type, Name, or Language level) and an address of either a data description or another directory table. If the address points to a data description, then the data is a leaf in the tree. If the address points to another directory table, then that table lists directory entries at the next level down.

A leaf's Type, Name, and Language IDs are determined by the path that is 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 ID but different Name IDs. Third-level trees have the same Type and Name IDs but different Language IDs. Each individual table is immediately followed by directory entries, in which each entry has a name or numeric identifier and 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 that is pointed to by directory entries.

Data	Description
Resource Data Description	An array of records, pointed to by tables, that 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 field delimit the individual regions of resource data.

6.9.1. 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 (described in section 6.9.2, "Resource Directory Entries") and this structure:

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

6.9.2. Resource Directory Entries

The directory entries make up the rows of a table. Each resource directory entry has the following format. 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-sensitive string and the ID entries by numeric value. Offsets are relative to the address in the IMAGE_DIRECTORY_ENTRY_RESOURCE DataDirectory.

Offset	Size	Field	Description
0	4	Name Offset	The offset of a string that gives the Type, Name, or Language ID entry, depending on level of table.
0	4	Integer ID	A 32-bit integer that identifies the Type, Name, or Language ID entry.
4	4	Data Entry Offset	High bit 0. Address of a Resource Data entry (a leaf).
4	4	Subdirectory Offset	High bit 1. The lower 31 bits are the address of another resource directory table (the next level down).

6.9.3. 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	The size of the string, not including length field itself.
2	variable	Unicode String	The variable-length Unicode string data, word-aligned.

6.9.4. Resource Data Entry

Each Resource Data entry describes an actual unit of raw data in the Resource Data area. A Resource Data entry has the following format:

Offset	Size	Field	Description
0	4	Data RVA	The address of a unit of resource data in the Resource Data area.
4	4	Size	The size, in bytes, of the resource data that is pointed to by the Data RVA field.
8	4	Codepage	The code page that is 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 0.	

6.10.The .cormeta Section (Object Only)

CLR metadata is stored in this section. It is used to indicate that the object file contains managed code. The format of the metadata is not documented, but can be handed to the CLR interfaces for handling metadata.

6.11.The .sxdata Section

The valid exception handlers of an object are listed in the **.sxdata** section of that object. The section is marked IMAGE_SCN_LNK_INFO. It contains the COFF symbol index of each valid handler, using 4 bytes per index.

Additionally, the compiler marks a COFF object as registered SEH by emitting the absolute symbol “@feat.00” with the LSB of the value field set to 1. A COFF object with no registered SEH handlers would have the “@feat.00” symbol, but no **.sxdata** section.

7. Archive (Library) File Format

The COFF archive format provides a standard mechanism for storing collections of object files. These collections are commonly called *libraries* in programming documentation.

The first 8 bytes of an archive consist of the file signature. The rest of the archive consists of a series of archive members, as follows:

- The first and second members are “linker members.” Each of these members has its own format as described in section 8.3, "Import Name Type." Typically, a linker places information into these archive members. The linker members contain the directory of the archive.
- The third member is the "longnames" member. This member consists of a series of null-terminated ASCII strings in which each string is the name of another archive member.
- The rest of the archive consists of standard (object-file) members. Each of these members contains the contents of one object file in its entirety.

An archive member header precedes each member. The following figure shows the general structure of an archive:

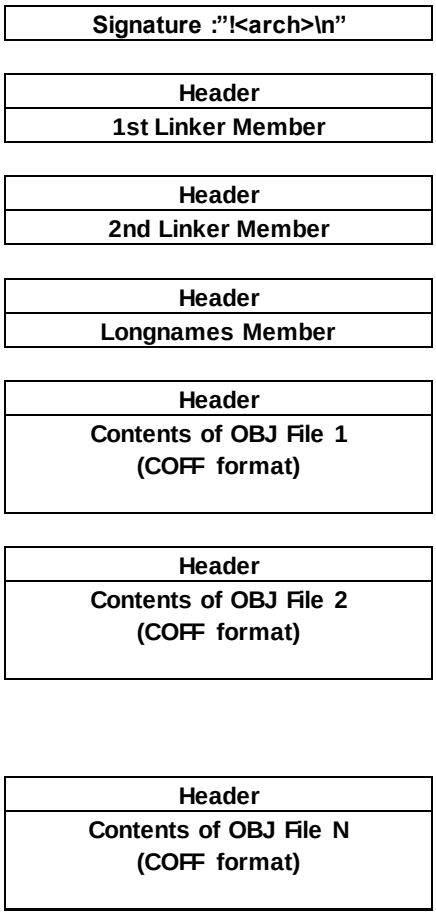

Figure 4. Archive File Structure

7.1. Archive File Signature

The archive file signature identifies the file type. Any utility (for example, a linker) that takes an archive file as input can check the file type by reading this signature. The signature consists of the following ASCII characters, in which each character below is represented literally, except for the newline (\n) character:

```
!<arch>\n
```

7.2. Archive Member Headers

Each member (linker, longnames, or object-file member) is preceded by a header. An archive member header has the following format, in which each field is an ASCII text string that is left justified and padded with spaces to the end of the field. There is no terminating null character in any of these fields.

Each member header starts on the first even address after the end of the previous archive member.

Offset	Size	Field	Description
0	16	Name	The name of the archive member, with a slash (/) appended to terminate the name. If the first character is a slash, the name has a special interpretation, as described in the following table.
16	12	Date	The date and time that the archive member was created: This is the ASCII decimal representation of the number of seconds since 1/1/1970 UCT.
28	6	User ID	An ASCII decimal representation of the user ID. This field does not contain a meaningful value on Windows platforms because Microsoft tools emit all blanks.
34	6	Group ID	An ASCII decimal representation of the group ID. This field does not contain a meaningful value on Windows platforms because Microsoft tools emit all blanks.
40	8	Mode	An ASCII octal representation of the member's file mode. This is the ST_MODE value from the C run-time function _wstat.
48	10	Size	An ASCII decimal representation of the total size of the archive member, not including the size of the header.
58	2	End of Header	The two bytes in the C string "`\n" (0x60 0x0A).

The Name field has one of the formats shown in the following table. As mentioned earlier, each of these strings is left justified and padded with trailing spaces within a field of 16 bytes:

Contents of Name field	Description
name/	The name of the archive member.
/	The archive member is one of the two linker members. Both of the linker members have this name.
//	The archive member is the longnames member, which consists of a series of null-terminated ASCII strings. The longnames member is the third archive member and must always be present even if the contents are empty.
/n	The name of the archive member is located at offset *n* within the longnames member. The number *n* is the decimal representation of the offset. For example: "/26" indicates that the name of the archive member is located 26 bytes beyond the beginning of the longnames member contents.

7.3. First Linker Member

The name of the first linker member is "\". The first linker member is included for backward compatibility. It is not used by current linkers, but its format must be correct. This linker member provides a directory of symbol names, as does the second linker member. For each symbol, the information indicates where to find the archive member that contains the symbol.

The first linker member has the following format. This information appears after the header:

Offset	Size	Field	Description
0	4	Number of Symbols	Unsigned **long** that contains the number of indexed symbols. This number is stored in big-endian format. Each object-file member typically defines one or more external symbols.
4	4 * n	Offsets	An array of file offsets to archive member headers, in which *n* is equal to the Number of Symbols field. Each number in the array is an unsigned **long** stored in big-endian format. For each symbol that is named in the string table, the corresponding element in the offsets array gives the location of the archive member that contains the symbol.
*	*	String Table	A series of null-terminated strings that name all the symbols in the directory. Each string begins immediately after the null character in the previous string. The number of strings must be equal to the value of the Number of Symbols field.

The elements in the offsets array must be arranged in ascending order. This fact implies that the symbols in the string table must be arranged according to the order of archive members. For example, all the symbols in the first object-file member would have to be listed before the symbols in the second object file.

7.4. Second Linker Member

The second linker member has the name "\" as does the first linker member. Although both linker members provide a directory of symbols and archive members that contain them, the second linker member is used in preference to the first by all current linkers. The second linker member includes symbol names in lexical order, which enables faster searching by name.

The second member has the following format. This information appears after the header:

Offset	Size	Field	Description
0	4	Number of Members	An unsigned **long** that contains the number of archive members.
4	4 * m	Offsets	An array of file offsets to archive member headers, arranged in ascending order. Each offset is an unsigned **long**. The number *m* is equal to the value of the Number of Members field.
*	4	Number of Symbols	An unsigned **long** that contains the number of symbols indexed. Each object-file member typically defines one or more external symbols.
*	2 * n	Indices	An array of 1-based indexes (unsigned **short**) that map symbol names to archive member offsets. The number *n* is equal to the Number of Symbols field. For each symbol that is named in the string table, the corresponding element in the Indices array gives an index into the offsets array. The offsets array, in turn, gives the location of the archive member that contains the symbol.
*	*	String Table	A series of null-terminated strings that name all of the symbols in the directory. Each string begins immediately after the null byte in the previous string. The number of strings must be equal to the value of the Number of Symbols field. This table lists all the symbol names in ascending lexical order.

7.5. Longnames Member

The name of the longnames member is "\\". The longnames member is a series of strings of archive member names. A name appears here only when there is insufficient room in the Name field (16 bytes). The longnames member can be empty, though its header must appear.

The strings are null-terminated. Each string begins immediately after the null byte in the previous string.

8. Import Library Format

Traditional import libraries, that is, libraries that describe the exports from one image for use by another, typically follow the layout described in section 7, "Archive (Library) File Format." The primary difference is that import library members contain pseudo-object files instead of real ones, in which each member includes the section contributions that are required to build the import tables that are described in section 6.4, "The **.idata** Section." The linker generates this archive while building the exporting application.

The section contributions for an import can be inferred from a small set of information. The linker can either generate the complete, verbose information into the import library for each member at the time of the library's creation or write only the canonical information to the library and let the application that later uses it generate the necessary data on the fly.

In an import library with the long format, a single member contains the following information:

Archive member header
File header
Section headers
Data that corresponds to each of the section headers
COFF symbol table
Strings

In contrast, a short import library is written as follows:

Archive member header
Import header
Null-terminated import name string
Null-terminated DLL name string

This is sufficient information to accurately reconstruct the entire contents of the member at the time of its use.

8.1. Import Header

The import header contains the following fields and offsets:

Offset	Size	Field	Description
0	2	Sig1	Must be IMAGE_FILE_MACHINE_UNKNOWN. For more information, see section 3.3.1, "Machine Types."
2	2	Sig2	Must be 0xFFFF.
4	2	Version	The structure version.
6	2	Machine	The number that identifies the type of target machine. For more information, see section 3.3.1, "Machine Types."
8	4	Time-Date Stamp	The time and date that the file was created.

Offset	Size	Field	Description
12	4	Size Of Data	The size of the strings that follow the header.
16	2	Ordinal/Hint	Either the ordinal or the hint for the import, determined by the value in the Name Type field.
18	2 bits	Type	The import type. For specific values and descriptions, see section 8.2, "Import Type."
	3 bits	Name Type	The import name type. For specific values and descriptions, see section 8.3. Import Name Type."
	11 bits	Reserved	Reserved, must be 0.

This structure is followed by two null-terminated strings that describe the imported symbol's name and the DLL from which it came.

8.2. Import Type

The following values are defined for the Type field in the import header:

Constant	Value	Description
IMPORT_CODE	0	Executable code.
IMPORT_DATA	1	Data.
IMPORT_CONST	2	Specified as CONST in the **.def** file.

These values are used to determine which section contributions must be generated by the tool that uses the library if it must access that data.

8.3. Import Name Type

The null-terminated import symbol name immediately follows its associated import header. The following values are defined for the Name Type field in the import header. They indicate how the name is to be used to generate the correct symbols that represent the import:

Constant	Value	Description
IMPORT_ORDINAL	0	The import is by ordinal. This indicates that the value in the Ordinal/Hint field of the import header is the import's ordinal. If this constant is not specified, then the Ordinal/Hint field should always be interpreted as the import's hint.
IMPORT_NAME	1	The import name is identical to the public symbol name.
IMPORT_NAME_NOPREFIX	2	The import name is the public symbol name, but skipping the leading ?, @, or optionally_.
IMPORT_NAME_UNDECORATE	3	The import name is the public symbol name, but skipping the leading ?, @, or optionally_, and truncating at the first @.

Appendix A: Calculating Authenticode PE Image Hash

Several attribute certificates are expected to be used to verify the integrity of the images. However, the most common is Authenticode signature. An Authenticode signature can be used to verify that the relevant sections of a PE image file have not been altered in any way from the file's original form. To accomplish this task, Authenticode signatures contain something called a PE image hash.

A.1 What is an Authenticode PE Image Hash?

The Authenticode PE image hash, or file hash for short, is similar to a file checksum in that it produces a small value that relates to the integrity of a file. A checksum is produced by a simple algorithm and is used primarily to detect memory failures. That is, it is used to detect whether a block of memory on disk has gone bad and the values stored there have become corrupted. A file hash is similar to a checksum in that it also detects file corruption. However, unlike most checksum algorithms, it is very difficult to modify a file so that it has the same file hash as its original (unmodified) form. That is, a checksum is intended to detect simple memory failures that lead to corruption, but a file hash can be used to detect intentional and even subtle modifications to a file, such as those introduced by viruses, hackers, or Trojan horse programs.

In an Authenticode signature, the file hash is digitally signed by using a private key known only to the signer of the file. A software consumer can verify the integrity of the file by calculating the hash value of the file and comparing it to the value of signed hash contained in the Authenticode digital signature. If the file hashes do not match, part of the file covered by the PE image hash has been modified.

A.2 What is Covered in an Authenticode PE Image Hash?

It is not possible or desirable to include all image file data in the calculation of the PE image hash. Sometimes it simply presents undesirable characteristics (for example, debugging information cannot be removed from publicly released files); sometimes it is simply impossible. For example, it is not possible to include all information within an image file in an Authenticode signature, then insert the Authenticode signature that contains that PE image hash into the PE image, and later be able to generate an identical PE image hash by including all image file data in the calculation again, because the file now contains the Authenticode signature that was not originally there.

This appendix illustrates how a PE image hash is calculated and what parts of the PE image can be modified without invalidating the Authenticode signature.

It is worth noting that the PE image hash for a specific file can be included in a separate catalog file without including an attribute certificate within the hashed file. This is relevant, because it becomes possible to invalidate the PE image hash in an Authenticode-signed catalog file by modifying a PE image that does not actually contain an Authenticode signature.

Process for Generating the Authenticode PE Image Hash

All data in sections of the PE image that are specified in the section table are hashed in their entirety except for the following exclusion ranges:

- **The file CheckSum field of the Windows-specific fields of the optional header**. This checksum includes the entire file (including any attribute certificates in the file). In all likelihood, the checksum will be different than the original value after inserting the Authenticode signature.

- **Information related to attribute certificates**. The areas of the PE image that are related to the Authenticode signature are not included in the calculation of the PE image hash because Authenticode signatures can be added to or removed from an image without affecting the overall integrity of the image. This is not a problem, because there are user scenarios that depend on re-signing PE images or adding a time stamp. Authenticode excludes the following information from the hash calculation:

  The Certificate Table field of the optional header data directories.

  The Certificate Table and corresponding certificates that are pointed to by the Certificate Table field listed immediately above.

  To calculate the PE image hash, Authenticode orders the sections that are specified in the section table by address range, then hashes the resulting sequence of bytes, passing over the exclusion ranges.

- **Information past of the end of the last section**. The area past the last section (defined by highest offset) is not hashed. This area commonly contains debug information. Debug information can generally be considered advisory to debuggers; it does not affect the actual integrity of the executable program. It is quite literally possible to remove debug information from an image after a product has been delivered and not affect the functionality of the program. In fact, this is sometimes done as a disk-saving measure. It is worth noting that debug information contained within the specified sections of the PE Image cannot be removed without invalidating the Authenticode signature.

You can use the makecert and signtool tools provided in the Windows Platform SDK to experiment with creating and verifying Authenticode signatures. For more information, see "References" at the end of this specification.

References

Downloads and tools for Windows (includes the Windows SDK)
https://developer.microsoft.com/en-us/windows/downloads

Creating, Viewing, and Managing Certificates
https://msdn.microsoft.com/en-us/library/aa379872.aspx

Kernel-Mode Code Signing Walkthrough
http://www.microsoft.com/whdc/winlogo/drvsign/kmcs_walkthrough.mspx

SignTool
https://msdn.microsoft.com/en-us/library/aa387764.aspx

Windows Authenticode Portable Executable Signature Format
http://www.microsoft.com/whdc/winlogo/drvsign/Authenticode_PE.mspx

ImageHlp Functions
https://msdn.microsoft.com/en-us/library/ms680181.aspx