Tool Interface Standard (TIS)
Formats Specification for Windows™

Version 1.0

TIS Committee
February 1993

Additional copies of this manual can be obtained from:

Intel Corporation
Literature Center
P.O. Box 7641
Mt. Prospect, IL 60056-7641

Or call: 1-800-548-4725

Refer to Intel Order Number 241597.

Introduction

This Tool Interface Standards Formats Specification for Windows™, Version 1.0 is the result of the work of the TIS Committee--an association of members of the microcomputer industry formed to work toward standardization of the software interfaces visible to development tools for 32-bit Intel X86 operating environments. Such interfaces include object module formats, executable file formats, and debug record information and formats.

The goal of the committee is to help streamline the software development process throughout the microcomputer industry, currently concentrating on 32-bit operating environments. To that end, the committee has developed two specifications--one for file formats that are portable across leading industry operating systems and another describing formats for 32-bit Windows operating systems. These specifications will allow software developers to standardize on a set of binary interface definitions that extend across multiple operating environments and reduce the number of different interface implementations that currently must be considered in any single environment. This should permit developers to spend their time innovating and adding value instead of recoding or recompiling for yet another tool interface format.

TIS Committee members include representatives from Borland International Corporation, IBM Corporation, Intel Corporation, Lotus Corporation, MetaWare Corporation, Microsoft Corporation, The Santa Cruz Operation, and WATCOM International Corporation. PharLap Software Incorporated and Symantec Corporation also participated in the specification definition efforts.

TIS Portable Formats Specification, Version 1.0 and TIS Formats Specification for Windows™, Version 1.0 are the first deliverables of the TIS Committee. They are based on existing, proven formats in keeping with the TIS Committee's goal to adopt, and when necessary, extend existing standards rather than invent new ones.

Within the Formats Specification for Windows are definitions for both loadable and debug formats. The following table shows which standards are included and the source of each:

Tool Interface Type	**Tool Interface Format**	**Industry Source**
Loadable	PE (Portable Executable)	Microsoft Corporation
Debug	MS Symbol and Type Information	Microsoft Corporation

These, in conjunction with the portable formats, represent the tool interfaces currently agreed upon by TIS Committee members as TIS standards. In the future, the Committee expects to work on standardization efforts for tool interfaces in other areas that will benefit the microcomputer software industry, such as dump file formats, object mapping, and 64-bit operating environments.

Table of Contents

I

Portable Executable (PE) Format

TIS Formats Specification for Windows™, Version 1.0
Portable Executable (PE) Format

The following document is provided by Microsoft Corporation as a definition of the Portable Executable Format (PE). PE is the native executable format for the Microsoft Windows NT 32-bit operating system. The TIS Committee formed a subcommittee to evaluate the widely available formats with the objective of adopting one as the TIS standard. After studying many different executable formats, the committee recommended PE as a loadable information format standard for Windows environments.

No technical modifications have been made by the TIS committee. All information contained herein is provided and controlled by Microsoft Corporation.

Portable Executable Formats

TABLE OF CONTENTS

1.0 OVERVIEW

DOS 2.0 Compatible EXE Header
Unused
OEM Identifier OEM Info Offset to PE Header
DOS 2.0 Stub Program & Relocation Information
Unused
PE Header (aligned on 8-byte boundary)
Object Table
Image Pages import info export info fixup info resource info debug info

Figure 1. A Typical 32-bit Portable EXE File Layout

2.0 PE HEADER

SIGNATURE STAMP		CPU TYPE	# OBJECTS
TIME/DATE STAMP		RESERVED	
RESERVED		NT HDR SIZE	FLAGS
RESERVED	LMAJOR	LMINOR	RESERVED
RESERVED		RESERVED	
ENTRYPOINT RVA		RESERVED	
RESERVED		IMAGE BASE	
OBJECT ALIGN		FILE ALIGN	
OS MAJOR	OS MINOR	USER MAJOR	USER MINOR
SUBSYS MAJOR	SUBSYS MINOR	RESERVED	
IMAGE SIZE		HEADER SIZE	
FILE CHECKSUM		SUBSYSTEM	DLL FLAGS
STACK RESERVE SIZE		STACK COMMIT SIZE	
HEAP RESERVE SIZE		HEAP COMMIT SIZE	
RESERVED		# INTERESTING RVA/SIZES	
EXPORT TABLE RVA		TOTAL EXPORT DATA SIZE	
IMPORT TABLE RVA		TOTAL IMPORT DATA SIZE	
RESOURCE TABLE RVA		TOTAL RESOURCE DATA SIZE	
EXCEPTION TABLE RVA		TOTAL EXCEPTION DATA SIZE	
SECURITY TABLE RVA		TOTAL SECURITY DATA SIZE	
FIXUP TABLE RVA		TOTAL FIXUP DATA SIZE	
DEBUG TABLE RVA		TOTAL DEBUG DIRECTORIES	
IMAGE DESCRIPTION RVA		TOTAL DESCRIPTION SIZE	
MACHINE SPECIFIC RVA		MACHINE SPECIFIC SIZE	
THREAD LOCAL STORAGE RVA		TOTAL TLS SIZE	

Figure 2. The PE Header

Notes:

- A VA is a virtual address that is already biased by the Image Base found in the PE Header. An RVA is a virtual address that is relative to the Image Base.
- An RVA in the PE Header that has a value of zero indicates the field isn't used.
- Image pages are aligned and zero padded to a File Align boundaries. The bases of all other tables and structures must be aligned on DWORD (4 byte) boundaries. Thus, all VA's and RVA's must be on a 32-bit boundary. All table and structure fields must be aligned on their "natural" boundaries, with the possible exception of the Debug Info.

SIGNATURE BYTES = DB * 4

Current value is "PE/0/0"; PE is followed by two zeros (nulls).

CPU TYPE = DW CPU Type

This field specifies the type of CPU compatibility required by this image to run. The values are:

Value	CPU Type
0000h	Unknown
014Ch	80386
014Dh	80486
014Eh	Pentium™
0162h	MIPS Mark I (R2000, R3000)
0163h	MIPS Mark II (R6000)
0166h	MIPS Mark III (R4000)

OBJECTS = DW

Number of object entries. This field specifies the number of entries in the Object Table.

TIME/DATE STAMP = DD

Used to store the time and date the file was created or modified by the linker.

NT HDR SIZE = DW

This is the number of remaining bytes in the NT header that follows the Flags field.

FLAGS = DW

Flag bits for the image. The flag bits have the following definitions:

Flag Bit	Definition
0000h	Program image
0002h	Image is executable. If this bit isn't set, then it indicates that either errors were detected at link time or that the image is being incrementally linked and therefore can't be loaded.
0200h	Fixed. Indicates that if the image can't be loaded at the Image Base then do not load it.
2000h	Library image

LMAJOR/LMINOR = DB

The major/minor version number of the linker.

ENTRYPOINT RVA = DD

Entrypoint relative virtual address. The address is relative to the Image Base. The address is the starting address for program images and the library initialization and library termination address for library images.

IMAGE BASE = DD

The virtual base of the image. This will be the virtual address of the first byte of the file (DOS Header). This must be a multiple of 64K.

OBJECT ALIGN = DD

The alignment of the objects. This must be a power of 2 between 512 and 256M inclusive. The default is 64K.

FILE ALIGN = DD

Alignment factor used to align image pages. The alignment factor (in bytes) used to align the base of the image pages and to determine the granularity of per-object trailing zero pad. Larger alignment factors will cost more file space; smaller alignment factors will impact demand load performance, perhaps significantly. Of the two, wasting file space is preferable. This value should be a power of 2 between 512 and 64K inclusive.

OS MAJOR/MINOR = DW

The OS version number required to run this image.

USER MAJOR/MINOR # = DW

User major/minor version number. This is useful for differentiating between revisions of images/dynamic linked libraries. The values are specified at link time by the user.

SUBSYS MAJOR/MINOR # = DW

Subsystem major/minor version number.

IMAGE SIZE = DD

The virtual size (in bytes) of the image.
This includes all headers. The total image size must be a multiple of Object Align.

HEADER SIZE = DD

Total header size. The combined size of the DOS Header, PE Header and Object Table.

FILE CHECKSUM = DD

Checksum for entire file. Set to zero by the linker.

SUBSYSTEM = DW

NT subsystem required to run this image. The values are:

0000h - Unknown

0001h - Native

0002h - Windows GUI

0003h - Windows Character

0005h - OS/2 Character

0007h - POSIX Character

DLL FLAGS = DW

Indicates special loader requirements. This flag has the following bit values:

0001h - Per-Process Library Initialization

0002h - Per-Process Library Termination

0004h - Per-Thread Library Initialization

0008h - Per-Thread Library Termination

All other bits are reserved for future use and should be set to zero.

STACK RESERVE SIZE = DD

Stack size needed for image. The memory is reserved, but only the Stack Commit Size is committed. The next page of the stack is a 'guarded page.' When the application hits the guarded page, the guarded page becomes valid, and the next page becomes the guarded page. This continues until the Reserve Size is reached.

STACK COMMIT SIZE = DD

Stack commit size.

HEAP RESERVE SIZE = DD

Size of local heap to reserve.

HEAP COMMIT SIZE = DD

Amount to commit in local heap.

 # INTERESTING VA/SIZES = DD

Indicates the size of the VA/Size array that follows.

EXPORT TABLE RVA = DD

Relative Virtual Address (RVA) of the Export Table. This address is relative to the Image Base.

IMPORT TABLE RVA = DD

Relative Virtual Address of the Import Table. This address is relative to the Image Base.

RESOURCE TABLE RVA = DD

Relative Virtual Address of the Resource Table. This address is relative to the Image Base.

EXCEPTION TABLE RVA = DD

Relative Virtual Address of the Exception Table. This address is relative to the Image Base.

SECURITY TABLE RVA = DD

Relative Virtual Address of the Security Table. This address is relative to the Image Base.

FIXUP TABLE RVA = DD

Relative Virtual Address of the Fixup Table. This address is relative to the Image Base.

DEBUG TABLE RVA = DD

Relative Virtual Address of the Debug Table. This address is relative to the Image Base.

IMAGE DESCRIPTION RVA = DD

Relative Virtual Address of the description string specified in the module definition file.

MACHINE SPECIFIC RVA = DD

Relative Virtual Address of a machine-specific value. This address is relative to the Image Base.

TOTAL EXPORT DATA SIZE = DD

Total size of the export data.

TOTAL IMPORT DATA SIZE = DD

Total size of the import data.

TOTAL RESOURCE DATA SIZE = DD

Total size of the resource data.

TOTAL EXCEPTION DATA SIZE = DD

Total size of the exception data.

TOTAL SECURITY DATA SIZE = DD

Total size of the security data.

TOTAL FIXUP DATA SIZE = DD

Total size of the fixup data.

TOTAL DEBUG DIRECTORIES = DD

Total number of debug directories.

TOTAL DESCRIPTION SIZE = DD

Total size of the description data.

MACHINE SPECIFIC SIZE = DD

A machine-specific value.

3.0 OBJECT TABLE

The number of entries in the Object Table is supplied by the # Objects field in the PE Header. Entries in the Object Table are numbered starting from one. The Object Table immediately follows the PE Header. The code and data memory object entries are in the order chosen by the linker. The virtual addresses for objects must be assigned by the linker such that they are in ascending order and adjacent, and must be a multiple of Object Align in the PE header.

Each Object Table entry has the following format:

OBJECT NAME	
VIRTUAL SIZE	RVA
PHYSICAL SIZE	PHYSICAL OFFSET
RESERVED	RESERVED
RESERVED	OBJECT FLAGS

Figure 3. Object Table

OBJECT NAME = DB * 8

Object name. This is an eight-byte, null-padded ASCII string representing the object name.

VIRTUAL SIZE = DD

Virtual memory size. The size of the object that will be allocated when the object is loaded. Any difference between Physical Size and Virtual Size is zero filled.

RVA = DD

Relative Virtual Address. This is the virtual address that the object is currently relocated to relative to the Image Base. Each Object's virtual address space consumes a multiple of Object Align (power of 2 between 512 and 256M inclusive. The default is 64K.), and immediately follows the previous Object in the virtual address space (the virtual address space for an image must be dense).

PHYSICAL SIZE = DD

Physical file size of initialized data. The size of the initialized data in the file for the object. The physical size must be a multiple of the File Align field in the PE Header, and must be less than or equal to the Virtual Size.

PHYSICAL OFFSET = DD

Physical offset for the object's first page. This offset is relative to the beginning of the EXE file, and is aligned on a multiple of the File Align field in the PE Header. The offset is used as a seek value.

OBJECT FLAGS = DD

Flag bits for the object. The object flag bits have the following definitions:

Object Flag Bit	Definition
000000020h	Code object
000000040h	Initialized data object
000000080h	Uninitialized data object
040000000h	Object must not be cached
080000000h	Object is not pageable
100000000h	Object is shared
200000000h	Executable object
400000000h	Readable object
800000000h	Writeable object

All other bits are reserved for future use and should be set to zero.

4.0 IMAGE PAGES

The Image Pages section contains all initialized data for all objects. The seek offset for the first page in each object is specified in the Object Table and is aligned on a File Align boundary. The objects are ordered by the RVA. Every object begins on a multiple of Object Align.

5.0 EXPORTS

A typical file layout for the export information follows:

DIRECTORY TABLE
ADDRESS TABLE
NAME POINTER TABLE
ORDINAL TABLE
NAME STRINGS

Figure 4. Export File Layout

5.1 Export Directory Table

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

EXPORT FLAGS	
TIME/DATE STAMP	
MAJOR VERSION	MINOR VERSION
NAME RVA	
ORDINAL BASE	
# EAT ENTRIES	
# NAME POINTERS	
ADDRESS TABLE RVA	
NAME POINTER TABLE RVA	
ORDINAL TABLE RVA	

Figure 5. Export Directory Table Entry

EXPORT FLAGS = DD

Currently set to zero.

TIME/DATE STAMP = DD

Time/Date the export data was created.

MAJOR/MINOR VERSION = DW

A user settable major/minor version number.

NAME RVA = DD

Relative virtual address of the DLL ASCII Name. This is the address relative to the Image Base.

ORDINAL BASE = DD

First valid exported ordinal. This field specifies the starting ordinal number for the Export Address Table for this image. Normally set to 1.

 # EAT ENTRIES = DD

Indicates number of entries in the Export Address Table.

 # NAME PTRS = DD

This indicates the number of entries in the Name Pointer Table (and parallel Ordinal Table).

ADDRESS TABLE RVA = DD

Relative virtual address of the Export Address Table. This address is relative to the Image Base.

NAME TABLE RVA = DD

Relative virtual address of the Export Name Table Pointers. This address is relative to the beginning of the Image Base. This table is an array of RVA’s with #Names entries.

ORDINAL TABLE RVA = DD

Relative virtual address of Export Ordinals Table Entry. This address is relative to the beginning of the Image Base.

5.2 Export Address Table

The Export Address Table contains the address of exported entrypoints and exported data and absolutes. An ordinal number is used to index the Export Address Table. The Ordinal Base must be subtracted from the ordinal number before indexing into this table.

Export Address Table entry formats are described as follows:

EXPORTED RVA (DWORD)

Figure 6. Export Address Table Entry

EXPORTED RVA = DD

Export address. This field contains the relative virtual address of the exported entry (relative to the Image Base).

5.3 Export Name Table Pointers

The Export Name Table pointers array contains an address 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.

5.4 Export Ordinal Table

The Export Name Table Pointers and the Export Ordinal Table form two parallel arrays, separated to allow natural field alignment. The export ordinal table array contains the Export Address Table ordinal numbers associated with the named export referenced by corresponding Export Name Table Pointers.

The ordinals are 16-bits each, and already include the Ordinal Base stored in the Export Directory Table.

5.5 Export Name Table

The Export Name Table contains optional ASCII names for exported entries in the image. These tables are used with the array of Export Name Table Pointers and the array of Export Ordinals to translate a procedure name string into an ordinal number by searching for a matching name string. The ordinal number is used to locate the entry point information in the Export Address Table.

Import references by name require the Export Name Table Pointers table to be binary searched to find the matching name, then the corresponding Export Ordinal Table is known to contain the entry point ordinal number. Import references by ordinal number provide the fastest lookup because searching the name table is not required.

Each name table entry has the following format:

ASCII STRING (Zero Terminated)

Figure 7. Export Name Table Entry

ASCII STRING = DB

ASCII String. The string is case sensitive and is terminated by a null byte.

6.0 IMPORTS

A typical file layout for the import information follows:

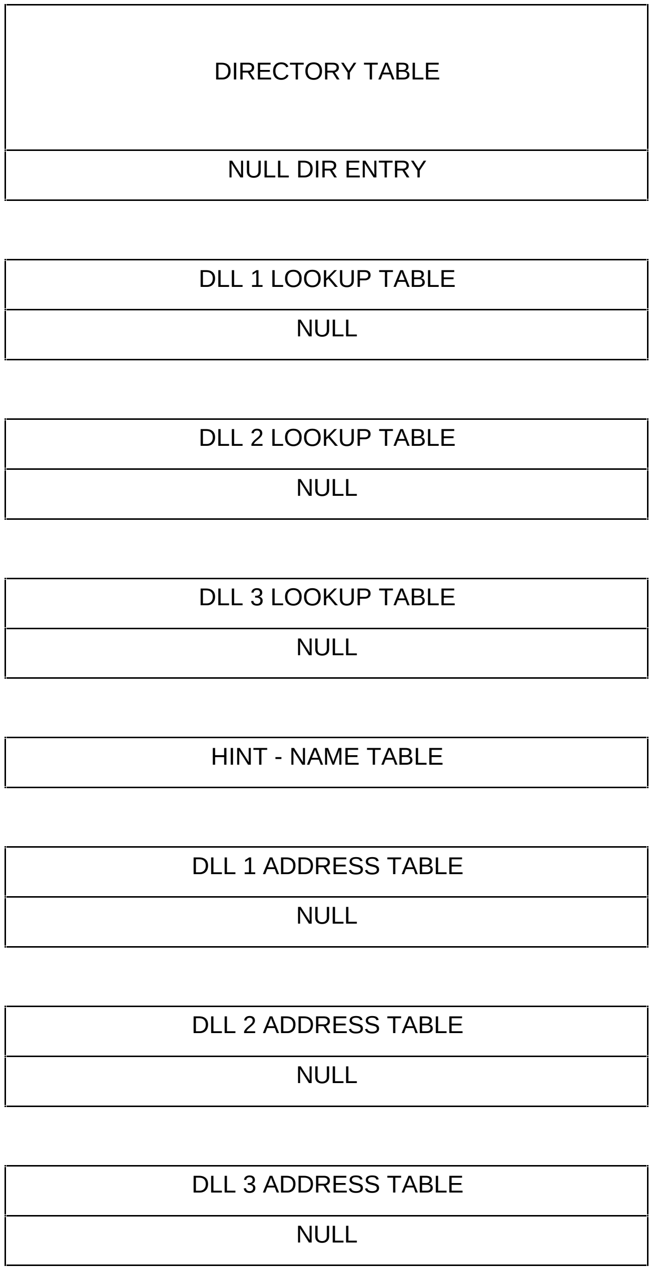

Figure 8. Import File Layout

6.1 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 this image references. The last directory entry is empty (Null) which indicates the end of the directory table.

An Import Directory Entry has the following format:

<table>
<tr><td colspan="2">IMPORT FLAGS</td></tr>
<tr><td colspan="2">TIME/DATE STAMP</td></tr>
<tr><td>MAJOR VERSION</td><td>MINOR VERSION</td></tr>
<tr><td colspan="2">NAME RVA</td></tr>
<tr><td colspan="2">IMPORT LOOKUP TABLE RVA</td></tr>
<tr><td colspan="2">IMPORT ADDRESS TABLE RVA</td></tr>
</table>

Figure 9. Import Directory Entry

IMPORT FLAGS = DD

Currently set to zero.

TIME/DATE STAMP = DD

Time/Date the import data was pre-snapped or zero if not pre-snapped.

MAJOR/MINOR VERSION = DW

The major/minor version number of the DLL being referenced.

NAME RVA = DD

Relative virtual address of the DLL ASCII Name. This is the address relative to the Image Base.

IMPORT LOOKUP TABLE RVA = DD

This field contains the address of the start of the Import Lookup Table for this image. The address is relative to the beginning of the Image Base.

IMPORT ADDRESS TABLE RVA = DD

This field contains the address of the start of the import addresses for this image. The address is relative to the beginning of the Image Base.

6.2 Import Lookup Table

The Import Lookup Table is an array of ordinal or hint/name RVA's for each DLL. The last entry is empty (Null) which indicates the end of the table.

The last element is empty.

31		0
0		ORDINAL #/ HINT-NAME TABLE RVA

Figure 10. Import Address Table Format

ORDINAL/HINT-NAME TABLE RVA = 31-bits (mask = 7fffffffh)

Ordinal Number or Name Table RVA. If the import is by ordinal, this field contains a 31-bit ordinal number. If the import is by name, this field contains a 31-bit address relative to the Image Base to the Hint-Name Table.

O = 1-bit (mask = 80000000h)	Import by ordinal flag
00000000h -	Import by name
80000000h -	Import by ordinal

6.3 Hint-Name Table

The Hint-Name Table format follows:

HINT (WORD)	ASCII STRING (Zero Terminated)	Pad

Figure 11. Import Hint-Name Table

The Pad field is used to obtain word alignment for the next entry.

HINT = DW

Hint into Export Name Table Pointers. The hint value is used to index the Export Name Table Pointers array, allowing faster by-name imports. If the hint is incorrect, then a binary search is performed on the Export Name Pointer Table.

ASCII STRING = DB

ASCII String. The string is case sensitive and is terminated by a null byte.

PAD = DB

Zero pad byte. A trailing zero pad byte appears after the trailing null byte if necessary to align the next entry on an even boundary.

The loader overwrites the Import Address Table when loading the image with the 32-bit address of the import.

6.4 Import Address Table

The Import Address Table is an array of addresses of the imported routines for each DLL. The last entry is empty (Null) which indicates the end of the table.

7.0 THREAD LOCAL STORAGE

Thread Local Storage (TLS) is a special contiguous block of data. Each thread will gets its own block upon creation of the thread.

The file layout for thread local storage follows:

DIRECTORY TABLE
TLS DATA
INDEX VARIABLE
CALLBACK ADDRESSES

Figure 12. Thread Local Storage Layout

7.1 Thread Local Storage Directory Table

The Thread Local Storage Directory Table contains address information that is used to describe the rest of TLS.

The Thread Local Storage Directory Table has the following format:

START DATA BLOCK VA
END DATA BLOCK VA
INDEX VA
CALLBACK TABLE VA

Figure 13. Thread Local Storage Directory Table

START DATA BLOCK VA = DD

Virtual address of the start of the Thread Local Storage data block.

END DATA BLOCK VA = DD

Virtual address of the end of the Thread Local Storage data block.

INDEX VA = DD

Virtual address of the index variable used to access the Thread Local Storage data block.

CALLBACK TABLE VA = DD

Virtual address of the Callback Table.

7.2 Thread Local Storage CallBack Table

The Thread Local Storage Callbacks is an array of the Virtual Address of functions to be called by the loader after thread creation and thread termination. The last entry is empty (NULL) which indicates the end of the table.

The Thread Local Storage CallBack Table has the following format:

FUNCTION1 VA (DWORD)
FUNCTION2 VA (DWORD)
....
NULL

Figure 14. Thread Local Storage CallBack Table

8.0 RESOURCES

Resources are indexed by a multiple level binary-sorted tree structure. The overall design can incorporate 2**31 levels; however, NT uses only three: the highest is Type, then Name, then Language.

A typical file layout for the resource information follows:

RESOURCE DIRECTORY
RESOURCE DATA

Figure 15. Resource File Layout

The Resource directory is made up of the following tables.

8.1 Resource Directory Table

RESOURCE FLAGS	
TIME/DATE STAMP	
MAJOR VERSION	MINOR VERSION
# NAME ENTRY	# ID ENTRY
RESOURCE DIR ENTRIES	

Figure 16. Resource Table Entry

RESOURCE FLAGS = DD

Currently set to zero.

TIME/DATE STAMP = DD

Time/Date the resource data was created by the resource compiler.

MAJOR/MINOR VERSION = DW

A user settable major/minor version number.

 # NAME ENTRY = DW

The number of name entries. This field contains the number of entries at the beginning of the array of directory entries which have actual string names associated with them.

 # ID ENTRY = DW

The number of ID integer entries. This field contains the number of 32-bit integer IDs as their names in the array of directory entries.

The resource directory is followed by a variable length array of directory entries. # Name Entry is the number of entries at the beginning of the array that have actual names associated with each entry. The entries are in ascending order, case insensitive strings. # ID Entry identifies the number of entries that have 32-bit integer IDs as their name. These entries are also sorted in ascending order.

This structure allows fast lookup by either name or number, but for any given resource entry only one form of lookup is supported, not both. This is consistent with the syntax of the .RC file and the .RES file.

The array of directory entries have the following format:

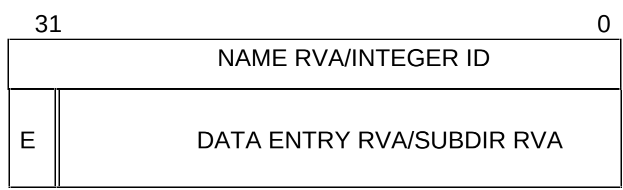

Figure 17. Resource Directory Entry

INTEGER ID = DD

ID. This field contains an integer ID field to identify a resource.

NAME RVA = DD

Name RVA address. This field contains a 31-bit address relative to the beginning of the Image Base to a Resource Directory String Entry.

E = 1-bit (mask 80000000h) Unescape bit.

This bit is zero for unescaped Resource Data Entries.

DATA RVA = 31-bits (mask 7fffffffh) Data entry address

This field contains a 31-bit address relative to the beginning of the Image Base to a Resource Data Entry.

E = 1-bit (mask 80000000h) Escape bit.

This bit is 1 for escaped Subdirectory Entry.

DATA RVA = 31-bits (mask 7fffffffh) Directory entries

This field contains a 31-bit address relative to the beginning of the Image Base to Subdirectory Entry.

Each resource directory string entry has the following format:

LENGTH	UNICODE STRING
LENGTH	UNICODE STRING

Figure 18. Resource Directory String Entry

LENGTH = DW

Length of string.

UNICODE STRING = DW

Unicode String. All of these string objects are stored together after the last Resource Directory Entry and before the first resource data object. This minimizes the impact of these variable length objects on the alignment of the fixed size directory entry objects. The length needs to be word aligned.

Each Resource Data Entry has the following format:

DATA RVA
SIZE
CODEPAGE
RESERVED

Figure 19. Resource Data Entry

DATA RVA = DD

Address of Resource Data. This field contains the 32-bit virtual address of the resource data (relative to the Image Base).

SIZE = DD

Size of Resource Data. This field contains the size of the resource data for this resource.

CODEPAGE = DD

Code page.

RESERVED = DD

Reserved. It must be zero.

Each resource data entry describes a leaf node in the resource directory tree. It contains an address which is relative to the beginning of Image Base, a size field that gives the number of bytes of data at that address, a code page that should be used when decoding code point values within the resource data. Typically for new applications the code page would be the Unicode code page.

8.2 Resource Example

The following is an example for an application that wants to use the following data as resources:

TypeId#	NameId#	Language ID	Resource Data
00000001	00000001	0	00010001
00000001	00000001	1	10010001
00000001	00000002	0	00010002
00000001	00000003	0	00010003
00000002	00000001	0	00020001
00000002	00000002	0	00020002
00000002	00000003	0	00020003
00000002	00000004	0	00020004
00000009	00000001	0	00090001
00000009	00000009	0	00090009
00000009	00000009	1	10090009
00000009	00000009	2	20090009

Then the Resource Directory in the Portable format looks like:

```
Offset              Data
0000:       00000000 00000000 00000000 00030000   (3 entries in this directory)
0010:       00000001 80000028              (TypeId #1, Subdirectory at offset 0x28)
0018:       00000002 80000050              (TypeId #2, Subdirectory at offset 0x50)
0020:       00000009 80000080              (TypeId #9, Subdirectory at offset 0x80)
0028:       00000000 00000000 00000000 00030000   (3 entries in this directory)
0038:       00000001 800000A0              (NameId #1, Subdirectory at offset 0xA0)
0040:       00000002 00000108              (NameId #2, data desc at offset 0x108)
0048:       00000003 00000118              (NameId #3, data desc at offset 0x118)
0050:       00000000 00000000 00000000 00040000   (4 entries in this directory)
0060:       00000001 00000128              (NameId #1, data desc at offset 0x128)
0068:       00000002 00000138              (NameId #2, data desc at offset 0x138)
0070:       00000003 00000148              (NameId #3, data desc at offset 0x148)
0078:       00000004 00000158              (NameId #4, data desc at offset 0x158)
0080:       00000000 00000000 00000000 00020000   (2 entries in this directory)
0090:       00000001 00000168              (NameId #1, data desc at offset 0x168)
0098:       00000009 800000C0              (NameId #9, Subdirectory at offset 0xC0)
00A0:       00000000 00000000 00000000 00020000   (2 entries in this directory)
00B0:       00000000 000000E8              (Language ID 0, data desc at offset 0xE8
00B8:       00000001 000000F8              (Language ID 1, data desc at offset 0xF8
00C0:       00000000 00000000 00000000 00030000   (3 entries in this directory)
00D0:       00000001 00000178              (Language ID 0, data desc at offset 0x178
00D8:       00000001 00000188              (Language ID 1, data desc at offset 0x188
00E0:       00000001 00000198              (Language ID 2, data desc at offset 0x198

00E8:       000001A8      (At offset 0x1A8, for TypeId #1, NameId #1, Language id #0
            00000004      (4 bytes of data)
            00000000      (codepage)
            00000000      (reserved)
00F8:       000001AC      (At offset 0x1AC, for TypeId #1, NameId #1, Language id #1
            00000004      (4 bytes of data)
            00000000      (codepage)
            00000000      (reserved)
0108:       000001B0      (At offset 0x1B0, for TypeId #1, NameId #2,
            00000004      (4 bytes of data)
            00000000      (codepage)
            00000000      (reserved)
0118:       000001B4      (At offset 0x1B4, for TypeId #1, NameId #3,
            00000004      (4 bytes of data)
            00000000      (codepage)
            00000000      (reserved)
0128:       000001B8      (At offset 0x1B8, for TypeId #2, NameId #1,
            00000004      (4 bytes of data)
            00000000      (codepage)
            00000000      (reserved)
0138:       000001BC      (At offset 0x1BC, for TypeId #2, NameId #2,
            00000004      (4 bytes of data)
            00000000      (codepage)
            00000000      (reserved)
```

```
0148:       000001C0      (At offset 0x1C0, for TypeId #2, NameId #3,
            00000004      (4 bytes of data)
            00000000      (codepage)
            00000000      (reserved)
0158:       000001C4      (At offset 0x1C4, for TypeId #2, NameId #4,
            00000004      (4 bytes of data)
            00000000      (codepage)
            00000000      (reserved)
0168:       000001C8      (At offset 0x1C8, for TypeId #9, NameId #1,
            00000004      (4 bytes of data)
            00000000      (codepage)
            00000000      (reserved)
0178:       000001CC      (At offset 0x1CC, for TypeId #9, NameId #9, Language id #0
            00000004      (4 bytes of data)
            00000000      (codepage)
            00000000      (reserved)
0188:       000001D0      (At offset 0x1D0, for TypeId #9, NameId #9, Language id #1
            00000004      (4 bytes of data)
            00000000      (codepage)
            00000000      (reserved)
0198:       000001D4      (At offset 0x1D4, for TypeId #9, NameId #9, Language id #2
            00000004      (4 bytes of data)
            00000000      (codepage)
            00000000      (reserved)
```

And the data for the resources will look like:

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

9.0 FIXUP TABLE

The Fixup Table contains entries for all fixups in the image. The Total Fixup Data Size in the PE Header is the number of bytes in the Fixup Table. The Fixup Table is broken into blocks of fixups. Each block represents the fixups for a 4K page.

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

9.1 Fixup Block

Fixup blocks have the following format:

PAGE RVA	
BLOCK SIZE	
TYPE/OFFSET	TYPE/OFFSET
TYPE/OFFSET	TYPE/OFFSET

Figure 20. Fixup Block Format

To apply a fixup, a delta needs to be calculated. The 32-bit delta is the difference between the preferred base, and the base where the image is actually loaded. If the image is loaded at its preferred base, the delta would be zero, and thus the fixups would not have to be applied. Each block must start on a DWORD boundary. The Absolute fixup type can be used to pad a block.

PAGE RVA = DD

Page RVA. The image base plus the page RVA is added to each offset to create the virtual address of where the fixup needs to be applied.

BLOCK SIZE = DD

Number of bytes in the fixup block. This includes the Page RVA and Size fields.

Type/Offset is defined as:

15 11	0
TYPE	OFFSET

Figure 21. Fixup Record Format

Type = 4-bit fixup type. This value has the following definitions:

0h - Absolute. This is a NOP. The fixup is skipped.

1h - High. Add the high 16-bits of the delta to the 16-bit field at Offset. The 16-bit field represents the high value of a 32-bit word.

2h - Low. Add the low 16-bits of the delta to the 16-bit field at Offset. The 16-bit field represents the low half value of a 32-bit word. This fixup will only be emitted for a RISC machine when the image Object Align isn't the default of 64K.

3h - Highlow. Apply the 32-bit delta to the 32-bit field at Offset.

4h - Highadjust. This fixup requires a full 32-bit value. The high 16-bits is located at Offset, and the low 16-bits is located in the next Offset array element (this array element is included in the Size field). The two need to be combined into a signed variable. Add the 32-bit delta. Then add 0x8000 and store the high 16-bits of the signed variable to the 16-bit field at Offset.

5h - Mipsjmpaddr.

All other values are reserved.

10.0 DEBUG INFORMATION

The debug information is defined by the debugger and is not controlled by the portable EXE format or linker. The only data defined by the portable EXE format is the Debug Directory Table.

10.1 Debug Directory

The Debug Directory Table consists of one or more entries that have the following format:

<table>
<tr><td colspan="2">DEBUG FLAGS</td></tr>
<tr><td colspan="2">TIME/DATE STAMP</td></tr>
<tr><td>MAJOR VERSION</td><td>MINOR VERSION</td></tr>
<tr><td colspan="2">DEBUG TYPE</td></tr>
<tr><td colspan="2">DATA SIZE</td></tr>
<tr><td colspan="2">DATA RVA</td></tr>
<tr><td colspan="2">DATA SEEK</td></tr>
</table>

Figure 22. Debug Directory Entry

DEBUG FLAGS = DD

Set to zero.

TIME/DATE STAMP = DD

Time/Date the debug data was created.

MAJOR/MINOR VERSION = DW

Version stamp. This stamp can be used to determine the version of the debug data.

DEBUG TYPE = DD

Format type. To support multiple debuggers, this field determines the format of the debug information. This value has the following definitions:

0001h - Image contains COFF symbolics.

0001h - Image contains Microsoft symbol and type information.

0001h - Image contains FPO symbolics.

DATA SIZE = DD

The number of bytes in the debug data. This is the size of the actual debug data and does not include the debug directory.

DATA RVA = DD

The relative virtual address of the debug data. This address is relative to the beginning of the Image Base.

DATA SEEK = DD

The seek value from the beginning of the file to the debug data.

If the image contains more than one type of debug information, then the next debug directory will immediately follow the first debug directory.

II

Microsoft Symbol and Type Information

TIS Formats Specification for Windows™, Version 1.0
Microsoft Symbol and Type Information

This document describes Microsoft Symbol and Type Information, a debugging information format from Microsoft Corporation for the 32-bit Windows environment.

The TIS Committee formed a debug subcommittee to evaluate the widely available formats with the objective of adopting one as the TIS standard. After studying many different formats, the committee adopted Microsoft Symbol and Type Information as a standard debugging information format for 32-bit Windows environments.

The TIS Committee worked with Microsoft to make the standard extensible. The remainder of the information contained herein is provided by Microsoft, and no other technical modifications were recommended by the TIS committee.

Microsoft Symbol and Type Information

Table of Contents

1. Symbol and Type Information

This document describes the format and meaning of Microsoft symbol and type debugging information. The information is contained within two tables emitted by the language processor into the object file. Each table is treated as a stream of variable length records. The first table is called $$SYMBOLS and describes the symbols in the object file. The record for each symbol contains the symbol name, the symbol address and other information needed to describe the symbol. The second table is called $$TYPES and contains information about symbol typing. There are fields in the records contained in $$SYMBOLS that index into the records contained in $$TYPES. Records in $$TYPES can also index into the records contained in the $$TYPES table.

The records for $$SYMBOLS and $$TYPES are accumulated by the linker and are written into the executable file. There is a third table of symbol information for each object file that is generated by the linker and written into the executable file called the PUBLICS table. This table contains symbol records for each public symbol definition encountered in the object file.

Field sizes and arrangement in $$SYMBOLS and $$TYPES are arranged to maintain "natural alignment" to improve performance. Natural alignment indicates that a field begins on an address that is divisible by the size of the field. For example, a four byte (long) value begins on an address that is evenly divisible by four. Some architectures, such as the MIPS R4000, impose a severe penalty for loading data that is not in natural alignment. Even for Intel386™ and Intel486™ processors, there is a significant improvement when processing data that is in natural alignment.

Compilers that emit Symbol and Type OMF (object module formats) according to this specification indicate so by placing a signature of 0x00000001 at the beginning of the $$SYMBOLS and $$TYPES tables.

In all structure descriptions and value enumerations, all values not specified in this document are reserved for future use. All values should be referenced by the symbolic descriptions.

The CVPACK utility must be run on a linked executable file before the Microsoft debugger can process the file. This utility removes duplicate symbol and type information and rewrites the remaining information in a format optimized for processing by the debugger. CVPACK will recognize old Symbol and Type OMF and rewrite it to this format during packing.

1.1. Logical Segments

When the linker emits address information about a symbol, it is done in *segment:offset* format. The *segment* is a logical segment index assigned by the linker and the *offset* is the offset from the beginning of the logical segment. The physical address is assigned by the operating system when the program is loaded.

For PE-formatted executables, the *segment* field is interpreted as the PE section number.

1.2. Lexical Scope Linkage

The model of a program envisioned by this document is that programs have nested scopes. The outermost scope is module scope which encompasses all of the symbols not defined within any inner (lexical) scope. Symbols and types defined at one scoping level are visible to all scopes nested within it. Symbols and types defined at module scope are visible to all inner scopes.

The next level of scoping is "function" scope, which in turn contains lexical blocks (including other functions scopes) that can be further nested. Nested lexical scopes are opened by a procedure, method, thunk, with, or block start symbol. They are closed by the matching block-end symbol.

In general, symbol searching within a module's symbol table is performed in the following manner. The lexical scope that contains the current program address is searched for the symbol. If the symbol is not found within that scope, the enclosing lexical scope is searched. This search is repeated outward until the symbol is found or the module scope is searched unsuccessfully. Note that lexical scopes at the same depth level are not searched. As an optimization for the debugger, symbols that open a lexical scope have fields that contain offsets from the beginning of the symbols for the module, which point to the parent of the scope, the next lexical scope that is at the same scoping level, and the S_END symbol that closes this lexical scope.

The *pParent, pNext* and *pEnd* fields described below are filled in by the CVPACK utility and should be emitted as zeroes by the language processor.

Field	**Linkage**
pParent	Used in local procedures, global procedures, thunk start, with start, and block start symbols. If the scope is not enclosed by another lexical scope, then *pParent* is zero. Otherwise, the parent of this scope is the symbol within this module that opens the outer scope that encloses this scope but encloses no other scope that encloses this scope. The *pParent* field contains the offset from the beginning of the module's symbol table of the symbol that opens the enclosing lexical scope.
pNext	Used in start search local procedures, global procedures, and thunk start symbols. The *pNext* field, along with the start search symbol, defines a group of lexically scoped symbols within a symbol table that is contained within a code segment or PE section. For each segment or section represented in the symbol table, there is a start search symbol that contains the offset from the start of the symbols for this module to the first procedure or thunk contained in the segment. Each outermost lexical scope symbol has a next field containing the next outermost scope symbol contained in the segment. The last outermost scope in the symbol table for each segment has a next field of zero.
pEnd	This field is defined for local procedures, global procedures, thunk, block, and with symbols. The end field contains the offset from the start of the symbols for this module to the matching block end symbol that terminates the lexical scope.

1.3. Numeric Leaves

When the symbol or type processor knows that a numeric leaf is next in the symbol or type record, the next two bytes of the symbol or type string are examined. If the value of these two bytes is less than LF_NUMERIC (0x8000), then the two bytes contain the actual numeric value. If the value is greater than or equal to LF_NUMERIC (0x8000), then the numeric data follows the two-byte leaf index in the format specified by the numeric leaf index. It is the responsibility of routines reading numeric fields to handle the potential non alignment of the data fields. See Section 4 entitled Numeric Leaves for details.

1.4. Types Indices

All Symbol and Type OMF records which reference records in the $$TYPES table must use valid non-zero type indices. For public symbols a type index of 0x0000 (T_NOTYPE) is permitted.

Since many types (relating to hardware and language primitives) are common, type index values less than 0x1000 (CV_FIRST_NONPRIM) are reserved for a set of predefined primitive types. A list of predefined types and their indices are defined in this document in Section 5. Type indices of 0x1000 and higher are used to index into the set of non-primitive type definitions in the module's $$TYPES segment. Thus 0x1000 is the first type, 0x1001 the second, and so on. Non-primitive type indices must be sequential and cannot contain gaps in the numbering.

1.5. $$SYMBOLS and $$TYPES Definitions

$$TYPES Definition

OMF

Type information appears in OMF TYPDEF format as LEDATA records that contribute to the special $$TYPES debug segment. A SEGDEF or SEGDEF32 record for this segment must be produced in each module that contains Symbol and Type OMF type information and have the attributes:

Name:	$$TYPES
Combine type:	private
Class:	DEBTYP

The first four bytes of the $$TYPES table is used as a signature to specify the version of the Symbol and Type OMF contained in the $$TYPES segment. If the first two bytes of the $$TYPES segment are not 0x0000, the signature is invalid and the version is assumed to be that emitted for an earlier version of the Microsoft CodeView debugger (version 3.x and earlier). If the signature is 0x00000001, the Symbol and Type OMF has been written to conform to the later version of the Microsoft debugger (version 4.0) specification. All other values for the signature are reserved. The CVPACK utility rewrites previous versions of the Symbol and Type OMF to conform to this specification. The signatures of the $$TYPES and $$SYMBOLS tables must agree.

COFF

Type information appears in a COFF (common object file format) as initialized data sections. The attributes for the sections are:

NAME:	.debug$T
Attribute:	Read Only, Discardable, Initialized Data

As with OMF, the first four bytes in the types section must contain a valid signature and agree with the signature in the symbol table.

$$SYMBOLS Definition

OMF

Symbol information appears in OMF TYPDEF format as LEDATA records that contribute to the special $$SYMBOLS debug segment. A SEGDEF or SEGDEF32 record for this segment must be produced in each module that contains Symbol and Type OMF symbol information and have these attributes:

Name:	$$SYMBOLS
Combine type:	private
Class:	DEBSYM

The first four bytes of the $$SYMBOLS segment is used as a signature to specify the version of the Symbol and Type OMF contained in the $$SYMBOLS segment. If the first two bytes of the $$SYMBOLS segment are not 0x0000, the signature is invalid and the version is assumed to be that emitted for an earlier version of the Microsoft CodeView debugger, version 3.x and earlier. If the signature is 0x00000001, the Symbol and Type OMF has been written to conform to the version 4.0 specification of the Microsoft CodeView debugger. All other values for the signature are reserved. The CVPACK utility rewrites previous versions of the Symbol and Type OMF to conform to this specification. The signatures for the $$TYPES and $$SYMBOLS tables must agree.

COFF

Symbol information appears in separate sections. The attributes of the section are:

Name:	.debug$S
Attributes:	Read Only, Discardable, Initialized Data

There may be multiple symbol sections in an object. The first symbol section to appear in the object file must NOT be associated with a comdat section and must contain a valid signature. If a comdat section is present in the object then the symbol information for that comdat should be in a separate symbol section associated with the text comdat section. Symbol sections associated with comdats must not contain a signature.

2. Symbols

2.1. General

Format of Symbol Records

Data in the $$SYMBOLS segment is a stream of variable length records with the general format:

2	2	*
length	index	data...

length Length of record, excluding the length field.
index Type of symbol.
data Data specific to each symbol format.

The symbol records are described below. Numbers above the fields indicate the length in bytes, and * means variable length for that field.

Symbol indices are broken into five ranges. The first range is for symbols whose format does not change with the compilation model of the program or the target machine. These include register symbols, user-defined type symbols, and so on. The second range of symbols are those that contain 16:16 segmented addresses. The third symbol range is for symbols that contain 16:32 addresses. Note that for flat model programs, the segment is replaced with the section number for PE format .exe files. The fourth symbol range is for symbols that are specific to the MIPS architecture/compiler. The fifth range is for Microsoft CodeView optimization.

The symbol records are formatted such that most fields fall into natural alignment if the symbol length field is placed on a long word boundary. For all symbols, the variable length data is at the end of the symbol structure. Note specifically that fields that contain data in potentially nonaligned numeric fields must either pay the load penalty or first do a byte wise copy of the data to a memory that is in natural alignment. Refer to Section 4 for details about numeric leaves.

16:16 compilers do not have to emit padding bytes between symbols to maintain natural alignment. The CVPACK utility places the symbols into the executable files in natural alignment and zero pads the symbol to force alignment. The length of each symbol is adjusted to account for the pad bytes. 16:32 compilers must align symbols on a long word boundary.

Provisions for enabling future implementation of register tracking and a stack machine to perform computation on symbol addresses are provided in the symbols. When the symbol processor is examining a symbol, the length field of the symbol is compared with the offset of the byte following the end of the symbol name field. If these are the same, there is no stack machine code at the end of the symbol. If the length and offset are different, the byte following the end of the symbol name is examined. If the byte is zero, there is no stack machine code following the symbol. If the byte is not zero, then the byte indexes into the list of stack machine implementations and styles of register tracking. If stack machine code is present, the address field of the symbol becomes the initial value of the stack machine. Microsoft does not currently emit or process stack machine code or register tracking information. The opcodes and operation of the stack machine have not been defined.

Symbol Indices

0x0001	S_COMPILE	Compile flags symbol
0x0002	S_REGISTER	Register variable
0x0003	S_CONSTANT	Constant symbol
0x0004	S_UDT	User-defined Type
0x0005	S_SSEARCH	Start search
0x0006	S_END	End block, procedure, with, or thunk
0x0007	S_SKIP	Skip - Reserve symbol space
0x0008	S_CVRESERVE	Reserved for internal use by the Microsoft debugger
0x0009	S_OBJNAME	Specify name of object file
0x000a	S_ENDARG	Specify end of arguments in function symbols
0x000b	S_COBOLUDT	Microfocus COBOL user-defined type
0x000c	S_MANYREG	Many register symbol
0x000d	S_RETURN	Function return description
0x000e	S_ENTRYTHIS	Description of **this** pointer at entry
0x0100	S_BPREL16	BP relative 16:16
0x0101	S_LDATA16	Local data 16:16
0x0102	S_GDATA16	Global data 16:16
0x0103	S_PUB16	Public symbol 16:16
0x0104	S_LPROC16	Local procedure start 16:16
0x0105	S_GPROC16	Global procedure start 16:16
0x0106	S_THUNK16	Thunk start 16:16
0x0107	S_BLOCK16	Block start 16:16
0x0108	S_WITH16	With start 16:16
0x0109	S_LABEL16	Code label 16:16
0x010a	S_CEXMODEL16	Change execution model 16:16
0x010b	S_VFTPATH16	Virtual function table path descriptor 16:16
0x010c	S_REGREL16	Specify 16:16 offset relative to arbitrary register
0x0200	S_BPREL32	BP relative 16:32
0x0201	S_LDATA32	Local data 16:32
0x0202	S_GDATA32	Global data 16:32
0x0203	S_PUB32	Public symbol 16:32
0x0204	S_LPROC32	Local procedure start 16:32
0x0205	S_GPROC32	Global procedure start 16:32
0x0206	S_THUNK32	Thunk start 16:32
0x0207	S_BLOCK32	Block start 16:32
0x020b	S_VFTPATH32	Virtual function table path descriptor 16:32
0x020c	S_REGREL32	16:32 offset relative to arbitrary register
0x020d	S_LTHREAD32	Local Thread Storage data
0x020e	S_GTHREAD32	Global Thread Storage data
0x0300	S_LPROCMIPS	Local procedure start MIPS
0x0301	S_GPROCMIPS	Global procedure start MIPS
0x0400	S_PROCREF	Reference to a procedure
0x0401	S_DATAREF	Reference to data
0x0402	S_ALIGN	Page align symbols

2.2. Non-modal Symbols

(0x0001) Compile Flag

This symbol communicates with Microsoft debugger compile-time information, such as the language and version number of the language processor, the ambient model for code and data, and the target processor, on a per-module basis.

2	2	1	3	*
length	S_COMPILE	machine	flags	version

machine Enumeration specifying target processor. Values not specified in the following list are reserved:

0x00	Intel 8080
0x01	Intel 8086
0x02	Intel 80286
0x03	Intel 80386
0x04	Intel 80486
0x05	Intel Pentium
0x10	MIPS R4000
0x11	Reserved for future MIPS processor
0x12	Reserved for future MIPS processor
0x20	MC68000
0x21	MC68010
0x22	MC68020
0x23	MC68030
0x24	MC68040
0x30	DEC Alpha

flags Flags showing compile-time options, as follows:

Language	:8	
PCodePresent	:1	
FloatPrecision	:2	
FloatPackage	:2	
AmbientData	:3	
AmbientCode	:3	
Mode32	:1	Compiled for 32-bit addresses
Reserved		:4

Language enumerations:

0	C
1	C++
2	Fortran
3	Masm
4	Pascal
5	Basic
6	COBOL
7 - 255	Reserved

Ambient code and data memory model enumeration:

0	Near
1	Far
2	Huge
3 - 7	Reserved

Floating-package enumeration:

0	Hardware processor (80x87 for Intel 80x86 processors)
1	Emulator
2	Altmath
3	Reserved

The FloatPrecision flag is set to 1 if the compiler follows the ANSI C floating-point precision rules. This is specified for Microsoft C compilers by setting the -Op option.

version Length-prefixed string specifying language processor version. Language processors can place additional data in version string if desired.

(0x0002) Register

This symbol record describes a symbol that has been placed in a register. Provisions for enabling future implementation tracking of a symbol into and out of registers is provided in this symbol. When the symbol processor is examining a register symbol, the length field of the symbol is compared with the offset of the byte following the symbol name field. If these are the same, there is no register tracking information. If the length and offset are different, the byte following the end of the symbol name is examined. If the byte is zero, there is no register tracking information following the symbol. If the byte is not zero, then the byte is the index into the list of stack machine implementations and styles of register tracking. Microsoft does not currently emit or process register-tracking information.

2	2	2	2	*	*
length	S_REGISTER	@type	register	name	tracking

@type Type of symbol.

register Enumeration of the registers in which the symbol value is stored. This field is treated as two bytes. The high order byte specifies the register in which the high order part of the value is stored. The low byte specifies the register for the low order part of the value. If the value is not stored in two registers then high order register field contains the enumeration value for no register. For register enumeration values, see Section 6. The register index enumeration is specific to the processor model for the module.

name Length-prefixed name of the symbol stored in the register.

tracking Register-tracking information. Format unspecified.

(0x0003) Constant

This record is used to output constants and C enumerations. If used to output an enumeration, then the type index refers to the containing enum.

2	2	2	*	*
length	S_CONSTANT	@type	value	name

@type Type of symbol or containing enum.
value Numeric leaf containing the value of symbol.
name Length-prefixed name of symbol.

(0x0004) User-defined Type

This specifies a C typedef or user-defined type, such as classes, structures, unions, or enums.

2	2	2	*
length	S_UDT	@type	name

@type Type of symbol.
name Length-prefixed name of the user defined type.

(0x0005) Start Search

These records are always the first symbol records in a module's $$SYMBOL section. There is one Start Search symbol for each segment (PE section) to which the module contributes code. Each Start Search symbol contains the segment (PE section) number and $$SYMBOL offset of the record of the outermost lexical scope in this module that physically appears first in the specified segment of the load image. This referenced symbol is the symbol used to initiate context searches within this module. The Start Search symbols are inserted into the $$SYMBOLS table by the CVPACK utility and must not be emitted by the language processor.

2	2	4	2
length	S_SSEARCH	sym off	segment

sym off $$SYMBOL offset of the procedure or thunk record for this module that has the lowest offset for the specified segment. See Section 1.2 on lexical scope linking.
segment Segment (PE section) to which this Start Search refers.

(0x0006) End of Block

Closes the scope of the nearest preceding Block Start, Global Procedure Start, Local Procedure Start, With Start, or Thunk Start definition.

2	2
length	S_END

(0x0007) Skip Record

This record reserves symbol space for incremental compilers. The compiler can reserve a dead space in the OMF for future expansions due to an incremental build. This symbol and the associated reserved space is removed by the CVPACK utility.

2	2	*
length	S_SKIP	skip data

skip data Unused data. Use the length field that precedes every symbol record to skip this record.

(0x0008) Microsoft Debugger Internal

This symbol is used internally by the Microsoft debugger and never appears in the executable file. Its format is unspecified.

(0x0009) Object File Name

This symbol specifies the name of the object file for this module.

2	2	4	*
length	S_OBJNAME	signature	name

signature Signature for the Microsoft symbol and type information contained in this module. If the object file contains precompiled types, then the signature will be checked against the signature in the LF_PRECOMP type record contained in the $$TYPES table for the user of the precompiled types. The signature check is used to detect recompilation of the supplier of the precompiled types without recompilation of all of the users of the precompiled types. The method for computing the signature is unspecified, but should be sufficiently robust to detect failures to recompile.

name Length-prefixed name of the object file without any path information prepended to the name.

(0x000a) End of Arguments

This symbol specifies the end of symbol records used in formal arguments for a function. Use of this symbol is optional for OMF and required for MIPS-compiled code. In OMF format, the end of arguments can also be deduced from the fact that arguments for a function have a positive offset from the frame pointer.

2	2
length	S_ENDARG

(0x000b) COBOL User-defined Type

This record is used to define a user-defined type for the Microfocus COBOL compiler. This record cannot be moved into the global symbol table by the CVPACK utility.

2	2	2	*
length	S_COBOLUDT	@type	name

@type Type of symbol.
name Length-prefixed name of the user-defined type.

(0x000c) Many Registers

This record is used to specify that a symbol is stored in a set of registers.

2	2	2	1	1 * count	*
length	S_MANYREG	@type	count	reglist	name

@type Type index of the symbol.
count Count of the register enumerations that follow.
reglist List of registers in which the symbol is stored. The registers are listed high order register first.
name Name of the symbol.

(0x000d) Function Return

This symbol is used to describe how a function is called, how the return value, if any, is returned, and how the stack is cleaned up.

2	2	2	1	*
length	S_RETURN	flags	style	data

flags Flags for function call:

cstyle	:1	push varargs right to left, if true
rsclean	:1	returnee stack cleanup, if true
unused	:14	

style Function return style:

0x00	void return
0x01	return value is in the registers specified in *data*
0x02	indirect caller-allocated near
0x03	indirect caller-allocated far
0x04	indirect returnee-allocated near
0x05	indirect returnee-allocated far

data Data required by function return style.
If *style* is 0x01, then data is the following format.

1	1 * count
count	reglist

count Count of the number of registers.
reglist Registers (high order first) containing the value.

(0x000e) this at Method Entry

This record is used to describe the **this** pointer at entry to a method. It is really a wrapper for another symbol that describes the **this** pointer.

2	2	*
length	S_ENTRYTHIS	symbol

symbol Full symbol, including length and symbol type fields, which describes the **this** pointer.

2.3. Symbols for 16:16 Segmented Architectures

(0x0100) BP Relative 16:16

This symbol specifies symbols that are allocated on the stack for a procedure. For C and C++, these include the actual function parameters and the local nonstatic variables of functions.

2	2	2	2	*
length	S_BPREL16	offset	@type	name

offset Signed offset relative to BP. If *offset* is 0, the symbol was assigned to a register or never instantiated by the optimizer and cannot be evaluated because its location is unknown.
@type Type of symbol.
name Length-prefixed name of symbol.

(0x0101) Local Data 16:16

These symbols are used for data that is not exported from a module. In C and C++, symbols that are declared static are emitted as Local Data symbols. Symbols that are emitted as Local Data cannot be moved by the CVPACK utility into the global symbol table for the executable file.

2	2	2	2	2	*
length	S_LDATA16	offset	segment	@type	name

offset Offset portion of symbol address.
segment Segment portion of symbol address.
@type Type index of symbol.
name Length-prefixed name of symbol.

(0x0102) Global Data Symbol 16:16

This symbol record has the same format as the Local Data 16:16 except that the record type is S_GDATA16. For C and C++, symbols that are not specifically declared static are emitted as Global Data Symbols and can be compacted by the CVPACK utility into the global symbol table.

(0x0103) Public Symbol 16:16

This symbol has the same format as the Local Data 16:16 symbol. Its use is reserved for symbols in the public table that is emitted by the linker into the Symbol and Type OMF portion of the executable file. Current linkers (version 5.30 and later) emit the public symbols in the S_PUB16 format. Previous linkers emitted the public symbols in the following obsolete format:

2	2	2	*
offset	segment	@type	name

offset	Offset portion of symbol address.
segment	Segment portion of symbol address.
@type	Type index of symbol (can be zero).
name	Length-prefixed name of symbol.

For public symbols emitted in the obsolete format, the CVPACK utility rewrites them to the S_PUB16 format before compacting them into the global publics table. For more information about the format of the Symbol and Type OMF as written by the linker and CVPACK utilities, see Section 7 on executable file format.

(0x0104) Local Start 16:16

This symbol record defines local (file static) procedure definitions. For C and C++, functions that are declared static to a module are emitted as Local Procedure symbols. Functions not specifically declared static are emitted as Global Procedures (see below).

2	2	4	4	4	2	2	
length	symbol	pParent	pEnd	pNext	proc length	debug start	->

2	2	2	2	1	*
debug end	offset	segment	@proctype	flags	name

symbol	S_LPROC16 or S_GPROC16.
pParent	See the section on lexical scope linking.
pEnd	See the section on lexical scope linking.
pNext	See the section on lexical scope linking.
proc length	Length in bytes of this procedure.
debug start	Offset in bytes from the start of the procedure to the point where the stack frame has been set up. Frame and register variables can be viewed at this point.

debug end	Offset in bytes from the start of the procedure to the point where the procedure is ready to return and has calculated its return value, if any. Frame and register variables can still be viewed.
offset	Offset portion of the procedure address.
segment	Segment portion of the procedure address.
@proctype	Type index of the procedure type record.
flags	Procedure flags: fpo :1 True if function has frame pointer omitted. interrupt :1 True if function is interrupt routine. return :1 True if function performs far return. never :1 True if function never returns. unused :4
name	Length-prefixed name of procedure.

(0x0105) Global Procedure Start 16:16

This symbol is used for procedures that are not specifically declared static to a module. The format is the same as the Local Procedure Start 16:16 symbol (see above.)

(0x0106) Thunk Start 16:16

This symbol is used to specify any piece of code that exists outside of a procedure. The lexical scope started by the Thunk Start symbol is closed by a matching End record.

2	2	4	4	2	2	2	
length	S_THUNK16	pParent	pEnd	pNext	offset	segment	->

2	1	*	*
length	ordinal	name	variant

pParent	See the section on lexical scope linking.
pEnd	See the section on lexical scope linking.
pNext	See the section on lexical scope linking.
offset	Offset portion of the thunk address.
segment	Segment portion of the thunk address.
ordinal	Ordinal specifying the type of thunk: 0 NOTYPE 1 ADJUSTOR 2 VCALL 3 PCODE
length	Length in bytes of this thunk.
name	Length-prefixed name of thunk.
variant	Variant field, depending on the value of *ordinal*. If *ordinal* is NOTYPE, there will be no variant field. If *ordinal* is ADJUSTOR, the variant field will be a two-byte signed value specifying the delta to be added to the **this** pointer, followed by the name of the target function. If the *ordinal* is VCALL, then the variant field will be a 2-byte signed displacement into the virtual table. Note that because of the variable length name, the data in the variant field will not be in natural alignment. If *ordinal* is PCODE, the variant is the *segment:offset* of the pcode entry point.

(0x0107) Block Start 16:16

This symbol specifies the start of an inner block of lexically scoped symbols. The lexical scope is terminated by a matching S_END symbol.

2	2	4	4	2	2	2	*
length	S_BLOCK16	pParent	pEnd	length	offset	segment	name

pParent	See the section on lexical scope linking.
pEnd	See the section on lexical scope linking.
length	Length in bytes of the scope of this block.
offset	Offset portion of the segmented procedure address.
segment	Segment portion of the segmented procedure address.
name	Length-prefixed name of block.

(0x0108) With Start 16:16

This symbol describes the lexical scope of the Pascal **with** statement.

2	2	4	4	2	2	2	*
length	S_WITH16	pParent	pEnd	length	offset	segment	expr

pParent	See the section on lexical scope linking.
pEnd	See the section on lexical scope linking.
length	Length in bytes of the scope of the **with** block.
offset	Offset portion of the block start address.
segment	Segment portion of the block start address.
expr	Length-prefixed ASCII string of the expression used in the **with** statement, which is evaluated at run time.

(0x0109) Code Label 16:16

2	2	2	2	1	*
length	S_LABEL16	offset	segment	flags	name

offset Offset portion of the code label address.

segment Segment portion of the code label address.

flags Label flags. This uses the same flag definition as in the S_LPROC16 symbol record, as follows:

fpo	:1	True if function has frame pointer omitted.
interrupt	:1	True if function is interrupt routine.
return	:1	True if function performs far return.
never	:1	True if function never returns.
unused	:4	

name Length-prefixed name of code label.

(0x010a) Change Execution Model 16:16

This record is used to notify the debugger that, starting at the given code offset and until the address specified by the next Change Execution Model record, the execution model is of the specified type. The native execution model is assumed in the absence of Change Execution Model records.

2	2	2	2	2	*
length	S_CEXMODEL16	offset	segment	model	variant

offset	Offset portion of start of the block.	
segment	Segment portion of the start of block.	
model	The execution model.	
	0x00	Not executable code (e.g., a table)
	0x01	Compiler generated jump table
	0x02	Padding for data
	0x03 - 0x1f	Reserved for specific noncode types.
	0x20	Native model (no processor specified)
	0x21	Microfocus COBOL
	0x22	Code padding for alignment
	0x23	Code
	0x24 - 0x3F	Reserved
	0x40	Pcode
variant	Variable data dependent upon the execution model field. If the variant record contains segment or offset information, then the CVPACK utility and the Microsoft debugger must be modified to process the segment information.	

The variant field for 0x40 (C7 Pcode) data has the following format:

2	2
Fcn Header	SPI

Fcn Header	Offset of the Pcode procedure's Function Header.
SPI	Offset of the Pcode segment's Segment Pcode Information.

Both addresses are in the specified code segment.

The variant field for 0x21 (Microfocus COBOL) has the following format:

2	2
subtype	flag

subtype	COBOL execution model subtype.	
	0	Do not stop execution until next model record
	1	pfm
	2	False call - continue tracing
	3	External call

The other models do not have variant fields.

(0x010b) Virtual Function Table Path 16:16

This record is used to describe the base class path for the virtual function table descriptor.

2	2	2	2	2	2
length	S_VFTPATH16	offset	segment	@root	@path

offset	Offset portion of start of the virtual function table.
segment	Segment portion of the virtual function table.
@root	The type index of the class at the root of the path.
@path	Type index of the record describing the base class path from the root to the leaf class for the virtual function table.

(0x010c) Register Relative 16:16

This symbol specifies symbols that are allocated relative to a register.

2	2	2	2	2	*
length	S_REGREL16	offset	register	@type	name

offset	Signed offset relative to register.
register	Register enumeration for symbol base. Note that the register field can specify a register pair, such as ES:BX.
@type	Type of symbol.
name	Length-prefixed name of symbol.

2.4. Symbols for 16:32 Segmented Architectures

(0x0200) BP Relative 16:32

This symbol specifies symbols that are allocated on the stack for a procedure. For C and C++, these include the actual function parameters and the local non-static variables of functions.

2	2	4	2	*
length	S_BPREL32	offset	@type	name

offset	Signed offset relative to BP. If *offset* is 0, then the symbol was assigned to a register or never instantiated by the optimizer and cannot be evaluated because its location is unknown.
@type	Type of symbol.
name	Length-prefixed name of symbol.

(0x0201) Local Data 16:32

These symbols are used for data that is not exported from a module. In C and C++, symbols that are declared static are emitted as Local Data symbols. Symbols that are emitted as Local Data cannot be moved by the CVPACK utility into the global symbol table for the executable file.

2	2	4	2	2	*
length	S_LDATA32	offset	segment	@type	name

offset	Offset portion of symbol address.
segment	Segment portion of symbol address.
@type	Type index of symbol.
name	Length-prefixed name of symbol.

(0x0202) Global Data Symbol 16:32

This symbol record has the same format as the Local Data 16:32 except that the record type is S_GDATA32. For C and C++, symbols that are not specifically declared static are emitted as Global Data Symbols and can be compacted by the CVPACK utility into the global symbol table.

(0x0203) Public 16:32

This symbol has the same format as the Local Data 16:32 symbol. Its use is reserved to symbols in the publics table emitted by the linker into the Symbol and Type OMF portion of the executable file.

(0x0204) Local Procedure Start 16:32

This symbol record defines local (file static) procedure definition. For C and C++, functions that are declared static to a module are emitted as Local Procedure symbols. Functions not specifically declared static are emitted as Global Procedures (see below.)

2	2	4	4	4	4	4
length	symbol	pParent	pEnd	pNext	proc length	debug start

->

4	4	2	2	1	*
debug end	offset	segment	@proctype	flags	name

symbol	S_LPROC32 or S_GPROC32.
pParent	See the section on lexical scope linking.
pEnd	See the section on lexical scope linking.
pNext	See the section on lexical scope linking.
proc length	Length in bytes of this procedure.
debug start	Offset in bytes from the start of the procedure to the point where the stack frame has been set up. Parameter and frame variables can be viewed at this point.

debug end	Offset in bytes from the start of the procedure to the point where the procedure is ready to return and has calculated its return value, if any. Frame and register variables can still be viewed.		
offset	Offset portion of the procedure address.		
segment	Segment portion of the procedure address.		
@proctype	Type of the procedure type record.		
flags	Procedure flags:		
	fpo	:1	True if function has frame pointer omitted.
	interrupt	:1	True if function is interrupt routine.
	return	:1	True if function performs far return.
	never	:1	True if function never returns.
	unused	:4	
name	Length-prefixed name of procedure.		

(0x0205) Global Procedure Start 16:32

This symbol is used for procedures that are not specifically declared static to a module. The format is the same as the Local Procedure Start 16:32 symbol (see above.)

(0x0206) Thunk Start 16:32

This record is used to specify any piece of code that exists outside a procedure. It is followed by an End record. The thunk record is intended for small code fragments and a two byte length field is sufficient for its intended purpose.

2	2	4	4	4	4	2	
length	S_THUNK32	pParent	pEnd	pNext	offset	segment	->

2	1	*	*
thunk length	ordinal	name	variant

pParent	See the section on lexical scope linking.	
pEnd	See the section on lexical scope linking.	
pNext	See the section on lexical scope linking.	
offset	Offset portion of the thunk address.	
segment	Segment portion of the thunk address.	
thunk length	Length in bytes of this thunk.	
ordinal	Ordinal specifying the type of thunk, as follows:	
	0	NOTYPE
	1	ADJUSTOR
	2	VCALL
	3	PCODE
name	Length-prefixed name of thunk.	
variant	Variant field, depending on value of *ordinal*. If *ordinal* is NOTYPE, there is no variant field. If *ordinal* is ADJUSTOR, the variant field is a two-byte signed value specifying the delta to be added to the **this** pointer, followed by the length-prefixed name of the target function. If *ordinal* is VCALL, then the variant field is a two-byte signed displacement into the virtual table. If *ordinal* is PCODE, the variant is the *segment:offset* of the pcode entry point.	

(0x0207) Block Start 16:32

This symbol specifies the start of an inner block of lexically scoped symbols. The lexical scope is terminated by a matching S_END symbol.

2	2	4	4	4	4	2	*
length	S_BLOCK32	pParent	pEnd	length	offset	segment	name

pParent See the section on lexical scope linking.
pEnd See the section on lexical scope linking.
length Length in bytes of the scope of this block.
offset Offset portion of the segmented procedure address.
segment Segment portion of the segmented procedure address.
name Length-prefixed name of the block.

(0x0208) With Start 16:32

2	2	4	4	4	4	2	*
length	S_WITH32	pParent	pEnd	length	offset	segment	expr

pParent See the section on lexical scope linking.
pEnd See the section on lexical scope linking.
length Length in bytes of the scope of the with block.
offset Offset portion of the segmented address of the start of the block.
segment Segment portion of the segmented address of the start of the block.
expr Length-prefixed ASCII string, evaluated at run time, of the expression used in the **with** statement.

(0x0209) Code Label 16:32

2	2	4	2	1	*
length	S_LABEL32	offset	segment	flags	name

offset Offset portion of the segmented address of the start of the block.
segment Segment portion of the segmented address of the start of the block.
flags Label flags. This uses the same flag definition as in the S_LPROC16 symbol record, as follows:

fpo	:1	True if function has frame pointer omitted.
interrupt	:1	True if function is interrupt routine.
return	:1	True if function performs far return.
never	:1	True if function never returns.
unused	:4	

name Length-prefixed name of label.

(0x020a) Change Execution Model 16:32

This record is used to notify the debugger that, starting at the given code offset and until the address specified by the next Change Execution Model record, the execution model is of the specified type. The native execution model is assumed in the absence of Change Execution Model records.

2	2	4	2	2	*
length	S_CEXMODEL32	offset	segment	model	variant

offset	Offset portion of start of block.	
segment	Segment portion of the start of block.	
model	Execution model, as follows:	
	0x00	Not executable code (e.g., a table)
	0x01	Compiler generated jump table
	0x02	Padding for data
	0x03 - 0x1f	Reserved for specific noncode types.
	0x20	Native model (no processor specified)
	0x21	Microfocus COBOL (unused in 16:32)
	0x22	Code padding for alignment
	0x23	Code
	0x24 - 0x3f	Reserved
	0x40	Pcode (Reserved)
variant	Variable data dependent upon the execution model field. If the variant record contains segment or offset information, then the CVPACK utility and the Microsoft debugger must be modified to process the segment information.	

The other models do not have variant fields.

(0x020b) Virtual Function Table Path 16:32

This record is used to describe the base class path for the virtual function table descriptor.

2	2	4	2	2	2
length	S_VFTPATH32	offset	segment	@root	@path

offset	Offset portion of start of the virtual function table.
segment	Segment portion of the virtual function table.
@root	The type index of the class at the root of the path.
@path	Type index of the record describing the base class path from the root to the leaf class for the virtual function table.

(0x020c) Register Relative 16:32

This symbol specifies symbols that are allocated relative to a register.

2	2	4	2	2	*
length	S_REGREL32	offset	register	@type	name

offset	Signed offset relative to register.
register	Register enumerations on which the symbol is based. Note that the register field can specify a pair of registers, such as ES:EBX.
@type	Type of symbol.
name	Length-prefixed name of symbol.

(0x020d) Local Thread Storage 16:32

These symbols are used for data declared with the **__***thread* storage attribute that is not exported from a module. In C and C++, **__***thread* symbols that are declared static are emitted as Local Thread Storage 16:32 symbols. Symbols that are emitted as Local Thread Storage 16:32 cannot be moved by the CVPACK utility into the global symbol table for the executable file.

2	2	4	2	2	*
length	S_LTHREAD32	offset	segment	@type	name

offset	Offset into thread local storage.
segment	Segment of thread local storage.
@type	Type index.
name	Length-prefixed name.

(0x020e) Global Thread Storage 16:32

This symbol record has the same format as the Local Thread Storage 16:32 except that the symbol type is S_GTHREAD32. For C and C++, **__***thread* symbols that are not specifically declared static are emitted as Global Thread Storage 16:32 symbols and can be compacted by the CVPACK utility into the global symbol table.

2.5. Symbols for MIPS Architectures

(0x0300) Local Procedure Start MIPS

The symbol records define local (file static) procedures. For C and C++, functions that are declared static to a module are emitted as Local Procedure symbols.

2	2	4	4	4	4	4	
length	symbol	pParent	pEnd	pNext	length	debug start	->

4	4	4	4	4	
debug end	int save mask	fp save mask	int save offset	fp save offset	->

4	2	2	1	1	*
offset	seg	@proctype	retreg	frame pointer reg	name

symbol	S_LPROCMIPS or S_GPROCMIPS.
pParent	See the section on lexical scope linking.
pEnd	See the section on lexical scope linking.
pNext	See the section on lexical scope linking.
length	Length in bytes of this procedure.
debug start	Offset in bytes from the start of the procedure to the point where the stack frame has been set up. Parameter and frame variables can be viewed at this point.
debug end	Offset in bytes from the start of the procedure to the point where the procedure is ready to return and has calculated its return value, if any. Frame and register variables can still be viewed. If the procedure has multiple exits, this field is zero.
int save mask	Integer register save mask.
fp save mask	Floating-point register save mask.
int save offset	Offset from sp to the integer register save area.
fp save offset	Offset from sp to the floating point register save area.
offset	Offset portion of the address of the start of the procedure.
segment	Segment portion of the address of the start of the procedure.
@proctype	Type index of the procedure type record.
retreg	Index of the register that contains the return address. If this register is 31 and the integer register save mask indicates that the register has been saved, then the return address is in the integer register save area.
framepointer	Frame pointer register if not zero.
name	Length-prefixed name of procedure.

(0x0301) Global Procedure Start MIPS

This symbol is used for procedures that are not specifically declared static to a module. The format is the same as the Local Procedure Start 16:32 symbol (see above.)

2.6. Symbols for CVPACK Optimization

(0x0400) Procedure Reference

This symbol is inserted into the global and static symbol tables to reference a procedure. It is used so that the symbol procedure can be found in the hashed search of the global or static symbol table. Otherwise, procedures could be found only by searching the symbol table for every module.

2	2	4	4	2
length	S_PROCREF	checksum	offset	module

checksum Checksum of the referenced symbol name. The checksum used is the one specified in the header of the sstGlobalSym or sstStaticSym subsections. See Section 7.4 for more details on the subsection headers.

offset Offset of the procedure symbol record from the beginning of the $$SYMBOL table for the module.

module Index of the module that contains this procedure record.

(0x0401) Data Reference

This symbol is inserted into the global and static symbol tables to reference data. It is used so that the symbol procedure can be found in the hashed search of the global or static symbol table. Otherwise, data symbols could be found only by searching the symbol table for every module.

2	2	4	4	2
length	S_DATAREF	checksum	offset	module

checksum Checksum of the referenced symbol name.

offset Offset of the procedure symbol record from the beginning of the $$SYMBOL table for the module.

module Index of the module that contains this procedure record.

(0x0402) Symbol Page Alignment

This symbol is inserted by the CVPACK utility to pad symbol space so that the next symbol will not cross a page boundary.

2	2	*
length	S_ALIGN	pad

pad Unused data. Use the length field that precedes every symbol record to skip this record. The pad bytes must be zero. For sstGlobalSym and sstGlobalPub, the length of the pad field must be at least the sizeof (long). There must be an S_Align symbol at the end of these tables with a pad field containing 0xffffffff. The sstStaticSym table does not have this requirement.

3. Types Definition Segment ($$TYPES)

A $$TYPES segment may appear in linkable modules. It provides descriptions of the types of symbols found in the $$PUBLICS and $$SYMBOLS debug section for the module.

3.1. Type Record

A type record has the following format:

2	*
length	type string

length Length in bytes of the following type string. This count does not include the length field.

3.2. Type String

A type string is a series of consecutive leaf structures and has the following format:

2	*	2	*		2	*
leaf	data	leaf	data	...	leaf	data

leaf LF_... index, as described below.
data Data specified to each leaf type.

No LF_... index can have a value of 0x0000. The leaf indices are separated into four ranges according to the use of the type record. The first range is for the type records that are directly referenced in symbols. The second range is for type records that are not referenced by symbols, but instead are referenced by other type records. All type records must have a starting leaf index in these first two ranges.

The third range of leaf indices is used to build complex lists, such as the field list of a class type record. No type record can begin with one of the leaf indices in this range.

The fourth ranges of type indices are used to represent numeric data in a symbol or type records. These leaf indices are greater than 0x8000. At the point that the type or symbol processor is expecting a numeric field, the next two bytes in the type record are examined. If the value is less than 0x8000, then the two bytes contain the numeric value. If the value is greater than 0x8000, then the data follows the leaf index in a format specified by the leaf index. See Section 4 for a detailed description of numeric leaf indices.

Because of the method used to maintain natural alignment in complex lists, no leaf index can have a value greater than or equal to 0xf000. Also, no leaf index can have a value such that the least significant 8 bits of the value is greater than or equal to 0xf0.

Leaf indices for type records that can be referenced from symbols are the following:

0x0001	LF_MODIFIER
0x0002	LF_POINTER
0x0003	LF_ARRAY
0x0004	LF_CLASS
0x0005	LF_STRUCTURE
0x0006	LF_UNION
0x0007	LF_ENUM
0x0008	LF_PROCEDURE
0x0009	LF_MFUNCTION
0x000a	LF_VTSHAPE
0x000b	LF_COBOL0
0x000c	LF_COBOL1
0x000d	LF_BARRAY
0x000e	LF_LABEL
0x000f	LF_NULL
0x0010	LF_NOTTRAN
0x0011	LF_DIMARRAY
0x0012	LF_VFTPATH
0x0013	LF_PRECOMP
0x0014	LF_ENDPRECOMP
0x0015	LF_OEM
0x0016	Reserved

Leaf indices for type records that can be referenced from other type records are the following:

0x0200	LF_SKIP
0x0201	LF_ARGLIST
0x0202	LF_DEFARG
0x0203	LF_LIST
0x0204	LF_FIELDLIST
0x0205	LF_DERIVED
0x0206	LF_BITFIELD
0x0207	LF_METHODLIST
0x0208	LF_DIMCONU
0x0209	LF_DIMCONLU
0x020a	LF_DIMVARU
0x020b	LF_DIMVARLU
0x020c	LF_REFSYM

Leaf indices for fields of complex lists are the following:

0x0400	LF_BCLASS
0x0401	LF_VBCLASS
0x0402	LF_IVBCLASS
0x0403	LF_ENUMERATE
0x0404	LF_FRIENDFCN
0x0405	LF_INDEX
0x0406	LF_MEMBER
0x0407	LF_STMEMBER
0x0408	LF_METHOD
0x0409	LF_NESTTYPE
0x040a	LF_VFUNCTAB
0x040b	LF_FRIENDCLS
0x040c	LF_ONEMETHOD

0x040d	LF_VFUNCOFF

Leaf indices for numeric fields of symbols and type records are the following:

0x8000	LF_NUMERIC
0x8000	LF_CHAR
0x8001	LF_SHORT
0x8002	LF_USHORT
0x8003	LF_LONG
0x8004	LF_ULONG
0x8005	LF_REAL32
0x8006	LF_REAL64
0x8007	LF_REAL80
0x8008	LF_REAL128
0x8009	LF_QUADWORD
0x800a	LF_UQUADWORD
0x800b	LF_REAL48
0x800c	LF_COMPLEX32
0x800d	LF_COMPLEX64
0x800e	LF_COMPLEX80
0x800f	LF_COMPLEX128
0x8010	LF_VARSTRING

0xf0	LF_PAD0
0xf1	LF_PAD1
0xf2	LF_PAD2
0xf3	LF_PAD3
0xf4	LF_PAD4
0xf5	LF_PAD5
0xf6	LF_PAD6
0xf7	LF_PAD7
0xf8	LF_PAD8
0xf9	LF_PAD9
0xfa	LF_PAD10
0xfb	LF_PAD11
0xfc	LF_PAD12
0xfc	LF_PAD13
0xfe	LF_PAD14
0xff	LF_PAD15

Member Attribute Field

Several of the type records below reference a field attribute bit field. This bit field has the following format:

access	:2	Specifies the access protection of the item
	0	No access protection
	1	Private
	2	Protected
	3	Public
mprop	:3	Specifies the properties for methods
	0	Vanilla method
	1	Virtual method

	2	Static method
	3	Friend method
	4	Introducing virtual method
	5	Pure virtual method
	6	Pure introducing virtual method
	7	Reserved
pseudo	:1	True if the method is never instantiated by the compiler
noinherit	:1	True if the class cannot be inherited
noconstruct	:1	True if the class cannot be constructed
reserved	:8	

3.3. Leaf Indices Referenced from Symbols

(0x0001) Type Modifier

This record is used to indicate the **const**, **volatile** and **unaligned** properties for any particular type.

2	2	2
LF_MODIFIER	attribute	@index

attribute	const	:1	**const** attribute
	volatile	:1	**volatile** attribute
	unaligned	:1	**unaligned** attribute
	reserved	:13	
@index	type index of the modified type.		

(0x0002) Pointer

This record is the generic pointer type record. It supports the C++ reference type, pointer to data member, and pointer to method. It also conveys **const** and **volatile** pointer information.

2	2	2	*
LF_POINTER	attribute	@type	variant

attribute		**Consists of five bit fields:**
ptrtype	**:5**	**Ordinal specifying mode of pointer**
	0	Near
	1	Far
	2	Huge
	3	Based on segment
	4	Based on value
	5	Based on segment of value
	6	Based on address of symbol
	7	Based on segment of symbol address
	8	Based on type
	9	Based on self

	10	Near 32-bit pointer
	11	Far 32-bit pointer
	12-31	Reserved
ptrmode	**:3**	**Ordinal specifying pointer mode**
	0	Pointer
	1	Reference
	2	Pointer to data member
	3	Pointer to method
	4-7	Reserved
isflat32	**:1**	True if 16:32 pointer
volatile	**:1**	True if pointer is volatile
const	**:1**	True if pointer is const
unaligned	**:1**	True if pointer is unaligned
unused	**:4**	Unused and reserved
@type		Type index of object pointed to
variant		variant portion of the record, depending upon the pointer type
		based on segment - Segment value
		based on type- Index of type followed by length-prefixed name
		based on self - Nothing
		based on symbol - Copy of symbol record including length field
		pointer to data member - Union specifying pointer to data member
		pointer to method - Union specifying pointer to method

The union specifying the pointer to data member has the following format:

2	2
@class	format

class Type index of containing class.

format
- 0 16:16 data for class with no virtual functions or virtual bases
- 1 16:16 data for class with virtual functions
- 2 16:16 data for class with virtual bases
- 3 16:32 data for classes with or without virtual functions and no virtual bases
- 4 16:32 data for class with virtual bases
- 5 16:16 near method non-virtual bases with single address point
- 6 16:16 near method non-virtual bases with multiple address points
- 7 16:16 near method with virtual bases
- 8 16:16 far method non-virtual bases with single address point
- 9 16:16 far method non-virtual bases with multiple address points
- 10 16:16 far method with virtual bases
- 11 16:32 method non-virtual bases with single address point
- 12 16:32 method non-virtual bases with multiple address points
- 13 16:32 method with virtual bases

The pointer to data member and pointer to method have the following formats in memory. In the following descriptions of the format and value of the NULL pointer, * means any value.

- (00) 16:16 pointer to data member for a class with no virtual functions or bases.

2
mdisp

mdisp Displacement to data. NULL is 0xffff.

- (01) 16:16 pointer to data member for a class with virtual functions.

2
mdisp

mdisp Displacement to data. NULL is 0.

- (02) 16:16 pointer to data member for a class with virtual bases.

2	2	2
mdisp	pdisp	vdisp

mdisp	Displacement to data.
pdisp	**this** pointer displacement to virtual base table pointer.
vdisp	Displacement within virtual base table. NULL value is (,,0xffff).

- (03) 16:32 near pointer to data member for a class with and without virtual functions and no virtual bases.

4
mdisp

mdisp Displacement to data. NULL is 0x80000000.

- (04) 16:32 near pointer to data member for a class with virtual bases.

4	4	4
mdisp	pdisp	vdisp

mdisp	Displacement to data.
pdisp	**this** pointer displacement to virtual base table pointer.
vdisp	Displacement within virtual base table. NULL value is (,,0xffffffff).

- (05) 16:16 pointer to near member function for a class with no virtual functions or bases and a single address point.

2
off

off	Near address of method. NULL is 0.

- (06) 16:32 pointer to near member function for a class with no virtual bases with multiple address points.

2	2
off	disp

off	Offset of function.
disp	Displacement of address point. NULL is (0,*).

- (07) 16:16 pointer to near member function for a class with virtual bases.

2	2	2	2
off	mdisp	pdisp	vdisp

off	Offset of function.
mdisp	Displacement to data.
pdisp	**this** pointer displacement to virtual base table pointer.
vdisp	Displacement within virtual base table. NULL value is (0,*,*,*).

- (08) 16:16 pointer to far member function for a class with no virtual bases and a single address point.

2	2
off	seg

off	Offset of function.
disp	Displacement of address point. NULL is (0:0).

- (09) 16:16 pointer to far member function for a class with no virtual bases and multiple address points.

2	2	2
off	seg	disp

off	Offset of function.
seg	Segment of function.
disp	Displacement of address point. NULL is (0:0,*).

- (10) 16:16 pointer to far member function for a class with virtual bases.

2	2	2	2	2
off	seg	mdisp	pdisp	vdisp

off	Offset of function.
seg	Segment of function.
mdisp	Displacement to data.
pdisp	**this** pointer displacement to virtual base table pointer.
vdisp	Displacement within virtual base table. NULL value is (0,*,*,*).

- (11) 16:32 pointer to member function for a class with no virtual bases and a single address point.

4
off

off	Offset of function. NULL is 0L.

- (12) 16:32 pointer to member function for a class with no virtual bases and multiple address points.

4	4
off	disp

off	Offset of function.
disp	Displacement of address point. NULL is (0L:0L).

- (13) 16:32 pointer to member function for a class with virtual bases.

4	4	4	4
off	mdisp	pdisp	vdisp

off	Offset of function.
mdisp	Displacement to data.
pdisp	**this** pointer displacement to virtual base table pointer.
vdisp	Displacement within virtual base table. NULL value is (0L,*,*,*).

(0x0003) Simple Array

The format for a simple array is as follows:

2	2	2	*	*
LF_ARRAY	@elemtype	@idxtype	length	name

@elemtype	Type index of each array element.
@idxtype	Type index of indexing variable.
length	Length of array in bytes.
name	Length-prefixed name of array.

(0x0004) Classes

The format for classes is as follows:

2	2	2	2	2	2	*	*
leaf	count	@field	property	@dList	@vshape	length	name

leaf LF_CLASS or LF_STRUCTURE.

count Number of elements in the class or structure. This count includes direct, virtual, and indirect virtual bases, and methods including overloads, data members, static data members, friends, and so on.

@field Type index of the field list for this class.

property Property bit field

packed	:1	Structure is packed
ctor	:1	Class has constructors and/or destructors
overops	:1	Class has overloaded operators
isnested	:1	Class is a nested class
cnested	:1	Class contains nested classes
opassign	:1	Class has overloaded assignment
opcast	:1	Class has casting methods
fwdref	:1	Class/structure is a forward (incomplete) reference
scoped	:1	This is a scoped definition
reserved	:8	

@dList Type index of the derivation list. This is output by the compiler as 0x0000 and is filled in by the CVPACK utility to a LF_DERIVED record containing the type indices of those classes which immediately inherit the current class. A zero index indicates that no derivation information is available. An LF_NULL index indicates that the class is not inherited by other classes.

@vshape Type index of the virtual function table shape descriptor.

length Numeric leaf specifying size in bytes of the structure.

name Length-prefixed name this type.

(0x0005) Structures

Structures have the same format as classes. Structure type records are used exclusively by the C compiler. The C++ compiler emits both class and structure records depending upon the declaration.

(0x0006) Unions

The format for unions is as follows:

2	2	2	2	*	*
LF_UNION	count	@field	property	length	name

count	Number of fields in the union.
@field	Type index of field list.
property	Property bit field.
length	Numeric leaf specifying size in bytes of the union.
name	Length-prefixed name of union.

(0x0007) Enumeration

The format for an enum is as follows:

2	2	2	2	2	*
LF_ENUM	count	@type	@fList	property	name

count	Number of enumerations.
@type	Underlying type of enum.
@field	Type index of field list.
property	Property bit field.
name	Length-prefixed name of enum.

(0x0008) Procedure

The format for a procedure is as follows:

2	2	1	1	2	2
LF_PROCEDURE	@rvtype	call	reserved	#parms	@arglist

@rvtype Type index of the value returned by the procedure.

call Calling convention of the procedure, as follows:

0	Near C (arguments pushed right to left, caller pops arguments)
1	Far C.
2	Near Pascal (arguments pushed left to right, callee pops arguments)
3	Far Pascal
4	Near fastcall
5	Far fastcall
6	Reserved
7	Near stdcall
8	Far stdcall
9	Near syscall
10	Far syscall
11	This call

	12	MIPS call
	13	Generic
	14-255	Reserved
#parms	Number of parameters.	
@arglist	Type index of argument list type record.	

(0x0009) Member Function

The format for a member function is as follows:

2	2	2	2	1	1	
LF_MFUNCTION	@rvtype	@class	@this	call	res	->

2	2	4
#parms	@arglist	thisadjust

@rvtype	Type index of the value returned by the procedure.
@class	Type index of the containing class of the function.
@this	Type index of the **this** parameter of the member function. A type of void indicates that the member function is static and has no **this** parameter.
call	Calling convention of the procedure. See Procedure description.
res	Reserved. Must be emitted as zero.
#parms	Number of parameters. This count does not include the **this** parameter.
arglist	List of parameter specifiers. This list does not include the **this** parameter.
thisadjust	Logical **this** adjuster for the method. Whenever a class element is referenced via the **this** pointer, *thisadjust* will be added to the resultant offset before referencing the element.

(0x000a) Virtual Function Table Shape

This record describes the format of a virtual function table. This record is accessed via the vfunctabptr in the member list of the class which introduces the virtual function. The vfunctabptr is defined either by the LF_VFUNCTAB or LF_VFUNCOFF member record. If LF_VFUNCTAB record is used, then vfunctabptr is at the address point of the class. If LF_VFUNCOFF record is used, then vfunctabptr is at the specified offset from the class address point. The underlying type of the pointer is a VTShape type record. This record describes how to interpret the memory at the location pointed to by the virtual function table pointer.

2	2	4 bits
LF_VTSHAPE	count	descriptor
		repeated

count	Number of descriptors.

descriptor	A four-bit ordinal describing the entry in the virtual table	
	0	Near
	1	Far
	2	Thin
	3	Address point displacement to outermost class. This is at entry[-1] from table address
	4	Far pointer to metaclass descriptor. This is at entry[-2] from table address
	5	Near32
	6	Far32
	7 - 15	Reserved

(0x000b) COBOL0

This record has been reserved for the Microfocus COBOL compiler.

2	2	*
LF_COBOL0	@parent	data

@parent	Type index of the parent type.
data	Data.

(0x000c) COBOL1

This record has been reserved for the Microfocus COBOL compiler.

2	*
LF_COBOL1	data

data Data.

(0x000d) Basic Array

2	2
LF_BARRAY	@ type

type Type of each element in the array.

(0x000e) Label

This is used for assembler labels where there is no typing information about the label.

2	2
LF_LABEL	mode

mode	Addressing mode of the label, as follows:	
	0	Near label
	4	Far label

(0x000f) Null

This is used when the symbol requires a type record but the data content is null.

2
LF_NULL

(0x0010) Not Translated

This is used when CVPACK encounters a type record that has no equivalent in the Microsoft symbol information format.

2
LF_NOTTRANS

(0x0011) Multiply Dimensioned Array

This record is used to describe a multiply dimensioned array.

2	2	2	*
LF_DIMARRAY	@utype	@diminfo	name

@utype	Underlying type of the array.
@diminfo	Index of the type record containing the dimension information.
name	Length-prefixed name of the array.

(0x0012) Path to Virtual Function Table

This record is used to describe the path to the virtual function table.

2	2	2 * count
LF_VFTPATH	count	bases

count	Count or number of bases in the path to the virtual function table.
bases	Type indices of the base classes in the path.

(0x0013) Reference Precompiled Types

This record specifies that the type records are included from the precompiled types contained in another module in the executable. A module that contains this type record is considered to be a user of the precompiled types. When emitting to a COFF object, the section name should be .debug$P rather than .debug$T. All other attributes should be the same.

2	2	2	4	*
LF_PRECOMP	start	count	signature	name

start	Starting type index that is included. This number must correspond to the current type index in the current module.
count	Count or number of type indices included. After including the precompiled types, the type index must be *start* + *count*.
signature	Signature for the precompiled types being referenced by this module. The signature will be checked against the signature in the S_OBJNAME symbol record and the LF_ENDPRECOMP type record contained in the $$TYPES table of the creator of the precompiled types. The signature check is used to detect recompilation of the supplier of the precompiled types without recompilation of all of the users of the precompiled types. The method for computing the signature is unspecified. It should be sufficiently robust to detect failures to recompile.
name	Name of the module containing the precompiled types. This name must match the module name in the S_OBJNAME symbol emitted by the compiler for the object file containing the precompiled types.

(0x0014) End of Precompiled Types

This record specifies that the preceding type records in this module can be referenced by another module in the executable. A module that contains this type record is considered to be the creator of the precompiled types. The subsection index for the $$TYPES segment for a precompiled types creator is emitted as sstPreComp instead of sstTypes, so that the CVPACK utility can pack the precompiled types creators before the users. Precompiled types must be emitted as the first type records within the $$TYPES segment and must be self-contained. That is, they cannot reference a type record with an index greater than or equal to the type index of the LF_ENDPRECOMP type record.

2	4
LF_ENDPRECOMP	signature

signature	Signature of the precompiled types. The signatures in the S_OBJNAME symbol record, the LF_PRECOMP type record and this signature must match.

(0x0015) OEM Generic Type

This record is supplied to allow third party compiler vendors to emit debug OMF information in an arbitrary format and still allow the CVPACK utility to process the record. CVPACK processes this record by performing a left to right depth first recursive pack of the records specified by *indices* below. The remainder of the data is copied without alteration.

2	2	2	2	2 * *count*	*
LF_OEM	OEM	recOEM	count	indices	data

OEM	Microsoft-assigned OEM identifier.
recOEM	OEM-assigned record identifier. These record identifiers are unique per assigned OEM.
count	Number of type indices that follow.
indices	Type indices.
data	Remainder of type record.

(0x0016) Reserved

3.4. Type Records Referenced from Type Records

(0x0200) Skip

This is used by incremental compilers to reserve space for future indexes.

2	2	*
LF_SKIP	index	pad

index In processing $$TYPES, the index counter is advanced to index count, skipping all intermediate indices. This is the next valid index.

pad Space reserved for incremental compilations. Note that this record is removed by the link/pack utility, so there is no requirement for maintaining natural alignment for this record.

(0x0201) Argument List

2	2	*
LF_ARGLIST	argcount	indices

argcount Count or number of indices in list.

indices List of type indices for describing the formal parameters for a function or method.

(0x0202) Default Argument

2	2	*
LF_DEFARG	@index	expression

index Type index of resulting expression.

expression Length-prefixed string of supplied default.

(0x0203) Arbitrary List

2	*
LF_LIST	data

data A list of leaves with a format defined by the leaf that indexes the list. This leaf type has been superseded by more specific list types and its use is not recommended.

(0x0204) Field List

A field list contains the descriptors of the fields of a structure, class, union, or enumeration. The field list is composed of zero or more subfields. Because of the requirement for natural alignment, there may be padding between elements of the field list. As a program walks down the field list, the address of the next subfield is calculated by adding the length of the previous field to the address of the previous field. The byte at the new address is examined and if it is greater than 0xf0, the low four bits are extracted and added to the address to find the address of the next subfield. These padding fields are not included in the count field of the class, structure, union, or enumeration type records. If the field list is broken into two or more pieces by the compiler, then the last field of each piece is an LF_INDEX with the type being the index of the continuation record. The LF_INDEX and LF_PADx fields of the field list are not included in field list count specified in the class, structure, union, or enumeration record. See Section 3.5 for field list elements.

2	*	*	...	*	2	*
leaf	data	pad	...	pad	leaf	data

(0x0205) Derived Classes

This type record specifies all of the classes that are directly derived from the class that references this type record.

2	2	*
LF_DERIVED	count	@type

count	Number of types in the list.
@type	Type indices of the classes that directly inherit from the class that references this type record.

(0x0206) Bit Fields

Bit fields are represented by an entry in the field list that indexes a bit field type definition.

2	1	1	2
LF_BITFIELD	length	position	@type

length	Length in bits of the object.
position	Starting position (from bit 0) of the object in the word.
@type	Type index of the field.

(0x0207) Method List

2	2	2	4
LF_MLIST	attribute	@type	vtab offset
			optional
		repeated	

attribute	Attribute of the member function.
@type	Type index of the procedure record for this occurrence of the function.
vtab offset	Present only when property attribute is introducing virtual (optional). Offset in vtable of the class which contains the pointer to the function.

Once a method has been found in this list, its symbol is found by qualifying the method name with its class (T::name) and then searching the symbol table for a symbol by that name with the correct type index. Note that the number of repeats is determined by the subleaf of the field list that references this LF_MLIST record.

(0x0208) Dimensioned Array with Constant Upper Bound

This record is used to describe a dimensioned array with default lower bound and constant upper bound. The default lower bound is language specific.

2	2	2	**s***rank
LF_DIMCONU	rank	@index	bound

rank	Number of dimensions.
@index	Type of index.
bound	Constants for the upper bound of each dimension of the array. Each constant is of the size **s** specified by @index.

(0x0209) Dimensioned Array with Constant Lower and Upper Bounds

This record is used to describe a dimensioned array with constant lower and upper bound.

2	2	2	2***s***rank
LF_DIMCONLU	rank	@index	bound

rank	Number of dimensions.
@index	Type of index.
bound	Pairs of constants for the lower and upper bound of each dimension of the array. Each constant is of the size **s** specified by @index. The ordering is lower bound followed by upper bound for each dimension.

(0x020a) Dimensioned Array with Variable Upper Bound

This record is used to describe a dimensioned array with default lower bound and variable upper bound. The default lower bound is language specific.

2	2	2	2*rank
LF_DIMVARU	rank	@index	@var

rank	Number of dimensions.
@index	Type of index.
@var	Array of type index of LF_REFSYM record describing the variable upper bound. If one dimension of the array is variable, then all dimensions must be described using LF_REFSYM records.

(0x020b) Dimensioned Array with Variable Lower and Upper Bounds

This record is used to describe a dimensioned array with variable lower and upper bound.

2	2	2	2*rank
LF_DIMVARLU	rank	@index	var

rank	Number of dimensions.
@index	Type of index.
@var	Array of type indices of LF_REFSYM records describing the variable lower and upper bounds. If one dimension of the array is variable, then all dimensions must be described using LF_REFSYM records. The order is lower bound followed by upper bound for each dimension.

(0x020c) Referenced Symbol

This record is used to describe a symbol that is referenced by a type record. The record is defined because type records cannot reference symbols or locations in the $$SYMBOLS table and because global symbol compaction will move symbols.

2	*
LF_REFSYM	sym

sym	Copy of the referenced symbol including the length field.

3.5. Subfields of Complex Lists

Currently, the only complex list that uses the following leaf indices is the field list of a structure, class, union, or enumeration.

(0x0400) Real Base Class

This leaf specifies a real base class. If a class inherits real base classes, the corresponding Real Base Class records will precede all other member records in the field list of that class. Base class records are emitted in left-to-right declaration order for real bases.

	2	2	*
LF_BCLASS	@type	attribute	offset

@type	Index to type record of the class. The class name can be obtained from this record.
attribute	Member attribute bit field.
offset	Offset of subobject that represents the base class within the structure.

(0x0401) Direct Virtual Base Class

This leaf specifies directly inherited virtual base class. If a class directly inherits virtual base classes, the corresponding Direct Virtual BaseClass records will follow all Real Base Class member records and precede all other member records in the field list of that class. Direct Virtual Base class records are emitted in bottommost left-to-right inheritance order for directly inherited virtual bases.

2	2	2	2	*	*
type	@btype	@vbtype	attribute	vbpoff	vboff

type	LF_VBCLASS.
@btype	Index to type record of the direct or indirect virtual base class. The class name can be obtained from this record.
@vbptype	Type index of the virtual base pointer for this base
attribute	Member attribute bit field.
vbpoff	Numeric leaf specifying the offset of the virtual base pointer from the address point of the class for this virtual base.
vboff	Numeric leaf specifying the index into the virtual base displacement table of the entry that contains the displacement of the virtual base. The displacement is relative to the address point of the class plus *vbpoff*.

(0x0402) Indirect Virtual Base Class

This leaf specifies indirectly inherited virtual base class. If a class indirectly inherits virtual base classes, the corresponding Indirect Virtual Base Class records will follow all Real Base Class and Direct Virtual Base Class member records and precede all other member records in the field list of that class. Direct Virtual Base class records are emitted in bottommost left-to-right inheritance order for virtual bases.

2	2	2	2	*	*
type	@btype	@vbtype	attribute	vbpoff	vboff

type	LF_VBCLASS or LF_IVBCLASS.
@btype	Index to type record of the direct or indirect virtual base class. The class name can be obtained from this record.
@vbptype	Type index of the virtual base pointer for this base.
attribute	Member attribute bit field.
vbpoff	Numeric leaf specifying the offset of the virtual base pointer from the address point of the class for this virtual base.
vboff	Numeric leaf specifying the index into the virtual base displacement table of the entry that contains the displacement of the virtual base. The displacement is relative to the address point of the class plus *vbpoff*.

(0x0403) Enumeration Name and Value

This leaf specifies the name and value of an enumerate within an enumeration.

2	2	*	*
LF_ENUMERATE	attribute	value	name

attribute	Member attribute bit field.
value	Numeric leaf specifying the value of the enumeration.
name	Length-prefixed name of the member field.

(0x0404) Friend Function

This leaf specifies a friend function.

2	2	*
LF_FRIENDFCN	@type	name

@type	Index to type record of the friend function.
name	Length-prefixed name of friend function.

(0x0405) Index To Another Type Record

2	2
LF_INDEX	@index

index	Type index. This field is emitted by the compiler when a complex list needs to be split during writing.

(0x0406) Data Member

This leaf specifies non-static data members of a class.

2	2	2	*	*
LF_MEMBER	@type	attribute	offset	name

@type	Index to type record for field.
attribute	Member attribute bit field.
offset	Numeric leaf specifying the offset of field in the structure.
name	Length-prefixed name of the member field.

(0x0407) Static Data Member

This leaf specifies the static data member of a class. Once a static data member has been found in this list, its symbol is found by qualifying the name with its class (T::name) and then searching the symbol table for a symbol by that name with the correct type index.

2	2	2	*
LF_STMEMBER	@type	attribute	name

@type	Index to type record for field.
attribute	Member attribute bit field.
name	Length-prefixed name of the member field.

(0x0408) Method

This leaf specifies the overloaded member functions of a class. This type record can also be used to specify a non-overloaded method, but is inefficient. The LF_ONEMETHOD record should be used for non-overloaded methods.

2	2	2	*
LF_METHOD	count	@mList	name

count	Number of occurrences of function within the class. If the function is overloaded, there will be multiple entries in the method list.
@mList	Type index of method list.
name	Length-prefixed name of method.

(0x0409) Nested Type Definition

This leaf specifies nested type definition with classes, structures, unions, or enums.

2	2	*
LF_NESTEDTYPE	@index	name

@index	Type index of nested type.

name	Length-prefixed name of type.

(0x040a) Virtual Function Table Pointer

This leaf specifies virtual table pointers within the class. It is a requirement that this record be emitted in the field list before any virtual functions are emitted to the field list.

2	2
LF_VFUNCTAB	@type

@type	Index to the pointer record describing the pointer. The pointer will in turn have an LF_VTSHAPE type record as the underlying type. Note that the offset of the virtual function table pointer from the address point of the class is always zero.

(0x040b) Friend Class

This leaf specifies a friend class.

2	2
LF_FRIENDCLS	@type

@type	Index to type record of the friend class. The name of the class can be obtained from the referenced record.

(0x040c) One Method

This record is used to specify a method of a class that is not overloaded.

2	2	2	4	*
LF_ONEMETHOD	attribute	@type	vbaseoff	name

attribute	Method attribute.
@type	Type index of method.
vbaseoff	Offset in virtual function table if virtual method. If the method is not virtual, then this field is not present.
name	Length-prefixed name of method.

(0x040d) Virtual Function Offset

This record is used to specify a virtual function table pointer at a non-zero offset relative to the address point of a class.

2	2	4
LF_VFUNCOFF	@type	offset

@type	Type index of virtual function table pointer.

offset	Offset of virtual function table pointer relative to address point of class.

4. Numeric Leaves

The following leaves are used in symbols and types where actual numeric values need to be specified. When the symbol or type processor knows that a numeric leaf is present, the next 2 bytes of the record are examined. If the value of these 2 bytes is less than LF_NUMERIC (0x8000), then the 2 bytes contain the actual value. If the value is greater than or equal to LF_NUMERIC (0x8000), then the numeric data follows the 2-byte leaf index and is contained in the number of bytes specified by the leaf index. Note that the LF_UCHAR numeric field is not necessary, because the value of the 8-bit unsigned character is less than 0x8000. Routines reading numeric fields must handle the potential non alignment of the data fields.

(0x8000) Signed Char

2	1
LF_CHAR	char

char 8-bit value.

(0x8001) Signed Short

2	2
LF_SHORT	short

short 16-bit signed value.

(0x8002) Unsigned Short

2	2
LF_USHORT	ushort

ushort 16-bit unsigned value.

(0x8003) Signed Long

2	4
LF_LONG	long

long 32-bit signed value.

(0x8004) Unsigned Long

2	4
LF_ULONG	ulong

ulong 32-bit unsigned value.

(0x8005) 32-bit Float

2	4
LF_REAL32	real32

real32 32-bit floating-point value.

(0x8006) 64-bit Float

2	8
LF_REAL64	real64

real64 64-bit floating-point value.

(0x8007) 80-bit Float

2	10
LF_REAL80	real80

real80 80-bit floating-point value.

(0x8008) 128 Bit Float

2	16
LF_REAL128	real128

real128 128-bit floating-point value.

(0x8009) Signed Quad Word

2	8
LF_QUADWORD	quadword

quadword 64-bit signed value.

(0x800a) Unsigned Quad Word

2	8
LF_UQUADWORD	uquadword

uquadword 64-bit unsigned value.

(0x800b) 48-bit Float

2	6
LF_REAL48	real48

real48 48-bit floating-point value.

(0x800c) 32-bit Complex

2	4	4
LF_COMPLEX32	real	imaginary

real Real part of complex number.
imaginary Imaginary part of complex number.

(0x800d) 64-bit Complex

2	8	8
LF_COMPLEX64	real	imaginary

real Real part of complex number.
imaginary Imaginary part of complex number.

(0x800e) 80-bit Complex

2	10	10
LF_COMPLEX80	real	imaginary

real Real part of complex number.
imaginary Imaginary part of complex number.

(0x800f) 128-bit Complex

2	16	16
LF_COMPLEX128	real	imaginary

real Real part of complex number.
imaginary Imaginary part of complex number.

(0x8010) Variable-length String

2	2	*
LF_VARSTRING	length	string

length Length of following string.

string	Variable-length string.

5. Predefined Primitive Types

5.1. Format of Reserved Types

Types 0 - 4095 (0 - 0x0fff) are reserved. These values are interpreted as bit fields with the following sizes and meanings.

11	10 - 8	7 - 4	3	2 - 0
reserved	mode	type	reserved	size

type One of the following types:

0x00	Special
0x01	Signed integral value
0x02	Unsigned integral value
0x03	Boolean
0x04	Real
0x05	Complex
0x06	Special2
0x07	Real int value
0x08	Reserved
0x09	Reserved
0x0a	Reserved
0x0b	Reserved
0x0c	Reserved
0x0d	Reserved
0x0e	Reserved
0x0f	Reserved for debugger expression evaluator

size Enumerated value for each of the types.

Type = special

0x00	No type
0x01	Absolute symbol
0x02	Segment
0x03	Void
0x04	Basic 8-byte currency value
0x05	Near Basic string
0x06	Far Basic string
0x07	Untranslated type from previous Microsoft symbol formats

Type = signed/unsigned integral and Boolean values

0x00	1 byte
0x01	2 byte
0x02	4 byte
0x03	8 byte
0x04	Reserved
0x05	Reserved
0x06	Reserved
0x07	Reserved

	Type = real and complex	
	0x00	32 bit
	0x01	64 bit
	0x02	80 bit
	0x03	128 bit
	0x04	48 bit
	0x05	Reserved
	0x06	Reserved
	0x07	Reserved
	Type = special2	
	0x00	Bit
	0x01	Pascal CHAR
	Type = Real int	
	0x00	Char
	0x01	Wide character
	0x02	2-byte signed integer
	0x03	2-byte unsigned integer
	0x04	4-byte signed integer
	0x05	4-byte unsigned integer
	0x06	8-byte signed integer
	0x07	8-byte unsigned integer
mode	Mode	
	0x00	Direct; not a pointer
	0x01	Near pointer
	0x02	Far pointer
	0x03	Huge pointer
	0x04	32-bit near pointer
	0x05	32-bit far pointer
	0x06	64-bit near pointer
	0x07	Reserved

5.2. Primitive Type Listing

Special Types

T_NOTYPE	0x0000	Uncharacterized type (no type)
T_ABS	0x0001	Absolute symbol
T_SEGMENT	0x0002	Segment type
T_VOID	0x0003	Void
T_PVOID	0x0103	Near pointer to void
T_PFVOID	0x0203	Far pointer to void
T_PHVOID	0x0303	Huge pointer to void
T_32PVOID	0x0403	32-bit near pointer to void
T_32PFVOID	0x0503	32-bit far pointer to void
T_CURRENCY	0x0004	Basic 8-byte currency value
T_NBASICSTR	0x0005	Near Basic string
T_FBASICSTR	0x0006	Far Basic string
T_NOTTRANS	0x0007	Untranslated type record from Microsoft symbol format
T_BIT	0x0060	Bit
T_PASCHAR	0x0061	Pascal CHAR

Character Types

T_CHAR	0x0010	8-bit signed
T_UCHAR	0x0020	8-bit unsigned
T_PCHAR	0x0110	Near pointer to 8-bit signed
T_PUCHAR	0x0120	Near pointer to 8-bit unsigned
T_PFCHAR	0x0210	Far pointer to 8-bit signed
T_PFUCHAR	0x0220	Far pointer to 8-bit unsigned
T_PHCHAR	0x0310	Huge pointer to 8-bit signed
T_PHUCHAR	0x0320	Huge pointer to 8-bit unsigned
T_32PCHAR	0x0410	16:32 near pointer to 8-bit signed
T_32PUCHAR	0x0420	16:32 near pointer to 8-bit unsigned
T_32PFCHAR	0x0510	16:32 far pointer to 8-bit signed
T_32PFUCHAR	0x0520	16:32 far pointer to 8-bit unsigned

Real Character Types

T_RCHAR	0x0070	Real char
T_PRCHAR	0x0170	Near pointer to a real char
T_PFRCHAR	0x0270	Far pointer to a real char
T_PHRCHAR	0x0370	Huge pointer to a real char
T_32PRCHAR	0x0470	16:32 near pointer to a real char
T_32PFRCHAR	0x0570	16:32 far pointer to a real char

Wide Character Types

T_WCHAR	0x0071	Wide char
T_PWCHAR	0x0171	Near pointer to a wide char
T_PFWCHAR	0x0271	Far pointer to a wide char
T_PHWCHAR	0x0371	Huge pointer to a wide char
T_32PWCHAR	0x0471	16:32 near pointer to a wide char
T_32PFWCHAR	0x0571	16:32 far pointer to a wide char

Real 16-bit Integer Types

T_INT2	0x0072	Real 16-bit signed int
T_UINT2	0x0073	Real 16-bit unsigned int
T_PINT2	0x0172	Near pointer to 16-bit signed int
T_PUINT2	0x0173	Near pointer to 16-bit unsigned int
T_PFINT2	0x0272	Far pointer to 16-bit signed int
T_PFUINT2	0x0273	Far pointer to 16-bit unsigned int
T_PHINT2	0x0372	Huge pointer to 16-bit signed int
T_PHUINT2	0x0373	Huge pointer to 16-bit unsigned int
T_32PINT2	0x0472	16:32 near pointer to 16-bit signed int
T_32PUINT2	0x0473	16:32 near pointer to 16-bit unsigned int
T_32PFINT2	0x0572	16:32 far pointer to 16-bit signed int
T_32PFUINT2	0x0573	16:32 far pointer to 16-bit unsigned int

16-bit Short Types

T_SHORT	0x0011	16-bit signed
T_USHORT	0x0021	16-bit unsigned
T_PSHORT	0x0111	Near pointer to 16-bit signed
T_PUSHORT	0x0121	Near pointer to 16-bit unsigned
T_PFSHORT	0x0211	Far pointer to 16-bit signed
T_PFUSHORT	0x0221	Far pointer to 16-bit unsigned
T_PHSHORT	0x0311	Huge pointer to 16-bit signed
T_PHUSHORT	0x0321	Huge pointer to 16-bit unsigned
T_32PSHORT	0x0411	16:32 near pointer to 16-bit signed
T_32PUSHORT	0x0421	16:32 near pointer to 16-bit unsigned
T_32PFSHORT	0x0511	16:32 far pointer to 16-bit signed
T_32PFUSHORT	0x0521	16:32 far pointer to 16-bit unsigned

Real 32-bit Integer Types

T_INT4	0x0074	Real 32-bit signed int
T_UINT4	0x0075	Real 32-bit unsigned int
T_PINT4	0x0174	Near pointer to 32-bit signed int
T_PUINT4	0x0175	Near pointer to 32-bit unsigned int
T_PFINT4	0x0274	Far pointer to 32-bit signed int
T_PFUINT4	0x0275	Far pointer to 32-bit unsigned int
T_PHINT4	0x0374	Huge pointer to 32-bit signed int
T_PHUINT4	0x0375	Huge pointer to 32-bit unsigned int
T_32PINT4	0x0474	16:32 near pointer to 32-bit signed int
T_32PUINT4	0x0475	16:32 near pointer to 32-bit unsigned int
T_32PFINT4	0x0574	16:32 far pointer to 32-bit signed int
T_32PFUINT4	0x0575	16:32 far pointer to 32-bit unsigned int

32-bit Long Types

T_LONG	0x0012	32-bit signed
T_ULONG	0x0022	32-bit unsigned
T_PLONG	0x0112	Near pointer to 32-bit signed
T_PULONG	0x0122	Near pointer to 32-bit unsigned
T_PFLONG	0x0212	Far pointer to 32-bit signed
T_PFULONG	0x0222	Far pointer to 32-bit unsigned
T_PHLONG	0x0312	Huge pointer to 32-bit signed
T_PHULONG	0x0322	Huge pointer to 32-bit unsigned
T_32PLONG	0x0412	16:32 near pointer to 32-bit signed
T_32PULONG	0x0422	16:32 near pointer to 32-bit unsigned
T_32PFLONG	0x0512	16:32 far pointer to 32-bit signed
T_32PFULONG	0x0522	16:32 far pointer to 32-bit unsigned

Real 64-bit int Types

T_INT8	0x0076	64-bit signed int
T_UINT8	0x0077	64-bit unsigned int
T_PINT8	0x0176	Near pointer to 64-bit signed int
T_PUINT8	0x0177	Near pointer to 64-bit unsigned int
T_PFINT8	0x0276	Far pointer to 64-bit signed int
T_PFUINT8	0x0277	Far pointer to 64-bit unsigned int
T_PHINT8	0x0376	Huge pointer to 64-bit signed int
T_PHUINT8	0x0377	Huge pointer to 64-bit unsigned int
T_32PINT8	0x0476	16:32 near pointer to 64-bit signed int
T_32PUINT8	0x0477	16:32 near pointer to 64-bit unsigned int
T_32PFINT8	0x0576	16:32 far pointer to 64-bit signed int
T_32PFUINT8	0x0577	16:32 far pointer to 64-bit unsigned int

64-bit Integral Types

T_QUAD	0x0013	64-bit signed
T_UQUAD	0x0023	64-bit unsigned
T_PQUAD	0x0113	Near pointer to 64-bit signed
T_PUQUAD	0x0123	Near pointer to 64-bit unsigned
T_PFQUAD	0x0213	Far pointer to 64-bit signed
T_PFUQUAD	0x0223	Far pointer to 64-bit unsigned
T_PHQUAD	0x0313	Huge pointer to 64-bit signed
T_PHUQUAD	0x0323	Huge pointer to 64-bit unsigned
T_32PQUAD	0x0413	16:32 near pointer to 64-bit signed
T_32PUQUAD	0x0423	16:32 near pointer to 64-bit unsigned
T_32PFQUAD	0x0513	16:32 far pointer to 64-bit signed
T_32PFUQUAD	0x0523	16:32 far pointer to 64-bit unsigned

32-bit Real Types

T_REAL32	0x0040	32-bit real
T_PREAL32	0x0140	Near pointer to 32-bit real
T_PFREAL32	0x0240	Far pointer to 32-bit real
T_PHREAL32	0x0340	Huge pointer to 32-bit real
T_32PREAL32	0x0440	16:32 near pointer to 32-bit real
T_32PFREAL32	0x0540	16:32 far pointer to 32-bit real

48-bit Real Types

T_REAL48	0x0044	48-bit real
T_PREAL48	0x0144	Near pointer to 48-bit real
T_PFREAL48	0x0244	Far pointer to 48-bit real
T_PHREAL48	0x0344	Huge pointer to 48-bit real
T_32PREAL48	0x0444	16:32 near pointer to 48-bit real
T_32PFREAL48	0x0544	16:32 far pointer to 48-bit real

64-bit Real Types

T_REAL64	0x0041	64-bit real
T_PREAL64	0x0141	Near pointer to 64-bit real
T_PFREAL64	0x0241	Far pointer to 64-bit real
T_PHREAL64	0x0341	Huge pointer to 64-bit real
T_32PREAL64	0x0441	16:32 near pointer to 64-bit real
T_32PFREAL64	0x0541	16:32 far pointer to 64-bit real

80-bit Real Types

T_REAL80	0x0042	80-bit real
T_PREAL80	0x0142	Near pointer to 80-bit real
T_PFREAL80	0x0242	Far pointer to 80-bit real
T_PHREAL80	0x0342	Huge pointer to 80-bit real
T_32PREAL80	0x0442	16:32 near pointer to 80-bit real
T_32PFREAL80	0x0542	16:32 far pointer to 80-bit real

128-bit Real Types

T_REAL128	0x0043	128-bit real
T_PREAL128	0x0143	Near pointer to 128-bit real
T_PFREAL128	0x0243	Far pointer to 128-bit real
T_PHREAL128	0x0343	Huge pointer to 128-bit real
T_32PREAL128	0x0443	16:32 near pointer to 128-bit real
T_32PFREAL128	0x0543	16:32 far pointer to 128-bit real

32-bit Complex Types

T_CPLX32	0x0050	32-bit complex
T_PCPLX32	0x0150	Near pointer to 32-bit complex
T_PFCPLX32	0x0250	Far pointer to 32-bit complex
T_PHCPLX32	0x0350	Huge pointer to 32-bit complex
T_32PCPLX32	0x0450	16:32 near pointer to 32-bit complex
T_32PFCPLX32	0x0550	16:32 far pointer to 32-bit complex

64-bit Complex Types

T_CPLX64	0x0051	64-bit complex
T_PCPLX64	0x0151	Near pointer to 64-bit complex
T_PFCPLX64	0x0251	Far pointer to 64-bit complex
T_PHCPLX64	0x0351	Huge pointer to 64-bit complex
T_32PCPLX64	0x0451	16:32 near pointer to 64-bit complex
T_32PFCPLX64	0x0551	16:32 far pointer to 64-bit complex

80-bit Complex Types

T_CPLX80	0x0052	80-bit complex
T_PCPLX80	0x0152	Near pointer to 80-bit complex
T_PFCPLX80	0x0252	Far pointer to 80-bit complex
T_PHCPLX80	0x0352	Huge pointer to 80-bit complex
T_32PCPLX80	0x0452	16:32 near pointer to 80-bit complex
T_32PFCPLX80	0x0552	16:32 far pointer to 80-bit complex

128-bit Complex Types

T_CPLX128	0x0053	128-bit complex
T_PCPLX128	0x0153	Near pointer to 128-bit complex
T_PFCPLX128	0x0253	Far pointer to 128-bit complex
T_PHCPLX128	0x0353	Huge pointer to 128-bit real
T_32PCPLX128	0x0453	16:32 near pointer to 128-bit complex
T_32PFCPLX128	0x0553	16:32 far pointer to 128-bit complex

Boolean Types

T_BOOL08	0x0030	8-bit Boolean
T_BOOL16	0x0031	16-bit Boolean
T_BOOL32	0x0032	32-bit Boolean
T_BOOL64	0x0033	64-bit Boolean
T_PBOOL08	0x0130	Near pointer to 8-bit Boolean
T_PBOOL16	0x0131	Near pointer to 16-bit Boolean
T_PBOOL32	0x0132	Near pointer to 32-bit Boolean
T_PBOOL64	0x0133	Near pointer to 64-bit Boolean
T_PFBOOL08	0x0230	Far pointer to 8-bit Boolean
T_PFBOOL16	0x0231	Far pointer to 16-bit Boolean
T_PFBOOL32	0x0232	Far pointer to 32-bit Boolean
T_PFBOOL32	0x0233	Far pointer to 64-bit Boolean
T_PHBOOL08	0x0330	Huge pointer to 8-bit Boolean
T_PHBOOL16	0x0331	Huge pointer to 16-bit Boolean
T_PHBOOL32	0x0332	Huge pointer to 32-bit Boolean
T_PHBOOL64	0x0333	Huge pointer to 64-bit Boolean
T_32PBOOL08	0x0430	16:32 near pointer to 8-bit Boolean
T_32PBOOL16	0x0431	16:32 near pointer to 16-bit Boolean
T_32PBOOL32	0x0432	16:32 near pointer to 32-bit Boolean
T_32PBOOL64	0x0433	16:32 near pointer to 64-bit Boolean
T_32PFBOOL08	0x0530	16:32 far pointer to 8-bit Boolean
T_32PFBOOL16	0x0531	16:32 far pointer to 16-bit Boolean
T_32PFBOOL32	0x0532	16:32 far pointer to 32-bit Boolean
T_32PFBOOL64	0x0533	16:32 far pointer to 64-bit Boolean

6. Register Enumerations

When the compiler emits a symbol that has been enregistered, the symbol record specifies the register by a register enumeration value. The enumeration is unique to each hardware architecture supported.

6.1. Intel 80x86/80x87 Architectures

0	none

8-bit Registers

1	AL
2	CL
3	DL
4	BL
5	AH
6	CH
7	DH
8	BH

16-bit Registers

9	AX
10	CX
11	DX
12	BX
13	SP
14	BP
15	SI
16	DI

32-bit Registers

17	EAX
18	ECX
19	EDX
20	EBX
21	ESP
22	EBP
23	ESI
24	EDI

Segment Registers

25	ES
26	CS
27	SS
28	DS
29	FS
30	GS

Special Cases

31	IP
32	FLAGS
33	EIP
34	EFLAGS

PCODE Registers

40	TEMP
41	TEMPH
42	QUOTE
43-47	Reserved

System Registers

80	CR0
81	CR1
82	CR2
83	CR3
90	DR0
91	DR1
92	DR2
93	DR3
94	DR4
95	DR5
96	DR6
97	DR7

Register Extensions for 80x87

128	ST(0)
130	ST(2)
131	ST(3)
132	ST(4)
133	ST(5)
134	ST(6)
135	ST(7)
136	CONTROL
137	STATUS
138	TAG
139	FPIP
140	FPCS
141	FPDO
142	FPDS
143	ISEM
144	FPEIP
145	FPEDO

6.2. Motorola 68000 Architectures

0	Data register 0
1	Data register 1
2	Data register 2
3	Data register 3
4	Data register 4
5	Data register 5
6	Data register 6
7	Data register 7
8	Address register 0
9	Address register 1
10	Address register 2
11	Address register 3
12	Address register 4
13	Address register 5
14	Address register 6
15	Address register 7
16	??CV_R68_CCR
17	??CV_R68_SR
18	??CV_R68_USP
19	??CV_R68_MSP
20	??CV_R68_SFC
21	??CV_R68_DFC
22	??CV_R68_CACR
23	??CV_R68_VBR
24	??CV_R68_CAAR
25	??CV_R68_ISP
26	??CV_R68_PC
27	Reserved
28	??CV_R68_FPCR
29	??CV_R68_FPSR
30	??CV_R68_FPIAR
31	Reserved
32	Floating-point 0
33	Floating-point 1
34	Floating-point 2
35	Floating-point 3
36	Floating-point 4
37	Floating-point 5
38	Floating-point 6
39	Floating-point 7
40 - 50	Reserved
51	CV_R68_PSR
52	CV_R68_PCSR

6.3. MIPS Architectures

Integer Register

0	NoRegister
10	IntZero
11	IntAT
12	IntV0
13	IntV1
14	IntA0
15	IntA1
16	IntA2
17	IntA3
18	IntT0
19	IntT1
20	IntT2
21	IntT3
22	IntT4
23	IntT5
24	IntT6
25	IntT7
26	IntS0
27	IntS1
28	IntS2
29	IntS3
30	IntS4
31	IntS5
32	IntS6
33	IntS7
34	IntT8
35	IntT9
36	Int KT0
37	IntKT1
38	IntGP
39	IntSP
40	IntS8
41	IntRA
42	Int Lo
43	Int Hi
50	Fir
51	PSR
60	Floating-point register 0
61	Floating-point register 1
62	Floating-point register 2
63	Floating-point register 3
64	Floating-point register 4
65	Floating-point register 5
66	Floating-point register 6
67	Floating-point register 7
68	Floating-point register 8

69	Floating-point register 9
70	Floating-point register 10
71	Floating-point register 11
72	Floating-point register 12
73	Floating-point register 13
74	Floating-point register 14
75	Floating-point register 15
76	Floating-point register 16
77	Floating-point register 17
78	Floating-point register 18
79	Floating-point register 19
80	Floating-point register 20
81	Floating-point register 21
82	Floating-point register 22
83	Floating-point register 23
84	Floating-point register 24
85	Floating-point register 25
86	Floating-point register 26
87	Floating-point register 27
88	Floating-point register 28
89	Floating-point register 29
90	Floating-point register 30
91	Floating-point register 31
92	Floating-point status register

7. Symbol and Type Format for Microsoft Executables

7.1. Introduction

This section describes the format used to embed debugging information into the executable file.

7.2. Debug Information Format

The debug information format encompasses a block of data that goes into the .exe file at a location dependent upon the executable file format. The version of the debug information is specified by a signature that is contained within the debug information. The signature has the format of **NBxx**, where xx is the version number and has the following meanings:

NB00	Not supported.
NB01	Not supported.
NB02	Linked by a Microsoft LINK, version 5.10, or equivalent OEM linker.
NB03	Not supported.
NB04	Not supported.
NB05	Emitted by LINK, version 5.20 and later linkers for a file before it has been packed.
NB06	Not supported.
NB07	Used for Quick C for Windows 1.0 only.
NB08	Used by Microsoft CodeView debugger, versions 4.00 through 4.05, for a file after it has been packed. Microsoft CodeView,, version 4.00 through 4.05 will not process a file that does not have this signature.
NB09	Used by Microsoft CodeView, version 4.10 for a file after it has been packed. Microsoft CodeView 4.10 will not process a file that does not have this signature.

The method for finding the debug information depends upon the executable format.

OMF

For OMF executables, the debug information is at the end of the .exe file, i.e., after the header plus load image, the overlays, and the Windows resource compiler information. The lower portion of the file is unaffected by the additional data. The last eight bytes of the file contain a signature and a long file offset from the end of the file (**lfoBase**). The long offset indicates the position in the file (relative to the end of the file) of the base address.

The value

lfaBase = length of the file **- lfoBase**

gives the base address of the start of the Symbol and Type OMF information relative to the beginning of the file.

executable header	
executable code + ...	
NBxx	Signature at **lfaBase**
lfoDirectory	Offset of directory from base address (**lfoDir**)
Subsection tables	sstModule, sstType, sstLibraries, ...
.	
.	
.	
Subsection Directory	At file offset **lfaBase + lfoDir**
NBxx	Signature
lfoBase	Offset of repeated signature from end of file

PE Format

For PE format executables, the base address **lfaBase** is found by examining the executable header. Note, currently Microsoft code uses the same method that is used for OMF format executables to find the debug information.

executable header	Contains pointer to debug information
executable code + ...	
NBxx	Signature at **lfaBase**
lfoDirectory	Offset of directory from base address (**lfoDir**)
Subsection tables	sstModule, sstType, sstLibraries, ...
.	
.	
.	
Subsection Directory	At file offset **lfaBase + lfoDir**
other information	

All other file offsets in the Symbol and Type OMF are relative to **lfaBase**. At the base address, the signature is repeated, followed by the long displacement to the subsection directory (**lfoDir**). All subsections start on a long word boundary and are designed to maintain natural alignment internally in each subsection and within the subsection directory.

7.3. Subsection Directory

The subsection directory has the following format:

Directory header
Directory entry 0
Directory entry 1
.
.
.
Directory entry *n*

The subsection directory is prefixed with a directory header structure indicating size and number of subsection directory entries that follow.

2	2	4	4	4
cbDirHeader	cbDirEntry	cDir	lfoNextDir	flags

cbDirHeader	Length of directory header.
cbDirEntry	Length of each directory entry.
cDir	Number of directory entries.
lfoNextDir	Offset from **lfaBase** of next directory. This field is currently unused, but is intended for use by the incremental linker to point to the next directory containing Symbol and Type OMF information from an incremental link.
flags	Flags describing directory and subsection tables. No values have been defined for this field.

The directory header structure is followed by the directory entries, which specify the subsection type, module index, if applicable, the subsection offset, and subsection size.

2	2	4	4
subsection	iMod	lfo	cb

subsection	Subdirectory index. See the table below for a listing of the valid subsection indices.
iMod	Module index. This number is 1 based and zero (0) is never a valid index. The index 0xffff is reserved for tables that are not associated with a specific module. These tables include sstLibraries, sstGlobalSym, sstGlobalPub, and sstGlobalTypes.
lfo	Offset from the base address **lfaBase**.
cb	Number of bytes in subsection.

There is no requirement for a particular subsection to exist for a particular module. There is a preferred order for subsections within the Symbol and Type OMF portion and the subsection directory of the file, as emitted by the linker (NB05 signature). The preferred order is the following:

$sstModule_1$	Module 1
.	.
$sstModule_n$	Module n
$sstTypes_1$	Module 1
$sstPublics_1$	Module 1
$sstSymbols_1$	Module 1
$sstSrcModule_1$	Module 1
.	
$sstTypes_n$	Module n
$sstPublics_n$	Module n
$sstSymbols_n$	Module n
$sstSrcModule_n$	Module n
sstLibraries	
directory	

However, if the tables are not written in this order by the linker, the CVPACK utility will sort the subsection table into this order and read the subsections in this order by seeking the correct location. The net effect is that packing will be less efficient, but it will work.

CVPACK will write the Symbol and Type OMF back to the file in the order listed below. The Microsoft debugger requires that the sstModule entries be first and sequential in the subsection directory. For performance reasons, it is recommended that the order of the subsections in the file match the order of the subsection directory entries.

For signatures prior to NB09, the packed file has the following subsections and ordering:

NBxx	Signature
lfoDir	Directory offset
$sstModule_1$	Module 1
.	.
$sstModule_n$	Module n
$sstAlignSym_1$	Module 1
$sstSrcModule_1$	Module 1
.	
$sstAlignSym_n$	Module n
$sstSrcModule_n$	Module n
sstGlobalPub	Global Publics
sstGlobalSym	Global Symbols
sstLibraries	Libraries
sstGlobalTypes	Global Types
Directory	
NBxx	Signature, if OMF executable
lfoBase	Offset of base, if OMF executable

For NB09 signatures, the packed file has the following subsections and ordering:

NBxx	Signature
lfoDir	Directory offset
$sstModule_1$	Module 1
.	.
$sstModule_n$	Module n
$sstAlignSym_1$	Module 1
$sstSrcModule_1$	Module 1
.	
$sstAlignSym_n$	Module n
$sstSrcModule_n$	Module n
sstGlobalPub	Global Publics
sstGlobalSym	Global Symbols
sstLibraries	Libraries
sstGlobalTypes	Global Types
sstStaticSym	Static Symbols
sstFileIndex	File Index
Directory	
NBxx	signature
lfoBase	offset

7.4. SubSection Types (sst...)

All values not defined in the following list are reserved for future use:

sstModule	0x120
sstTypes	0x121
sstPublic	0x122
sstPublicSym	0x123
sstSymbols	0x124
sstAlignSym	0x125
sstSrcLnSeg	0x126
sstSrcModule	0x127
sstLibraries	0x128
sstGlobalSym	0x129
sstGlobalPub	0x12a
sstGlobalTypes	0x12b
sstMPC	0x12c
sstSegMap	0x12d
sstSegName	0x12e
sstPreComp	0x12f
unused	0x130
reserved	0x131
reserved	0x132
sstFileIndex	0x133
sstStaticSym	0x134

(0x0120) sstModule

This describes the basic information about an object module, including code segments, module name, and the number of segments for the modules that follow. Directory entries for sstModules precede all other subsection directory entries.

2	2	2	2	*	*
ovlNumber	iLib	cSeg	Style	SegInfo	Name

ovlNumber	Overlay number.
iLib	Index into sstLibraries subsection if this module was linked from a library
cSeg	Count or number of code segments to which this module contributes.
Style	Debugging style for this module. Currently only "CV" is defined. A module can have only one debugging style. If a module contains debugging information in an unrecognized style, the information will be discarded.
SegInfo	Detailed information about each segment to which code is contributed. This is an array of *cSeg* count segment information descriptor structures.
Name	Length-prefixed name of module

SegInfo is a structure that describes each segment to which a module contributes code. It is formatted as follows:

2	2	4	4
Seg	pad	offset	cbSeg

Seg	Segment that this structure describes.
pad	Padding to maintain alignment This field is reserved for future use and must be emitted as zeroes.
offset	Offset in segment where the code starts.
cbSeg	Count or number of bytes of code in the segment.

(0x0121) sstTypes

The linker emits one of these subsections for every object file that contains a $$TYPES segment. CVPACK combines all of these subsections into an sstGlobalTypes subsection and deletes the sstTypes tables. The sstTypes table contains the contents of the $$TYPES segment, except that addresses within the $$TYPES segment have been fixed by the linker. (See also sstPreComp.)

(0x0122) sstPublic

The linker fills each subsection of this type with entries for the public symbols of a module. The CVPACK utility combines all of the sstPublics subsections into an sstGlobalPub subsection. This table has been replaced with the sstPublicSym, but is retained for compatibility with previous linkers.

2/4	2	2	*
offset	seg	type	name

offset Offset of public within segment. This will be a 16-bit offset unless the executable is a 32-bit executable. Note that if any public symbols are 16:32 model, then all publics are emitted as 16:32 addresses.
seg Segment index.
type Type index of the symbol. This will be zero if the module was compiled without Microsoft symbol and type information.
name Length-prefixed name of public

(0x0123) sstPublicSym

This table replaces the sstPublic subsection. The format of the public symbols contained in this table is that of an S_PUB16 or S_PUB32 symbol, as defined in Sections 2.3 and 2.4. This allows an executable to contain both 16:16 and 16:32 public symbols for mixed-mode executable files. As with symbols sections, public section records must start on a 4-byte boundary.

(0x0124) sstSymbols

The linker emits one of these subsections for every object file that contains a $$SYMBOLS segment. The sstSymbols table contains the contents of the $$SYMBOLS segment, except that addresses within the $$SYMBOLS segment have been fixed by the linker. The CVPACK utility moves global symbols from the sstSymbols subsection to the sstGlobalSum subsection during packing. When the remaining symbols are written executables, the subsection type is changed to sstAlignSym.

(0x0125) sstAlignSym

CVPACK writes the remaining unpacked symbols for a module back to the executable in a subsection of this type. All symbols have been padded to fall on a long word boundary, and the lexical scope linkage fields have been initialized.

(0x0126) sstSrcLnSeg

The linker fills in each subsection of this type with information obtained from any LINNUM records in the module. This table has been replaced by the sstSrcModule, but is retained for compatibility with previous linkers. CVPACK rewrites sstSrcLnSeg tables to sstSrcModule tables.

*	2	2	*
name	seg	cPair	line/offset

name Length-prefixed name of source file.
seg Segment.
cPair Count or number of line number offset pairs to follow.

line/offset	Line/offset pairs. This pair consists of the line number followed by the offset of the start of the code for that line within the segment. All offsets are relative to the beginning of the segment, not the start of the contribution of the module to the segment. For example, if the module contributes to segment _TEXT starting at offset 0x0100, and the code offset of the first line number is 0x0010 relative to the module, it will show up in the subsection as 0x0110. The offsets are 16 bits if the executable is a 16:16 executable. If any segment in the executable is 16:32 model, then all offsets in the line/offset pairs are 32-bit offsets.

(0x0127) sstSrcModule

The following table describes the source line number for addressing mapping information for a module. The table permits the description of a module containing multiple source files with each source file contributing code to one or more code segments. The base addresses of the tables described below are all relative to the beginning of the sstSrcModule table.

Module header
Information for source file 1
Information for segment 1
Information for segment 2
.
Information for source file 2
Information for segment 1
Information for segment 2
.
.

The module header structure describes the source file and code segment organization of the module.

2	2	4*cFile	8*cSeg	2*cSeg
cFile	cSeg	baseSrcFile	start/end	seg

cFile	Number of source files contributing code to segments.
cSeg	Number of code segments receiving code from this module.
baseSrcFile	An array of base offsets from the beginning of the sstSrcModule table.
start/end	An array of two 32-bit offsets per segment that receives code from this module. The first offset is the offset within the segment of the first byte of code from this module. The second offset is the ending address of the code from this module. The order of these pairs corresponds to the ordering of the segments in the *seg* array. Zeroes in these entries means that the information is not known, and the file and line tables described below need to be examined to determine if an address of interest is contained within the code from this module.
seg	An array of segment indices that receive code from this module. If the number of segments is not even, two pad characters are inserted to maintain natural alignment.

The file table describes the code segments that receive code from each source file.

2	2	4*cSeg	8*cSeg	2	*
cSeg	pad	baseSrcLn	start/end	cbName	Name

cSeg	Number of segments that receive code from this source file. If the source file contributes code multiple times to a segment, it is reflected in this count.
pad	Pad field used to maintain alignment. This field is reserved for future use and must be emitted as zero.
baseSrcLn	An array of offsets for the line/address mapping tables for each of the segments that receive code from this source file.
start/end	An array of two 32-bit offsets per segment that receives code from this module. The first offset is the offset within the segment of the first byte of code from this module. The second offset is the ending address of the code from this module. The order of these pairs corresponds to the ordering of the segments in the *seg* array. Zeroes in these entries means that the information is not known, and the file and line tables described below need to be examined to determine if an address of interest is contained within the code from this module.
cbName	Count or number of bytes in source file name.
Name	Source file name. This can be a fully or partially qualified path name.

The preferred ordering for this table is by offset order. Line number and offsets must be unique. The line number to address mapping information is contained in a table with the following format:

2	2	4*cPair	2*cPair
Seg	cPair	offset	linenumber

Seg	Segment index for this table.
cPair	Count or number of source line pairs to follow.
offset	An array of 32-bit offsets for the offset within the code segment of the start of the line contained in the parallel array *linenumber*.
linenumber	An array of 16-bit line numbers for the numbers of the lines in the source file that cause code to be emitted to the code segment. This array is parallel to the *offset* array. If *cPair* is not even, then a zero word is emitted to maintain natural alignment in the sstSrcModule table.

(0x0128) sstLibraries

There can be at most one sstLibraries SubSection. The format is an array of length-prefixed names, which define all the library files used during linking. The order of this list defines the library index number (seethe sstModules subsection). The first entry should be empty, i.e., a zero-length string, because library indices are 1-based.

(0x0129) sstGlobalSym

This subsection contains globally compacted symbols. The format of the table is a header specifying the symbol and address hash functions, the length of the symbol information, the length of the symbol hash function data, and the length of address hash function data. This is followed by the symbol information, which followed by the symbol hash tables, and then followed by the address hash tables. When the pack utility writes the sstGlobals subsection, each symbol is zero-padded such that the following symbol starts on a long boundary, and the length field is adjusted by the pad count. Note that symbol and/or address hash data can be discarded and the globally packed symbols are linearly searched. A hash function index 0 means that no hash data exists. See Section 7.5 for more information about the hashing functions.

The header has the following format:

2	2	4	4	4
symhash	addrhash	cbSymbol	cbSymHash	cbAddrHash

symhash	Index of the symbol hash function.
addrhash	Index of the address hash function.
cbSymbol	Count or number of bytes in the symbol table.
cbSymHash	Count or number of bytes in the symbol hash table.
cbAddrHash	Count or number of bytes in the address hashing table.

Starting with the NB09 signature files, the sstGlobalSym table can contain S_ALIGN symbols to maintain a 4-K alignment of symbols. Also, starting with NB09 signature files, the sstGlobal can contain S_PROCREF and S_DATAREF symbols to global procedures and to global data symbols that would not otherwise have been globally packed because of symbol type mismatches. See Section 2.6 for more information about the S_PROCREF and S_DATAREF symbols.

(0x012a) sstGlobalPub

This subsection contains the globally compacted public symbols from the sstPublics. The format of the table is a header specifying the symbol and address hash functions, the length of the symbol information, the length of the symbol hash function data, and the length of address hash function data. This is followed by symbol information, which is followed by the symbol hash tables, and then followed by the address hash tables. When the pack utility writes the sstGlobals subsection, each symbol is zero-padded such that the following symbol starts on a long boundary, and the length field of the symbol is adjusted by the pad count. Note that symbol and/or address hash data can be discarded and the globally packed symbolscan be linearly searched in low-memory situations. A hash function index 0 means that no hash data exists. See Section 7.5 for more information about the hashing functions.

The header has the following format:

2	2	4	4	4
symhash	addrhash	cbSymbol	cbSymHash	cbAddrHash

symhash	Index of the symbol hash function.
addrhash	Index of the address hash function.

cbSymbol	Count or number of bytes in the symbol table.
cbSymHash	Count or number of bytes in the symbol hash table.
cbAddrHash	Count or number of bytes in the address hashing table.

Starting with the NB09 signature files, the sstGlobalSym table can contain S_ALIGN symbols to maintain a 4-K alignment of symbols.

They contain S_ALIGN symbol records to maintain a 4-K alignment of tables. Note also that sstGlobalPub table contains S_PROCREF symbols.

(0x012b) sstGlobalTypes

This subsection contains the packed type records for the executable file. The first long word of the subsection contains the number of types in the table. This count is followed by a count-sized array of long offsets to the corresponding type record. As the sstGlobalTypes subsection is written, each type record is forced to start on a long word boundary. However, the length of the type string is not adjusted by the pad count. The remainder of the subsection contains the type records. This table is invalid for NB05 signatures.

Types are 48-K aligned as well as naturally aligned, so linear traversal of the type table is non-trivial. The 48-K alignment means that no type record crosses a 48-K boundary.

flags	Types table flag
cType	Count or number of types
offType[cType]	Offset of each type See note below.
type string 0	Type string for type index 0x1000
type string 1	Type string for type index 0x1001
.	
type string n	Type string for type index 0x1000 + n

Note that for NB07 and NB08 executables, the type string offset is from the beginning of the subsection table. For NB09 executables, the type string offset is from the first type record of the sstGlobalTypes subsection. Using the offset from the first type record simplifies demand loading of the sstGlobalTypes table.

The types table flags entry has the following format:

3	1
unused	signature

unused	Reserved for future use. Must be emitted as zeroes.
signature	Global types table signature.

(0x012c) sstMPC

This table is emitted by the Pcode MPC program when a segmented executable is processed into a non-segmented executable file. The table contains the mapping from segment indices to frame numbers.

2	2*cSeg
cSeg	mpSegFrame

cSeg Count or number of segments converted

mpSegFrame Segment-to-frame mapping table. A segmented address *segment:offset* is converted to a frame by *mpSegFrame*[segment-1]*16 + offset

(0x012d) sstSegMap

This table contains the mapping between the logical segment indices used in the symbol table and the physical segments where the program was loaded

There is one **sstSegMap** per executable or DLL.

2	cSeg	Count or number of segment descriptors in table
2	cSegLog	Count or number of logical segment descriptors
20	SegDesc 0	First segment descriptor
	.	
	.	
20	SegDesc N	cSeg'th segment descriptor

cSeg Total number of segment descriptors.

cSegLog Total number of logical segments. All group descriptors follow the logical segment descriptors. The number of group descriptors is given by *cSeg* - *cSegLog.*

SegDescN Array of segment descriptors. Information about a logical segment can be found by using *logical segment number* - *1* as an index into this array. Subtract 1 because the logical segment number is 1 based.

Each element of the segment descriptor array has the following format:

2	2	2	2	2	2	4	4
flags	ovl	group	frame	iSegName	iClassName	offset	cbseg

flags Descriptor flags bit field. See below for details.

ovl Logical overlay number.

group Group index into the descriptor array. The group index must either be 0 or *cSegLog* <= *group* < *cSeg.*

frame	This value has the following different meanings depending upon the values of *fAbs* and *fSel* in the *flags* bit array and *ovl*:

fAbs	*fSel*	*ovl*	**Operation**
0	0	0	Frame is added to PSP + 0x10 if not a .com file
0	0	0	Frame is added to PSP if it is a .com file
0	0	!= 0	Frame is added to current overlay base
1	0	x	Frame is absolute address
0	1	x	Frame contains a selector

iSegName	Byte index of the segment or group name in the **sstSegName** table. A value of 0xffff indicates that there is no name.
iClassName	Byte index of the class name in the **sstSegName** table. A value of 0xffff indicates that there is no name.
offset	Byte offset of the logical segment within the specified physical segment. If *fGroup* is set in *flags*, *offset* is the offset of the group in the physical segment. Currently all groups define physical segments, so *offset* will be zero for groups.
cbSeg	Byte count of the logical segment or group.

The descriptor flags bit field *flags* has the following format:

:3	:1	:2	:1	:1	:4	:1	:1	:1	:1
res	fGroup	res	fAbs	fSel	res	f32Bit	fExecut e	fWrite	fRead

res	Reserved and set to zero.
fGroup	If set, the descriptor represents a group. Because groups are not assigned logical segment numbers, these entries are placed after the logcial segment descriptors in the descriptor array.
fAbs	*frame* represents an absolute address.
fSel	*frame* represents a selector.
f32Bit	The descriptor describes a 32-bit linear address.
fExecute	The segment is executable.
fWrite	The segment is writable.
fRead	The segment is readable.

(0x012e) sstSegName

The **sstSegName** table contains all of the logical segment and class names. The table is an array of zero-terminated strings. Each string is indexed by its beginning from the start of the table. See sstSegMap above.

(0x012f) sstPreComp

The linker emits one of these sections for every OMF object that has the $$TYPES table flagged as sstPreComp and for every COFF object that contains a .debug$P section. During packing, the CVPACK utility processes modules with a types table having the sstPreComp index before modules with types table having the sstTypes index.

(0x0131) Reserved

Reserved for internal use.

(0x0132) Reserved

Reserved for internal use.

(0x0133) sstFileIndex

This subsection contains a list of all of the sources files that contribute code to any module (compiland) in the executable. File names are partially qualified relative to the compilation directory.

2	2	2 * cMod	2 * cModules	4 * cRef	*
cMod	cRef	ModStart	cRefCnt	NameRef	Names

cMod	Count or number of modules in the executable.
cRef	Count or total number of file name references.
ModStart	Array of indices into the *NameOffset* table for each module. Each index is the start of the file name references for each module.
cRefCnt	Number of file name references per module.
NameRef	Array of offsets into the *Names* table. For each module, the offset to first referenced file name is at *NameRef[ModStart]* and continues for *cRefCnt* entries.
Names	List of zero-terminated file names. Each file name is partially qualified relative to the compilation directory.

(0x0134) sstStaticSym

This subsection is structured exactly like the sstGlobalPub and sstGlobalSym subsections. It contains S_PROCREF for all static functions, as well as S_DATAREF for static module level data and non-static data that could not be included (due to type conflicts) in the sstGlobalSym subsection.

7.5. Hash table and sort table descriptions

The NB09 signature Microsoft symbol and type information contains hash/sort tables in the sstGlobalSym, sstGlobalPub, and sstStaticSym subsections.

Name hash table (symhash == 10):

The symbol name hash table uses the following checksum algorithm to generate the hash.

```
byt_toupper(b)    <- (b&0xDF)
dwrd_toupper(dw)  <- (dw&0xDFDFDFDF)

cb  = {Number of characters in the name}
lpbName= {pointer to the first character of the name}

ulEnd = 0;
while ( cb & 3 ) {
    ulEnd |= byt_toupper ( lpbName [ cb - 1 ] );
    ulEnd <<= 8;
    cb -= 1;
}

cul = cb / 4;

lpulName = lpbName;
for ( iul = 0; iul < cul; iul++ ) {
    ulSum ^= dwrd_toupper(lpulName[iul]);
    _lrotl ( ulSum, 4 );
}
ulSum ^= ulEnd;
```

The hash bucket number is derived from ulSum, by taking the modulo of ulSum with the total number of hash buckets.

The format of the table is as follows:

2	cHash(n)	Number of hash buckets.
2	Alignment	Filler to preserve alignment.
4n	Hash Table[n]	Each ulong entry is a file offset from the beginning of the chain table to the first chain item for each hash bucket.
4n	Bucket Counts[n]	Each ulong entry is the count of items in the chain for each hash bucket.
8m	Chain table[m]	Each entry is a pair of dwords. The first dword is the file offset of the referenced symbol from the beginning of the symbols. The second dword is the checksum of the referenced symbol generated by the above algorithm.

n = the number of hash buckets.
m = the number of symbols (with names) = the number of entries in the chain table.

Address sort table (addrhash == 12):

The address sort table is a grouping of logical segments (or sections) in which each symbol reference within the segment/section is sorted by its segment/section relative offset.

The format of the table is as follows:

2	cSeg(n)	Number of logical segments/sections.
2	Alignment	Filler to preserve alignment.
4n	Segment Table[n]	Each ulong entry is a file offset from the beginning of the offset table to the first offset item for each segment/section.
4n	Offset Counts[n]	Each ulong entry is the count of items in the offset table for each segment.
8m	Offset Table[m]	Each entry is a pair of dwords. The first dword is the file offset of the referenced symbol from the beginning of the symbols. The second dword is the segment/section relative offset of the referenced symbol in memory.

n = the number of segments/sections.

m = the number of symbols (with addresses) = the number of entries in the offset table.