Microsoft Symbol and Type Information

Contents

[1. Symbol and Type Information 3](#_Toc117848129)

[1.1. Logical Segments 3](#_Toc117848130)

[1.2. Lexical Scope Linkage 3](#_Toc117848131)

[1.3. Numeric Leaves 4](#_Toc117848132)

[1.4. Types Indices 4](#_Toc117848133)

[1.5. $$SYMBOLS and $$TYPES Definitions 4](#_Toc117848134)

[1.5.1. $$TYPES Definition 4](#_Toc117848135)

[1.5.2. $$SYMBOLS Definition 5](#_Toc117848136)

[2. Symbols 5](#_Toc117848137)

[2.1. General 5](#_Toc117848138)

[Format of Symbol Records 5](#_Toc117848139)

[Symbol Indices 6](#_Toc117848140)

[2.2. Non-modal Symbols 9](#_Toc117848141)

[2.3. Symbols for 16:16 Segmented Architectures 9](#_Toc117848142)

[2.4. Symbols for 16:32 Segmented Architectures 9](#_Toc117848143)

[2.5. Symbols for MIPS Architectures 9](#_Toc117848144)

[2.6. Symbols for CVPACK Optimization 9](#_Toc117848145)

[3. Types Definition Segment ($$TYPES) 9](#_Toc117848146)

[3.1. Type Record 9](#_Toc117848147)

[3.2. Type String 9](#_Toc117848148)

[3.3. Leaf Indices Referenced from Symbols 9](#_Toc117848149)

[Member Attribute Field 11](#_Toc117848150)

[3.4. Type Records Referenced from Type Records 12](#_Toc117848151)

[3.5. Subfields of Complex Lists 12](#_Toc117848152)

[4. Numeric Leaves 12](#_Toc117848153)

[5. Predefined Primitive Types 12](#_Toc117848154)

[5.1. Format of Reserved Types 12](#_Toc117848155)

[5.2. Primitive Type Listing 12](#_Toc117848156)

[6. Register Enumerations 12](#_Toc117848157)

[6.1. Intel 80x86/80x87 Architectures 12](#_Toc117848158)

[6.2. Motorola 68000 Architectures 12](#_Toc117848159)

[6.3. MIPS Architectures 12](#_Toc117848160)

[7. Symbol and Type Format for Microsoft Executables 12](#_Toc117848161)

[7.1. Introduction 12](#_Toc117848162)

[7.2. Debug Information Format 12](#_Toc117848163)

[OMF 13](#_Toc117848164)

[PE Format 13](#_Toc117848165)

[7.3. Subsection Directory 13](#_Toc117848166)

[7.4. SubSection Types (sst...) 15](#_Toc117848167)

[(0x0120) sstModule 15](#_Toc117848168)

[(0x0121) sstTypes 15](#_Toc117848169)

[(0x0122) sstPublic 15](#_Toc117848170)

[(0x0123) sstPublicSym 15](#_Toc117848171)

[(0x0124) sstSymbols 15](#_Toc117848172)

[(0x0125) sstAlignSym 16](#_Toc117848173)

[(0x0126) sstSrcLnSeg 16](#_Toc117848174)

[(0x0127) sstSrcModule 16](#_Toc117848175)

[(0x0128) sstLibraries 16](#_Toc117848176)

[(0x0129) sstGlobalSym 16](#_Toc117848177)

[(0x012a) sstGlobalPub 16](#_Toc117848178)

[(0x012b) sstGlobalTypes 16](#_Toc117848179)

[(0x012c) sstMPC 16](#_Toc117848180)

[(0x012d) sstSegMap 16](#_Toc117848181)

[(0x012e) sstSegName 16](#_Toc117848182)

[(0x012f) sstPreComp 16](#_Toc117848183)

[(0x0133) sstFileIndex 16](#_Toc117848184)

[(0x0134) sstStaticSym 16](#_Toc117848185)

[7.5. Hash table and sort table descriptions 16](#_Toc117848186)

# 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 Intel386TM and Intel486TM 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.

## 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.

## 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 blockend 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. |

## 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.

## 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.

## $$SYMBOLS and $$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.

# Symbols

## General

### Format of Symbol Records

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

| **Offset** | **Size** | **Field** | **Description** |
| --- | --- | --- | --- |
|  | 2 | length | Length of record, excluding the length field. |
|  | 2 | 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\_REGISTER16 | Register variable |
| 0x0003 | S\_CONSTANT16 | Constant symbol |
| 0x0004 | S\_UDT16 | 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\_ST | Specify name of object file |
| 0x000a | S\_ENDARG | Specify end of arguments in function symbols |
| 0x000b | S\_COBOLUDT16 | Microfocus COBOL user-defined type |
| 0x000c | S\_MANYREG16 | Many register symbol |
| 0x000d | S\_RETURN | Function return description |
| 0x000e | S\_ENTRYTHIS | Description of this pointer at entry |
| 0x0024 | S\_USES |  |
| 0x0025 | S\_NAMESPACE |  |
| 0x0026 | S\_USING |  |
| 0x0027 | S\_PCONSTANT |  |
| 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\_16 | BP relative 16:32 |
| 0x0201 | S\_LDATA32\_16 | Local data 16:32 |
| 0x0202 | S\_GDATA32\_16 | Global data 16:32 |
| 0x0203 | S\_PUB32\_16 | Public symbol 16:32 |
| 0x0204 | S\_LPROC32\_16 | Local procedure start 16:32 |
| 0x0205 | S\_GPROC32\_16 | Global procedure start 16:32 |
| 0x0206 | S\_THUNK32\_ST | Thunk start 16:32 |
| 0x0207 | S\_BLOCK32\_ST | Block start 16:32 |
| 0x0208 | S\_WITH32\_ST |  |
| 0x0209 | S\_LABEL32\_ST |  |
| 0x020a | S\_CEXMODEL32 |  |
| 0x020b | S\_VFTPATH32 | Virtual function table path descriptor 16:32 |
| 0x020c | S\_REGREL32\_16 | 16:32 offset relative to arbitrary register |
| 0x020d | S\_LTHREAD32\_16 | Local Thread Storage data |
| 0x020e | S\_GTHREAD32\_16 | Global Thread Storage data |
| 0x020f | S\_SLINK32 |  |
| 0x0211 | S\_OPTVAR32 |  |
| 0x0300 | S\_LPROCMIPS16 | Local procedure start MIPS |
| 0x0301 | S\_GPROCMIPS16 | Global procedure start MIPS |
| 0x0400 | S\_PROCREF\_ST | Reference to a procedure |
| 0x0401 | S\_DATAREF\_ST | Reference to data |
| 0x0402 | S\_ALIGN | Page align symbols |
| 0x0403 | S\_LPROCREF\_ST |  |
| 0x0404 | S\_OEM |  |
| 0x1000 | S\_TI16\_MAX |  |
| 0x1001 | S\_REGISTER\_ST | Register variable |
| 0x1002 | S\_CONSTANT\_ST | Constant symbol |
| 0x1003 | S\_UDT\_ST | User defined type |
| 0x1004 | S\_COBOLUDT\_ST | Special UDT for cobol that does not symbol pack |
| 0x1005 | S\_MANYREG\_ST | Multiple register variable |
| 0x1006 | S\_BPREL32\_ST | BP relative |
| 0x1007 | S\_LDATA32\_ST | Module-local symbol |
| 0x1008 | S\_GDATA32\_ST | Global data symbol |
| 0x1009 | S\_PUB32\_ST | Public symbol (CV internal reserved) |
| 0x100a | S\_LPROC32\_ST | Local procedure start |
| 0x100b | S\_GPROC32\_ST | Global procedure start |
| 0x100c | S\_VFTABLE32 | Address of virtual function table |
| 0x100d | S\_REGREL32\_ST | Register relative address |
| 0x100e | S\_LTHREAD32\_ST | Local thread storage |
| 0x100f | S\_GTHREAD32\_ST | Global thread storage |
| 0x1010 | S\_LPROCMIPS\_ST | Local procedure start |
| 0x1011 | S\_GPROCMIPS\_ST | Global procedure start |
| 0x1012 | S\_FRAMEPROC | Extra frame and proc information |
| 0x1013 | S\_COMPILE2\_ST | Extended compile flags and info |
| 0x1014 | S\_MANYREG2\_ST | Multiple register variable |
| 0x1015 | S\_LPROCIA64\_ST | Local procedure start (IA64) |
| 0x1016 | S\_GPROCIA64\_ST | Global procedure start (IA64) |
| 0x1017 | S\_LOCALSLOT\_ST | Local IL sym with field for local slot index |
| 0x1018 | S\_PARAMSLOT\_ST | Local IL sym with field for parameter slot index |
| 0x1019 | S\_ANNOTATION | Annotation string literals |
| 0x101a | S\_GMANPROC\_ST | Global proc |
| 0x101b | S\_LMANPROC\_ST | Local proc |
| 0x101c | S\_RESERVED1 | Reserved |
| 0x101d | S\_RESERVED2 | Reserved |
| 0x101e | S\_RESERVED3 | Reserved |
| 0x101f | S\_RESERVED4 | Reserved |
| 0x1020 | S\_LMANDATA\_ST |  |
| 0x1021 | S\_GMANDATA\_ST |  |
| 0x1022 | S\_MANFRAMEREL\_ST |  |
| 0x1023 | S\_MANREGISTER\_ST |  |
| 0x1024 | S\_MANSLOT\_ST |  |
| 0x1025 | S\_MANMANYREG\_ST |  |
| 0x1026 | S\_MANREGREL\_ST |  |
| 0x1027 | S\_MANMANYREG2\_ST |  |
| 0x1028 | S\_MANTYPREF | Index for type referenced by name from metadata |
| 0x1029 | S\_UNAMESPACE\_ST | Using namespace |
| 0x1100 | S\_ST\_MAX | Starting point for SZ name symbols |
| 0x1101 | S\_OBJNAME | Path to object file name |
| 0x1102 | S\_THUNK32 | Thunk Start |
| 0x1103 | S\_BLOCK32 | Block start |
| 0x1104 | S\_WITH32 | With start |
| 0x1105 | S\_LABEL32 | Code label |
| 0x1106 | S\_REGISTER | Register variable |
| 0x1107 | S\_CONSTANT | Constant symbol |
| 0x1108 | S\_UDT | User defined type |
| 0x1109 | S\_COBOLUDT | Special UDT for cobol that does not symbol pack |
| 0x110a | S\_MANYREG | Multiple register variable |
| 0x110b | S\_BPREL32 | BP relative |
| 0x110c | S\_LDATA32 | Module-local symbol |
| 0x110d | S\_GDATA32 | Global data symbol |
| 0x110e | S\_PUB32 | Public symbol (CV internal reserved) |
| 0x110f | S\_LPROC32 | Local procedure start |
| 0x1110 | S\_GPROC32 | Global procedure start |
| 0x1111 | S\_REGREL32 | Register relative address |
| 0x1112 | S\_LTHREAD32 | Local thread storage |
| 0x1113 | S\_GTHREAD32 | Global thread storage |
| 0x1114 | S\_LPROCMIPS | Local procedure start |
| 0x1115 | S\_GPROCMIPS | Global procedure start |
| 0x1116 | S\_COMPILE2 | Extended compile flags and info |
| 0x1117 | S\_MANYREG2 | Multiple register variable |
| 0x1118 | S\_LPROCIA64 | Local procedure start (IA64) |
| 0x1119 | S\_GPROCIA64 | Global procedure start (IA64) |
| 0x111a | S\_LOCALSLOT | Local IL sym with field for local slot index |
| 0x111b | S\_PARAMSLOT | Local IL sym with field for parameter slot index |
| 0x111c | S\_LMANDATA |  |
| 0x111d | S\_GMANDATA |  |
| 0x111e | S\_MANFRAMEREL |  |
| 0x111f | S\_MANREGISTER |  |
| 0x1120 | S\_MANSLOT |  |
| 0x1121 | S\_MANMANYREG |  |
| 0x1122 | S\_MANREGREL |  |
| 0x1123 | S\_MANMANYREG2 |  |
| 0x1124 | S\_UNAMESPACE | Using namespace |
| 0x1125 | S\_PROCREF | Reference to a procedure |
| 0x1126 | S\_DATAREF | Reference to data |
| 0x1127 | S\_LPROCREF | Local Reference to a procedure |
| 0x1128 | S\_ANNOTATIONREF | Reference to an S\_ANNOTATION symbol |
| 0x1129 | S\_TOKENREF | Reference to one of the many MANPROCSYM's |
| 0x112a | S\_GMANPROC | Global proc |
| 0x112b | S\_LMANPROC | Local proc |
| 0x112c | S\_TRAMPOLINE | Trampoline thunks |
| 0x112d | S\_MANCONSTANT | Constants with metadata type info |
| 0x112e | S\_ATTR\_FRAMEREL | Relative to virtual frame ptr |
| 0x112f | S\_ATTR\_REGISTER | Stored in a register |
| 0x1130 | S\_ATTR\_REGREL | Relative to register (alternate frame ptr) |
| 0x1131 | S\_ATTR\_MANYREG | Stored in >1 register |
| 0x1132 | S\_SEPCODE |  |
| 0x1133 | S\_LOCAL\_2005 | Defines a local symbol in optimized code |
| 0x1134 | S\_DEFRANGE\_2005 | Defines a single range of addresses in which symbol can be evaluated |
| 0x1135 | S\_DEFRANGE2\_2005 | Defines ranges of addresses in which symbol can be evaluated |
| 0x1136 | S\_SECTION | A COFF section in a PE executable |
| 0x1137 | S\_COFFGROUP | A COFF group |
| 0x1138 | S\_EXPORT | A export |
| 0x1139 | S\_CALLSITEINFO | Indirect call site information |
| 0x113a | S\_FRAMECOOKIE | Security cookie information |
| 0x113b | S\_DISCARDED | Discarded by LINK /OPT:REF (experimental, see richards) |
| 0x113c | S\_COMPILE3 | Replacement for S\_COMPILE2 |
| 0x113d | S\_ENVBLOCK | Environment block split off from S\_COMPILE2 |
| 0x113e | S\_LOCAL | Defines a local symbol in optimized code |
| 0x113f | S\_DEFRANGE | Defines a single range of addresses in which symbol can be evaluated |
| 0x1140 | S\_DEFRANGE\_SUBFIELD | Ranges for a subfield |
| 0x1141 | S\_DEFRANGE\_REGISTER | Ranges for en-registered symbol |
| 0x1142 | S\_DEFRANGE\_FRAMEPOINTER\_REL | Range for stack symbol. |
| 0x1143 | S\_DEFRANGE\_SUBFIELD\_REGISTER | Ranges for en-registered field of symbol |
| 0x1144 | S\_DEFRANGE\_FRAMEPOINTER\_REL\_FULL\_SCOPE | Range for stack symbol span valid full scope of function body, gap might apply. |
| 0x1145 | S\_DEFRANGE\_REGISTER\_REL | Range for symbol address as register + offset. |
| 0x1146 | S\_LPROC32\_ID |  |
| 0x1147 | S\_GPROC32\_ID |  |
| 0x1148 | S\_LPROCMIPS\_ID |  |
| 0x1149 | S\_GPROCMIPS\_ID |  |
| 0x114a | S\_LPROCIA64\_ID |  |
| 0x114b | S\_GPROCIA64\_ID |  |
| 0x114c | S\_BUILDINFO | Build information. |
| 0x114d | S\_INLINESITE | Inlined function callsite. |
| 0x114e | S\_INLINESITE\_END |  |
| 0x114f | S\_PROC\_ID\_END |  |
| 0x1150 | S\_DEFRANGE\_HLSL |  |
| 0x1151 | S\_GDATA\_HLSL |  |
| 0x1152 | S\_LDATA\_HLSL |  |
| 0x1153 | S\_FILESTATIC |  |
| 0x1154 | S\_LOCAL\_DPC\_GROUPSHARED | DPC groupshared variable |
| 0x1155 | S\_LPROC32\_DPC | DPC local procedure start |
| 0x1156 | S\_LPROC32\_DPC\_ID |  |
| 0x1157 | S\_DEFRANGE\_DPC\_PTR\_TAG | DPC pointer tag definition range |
| 0x1158 | S\_DPC\_SYM\_TAG\_MAP | DPC pointer tag value to symbol record map |
| 0x1159 | S\_ARMSWITCHTABLE |  |
| 0x115a | S\_CALLEES |  |
| 0x115b | S\_CALLERS |  |
| 0x115c | S\_POGODATA |  |
| 0x115d | S\_INLINESITE2 | Extended inline site information |
| 0x115e | S\_HEAPALLOCSITE | Heap allocation site |
| 0x115f | S\_MOD\_TYPEREF | Only generated at link time |
| 0x1160 | S\_REF\_MINIPDB | Only generated at link time for mini PDB |
| 0x1161 | S\_PDBMAP | Only generated at link time for mini PDB |
| 0x1162 | S\_GDATA\_HLSL32 |  |
| 0x1163 | S\_LDATA\_HLSL32 |  |
| 0x1164 | S\_GDATA\_HLSL32\_EX |  |
| 0x1165 | S\_LDATA\_HLSL32\_EX |  |

## Non-modal Symbols

## Symbols for 16:16 Segmented Architectures

## Symbols for 16:32 Segmented Architectures

## Symbols for MIPS Architectures

## Symbols for CVPACK Optimization

# 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.

## Type Record

A type record has the following format:

| **Offset** | **Size** | **Field** | **Description** |
| --- | --- | --- | --- |
|  | 2 | length | Length in bytes of the following type string. This count does not include the length field. |
|  | \* | type string |  |

## Type String

## Leaf Indices Referenced from Symbols

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 |

Leaf indices for TD32 specific:

|  |  |
| --- | --- |
| 0x0030 | LF\_SET |
| 0x0031 | LF\_SUBRANGE |
| 0x0032 | LF\_PARRAY |
| 0x0033 | LF\_PSTRING |
| 0x0034 | LF\_CLOSURE |
| 0x0035 | LF\_PROPERTY |
| 0x0036 | LF\_LSTRING |
| 0x0037 | LF\_VARIANT |
| 0x0038 | LF\_CLASSREF |
| 0x0039 | LF\_WSTRING |

|  |  |
| --- | --- |
| 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

|  |  |
| --- | --- |
| *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 |

## Type Records Referenced from Type Records

## Subfields of Complex Lists

# Numeric Leaves

# Predefined Primitive Types

## Format of Reserved Types

## Primitive Type Listing

# Register Enumerations

## Intel 80x86/80x87 Architectures

## Motorola 68000 Architectures

## MIPS Architectures

# Symbol and Type Format for Microsoft Executables

## Introduction

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

## 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 NB(FB)xx, where xx is the version number and has the following meanings:

|  |  |
| --- | --- |
| NB00 | 32-bit CodeView debugger format. |
| NB01 | AIX debugger format. |
| NB02 | Linked by a Microsoft LINK, version 5.10, or equivalent OEM linker (16-bit CodeView debugger format). |
| NB03 | Not supported. |
| NB04 | 32-bit OS/2 PM debugger (IBM) format. |
| 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. |
| FB09 | Borland 32-bit symbol file signature for Delphi |
| FB0A | Borland 32-bit symbol file signature for C++ Builder |

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 |  |
| 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.

## 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.

| **Offset** | **Size** | **Field** | **Description** |
| --- | --- | --- | --- |
|  | 2 | cbDirHeader | Length of directory header. |
|  | 2 | cbDirEntry | Length of each directory entry. |
|  | 4 | cDir | Number of directory entries. |
|  | 4 | 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. |
|  | 4 | 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.

| **Offset** | **Size** | **Field** | **Description** |
| --- | --- | --- | --- |
|  | 2 | subsection | Subdirectory index. See the table below for a listing of the valid subsection indices. |
|  | 2 | 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. |
|  | 4 | lfo | Offset from the base address lfaBase. |
|  | 4 | 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:

|  |  |
| --- | --- |
| sstModule1 | Module 1 |
| . |  |
| sstModulen | Module *n* |
| sstTypes1 | Module 1 |
| sstPublics1 | Module 1 |
| sstSymbols1 | Module 1 |
| sstSrcModule1 | Module 1 |
| . |  |
| sstTypesn | Module *n* |
| sstPublicsn | Module *n* |
| sstSymbolsn | Module *n* |
| sstSrcModulen | 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 |
| sstModule1 | Module 1 |
| . |  |
| sstModulen | Module *n* |
| sstAlignSym1 | Module 1 |
| sstSrcModule1 | Module 1 |
| . |  |
| sstAlignSymn | Module *n* |
| sstSrcModulen | 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 |
| sstModule1 | Module 1 |
| . |  |
| sstModulen | Module *n* |
| sstAlignSym1 | Module 1 |
| sstSrcModule1 | Module 1 |
| . |  |
| sstAlignSymn | Module *n* |
| sstSrcModulen | Module *n* |
| sstGlobalPub | Global Publics |
| sstGlobalSym | Global Symbols |
| sstLibraries | Libraries |
| sstGlobalTypes | Global Types |
| sstStaticSym | Static Symbols |
| sstFileIndex | File Index |
| Directory |  |
| **NBxx** | Signature, if OMF executable |
| lfoBase | Offset of base, if OMF executable |

## 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 |
| 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.

| **Offset** | **Size** | **Field** | **Description** |
| --- | --- | --- | --- |
|  | 2 | ovlNumber | Overlay number. |
|  | 2 | iLib | Index into sstLibraries subsection if this module was linked from a library |
|  | 2 | cSeg | Count or number of code segments to which this module contributes. |
|  | 2 | 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:

| **Offset** | **Size** | **Field** | **Description** |
| --- | --- | --- | --- |
|  | 2 | Seg | Segment that this structure describes. |
|  | 2 | pad | Padding to maintain alignment This field is reserved for future use and must be emitted as zeroes. |
|  | 4 | offset | Offset in segment where the code starts. |
|  | 4 | 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.

| **Offset** | **Size** | **Field** | **Description** |
| --- | --- | --- | --- |
|  | 2/4 | 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. |
|  | 2 | seg | Segment index. |
|  | 2 | 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.

| **Offset** | **Size** | **Field** | **Description** |
| --- | --- | --- | --- |
|  | \* | name | Length-prefixed name of source file. |
|  | 2 | seg | Segment. |
|  | 2 | 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.

| **Offset** | **Size** | **Field** | **Description** |
| --- | --- | --- | --- |
|  | 2 | *cFile* | Number of source files contributing code to segments. |
|  | 2 | *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* |  |

### (0x0128) sstLibraries

### (0x0129) sstGlobalSym

### (0x012a) sstGlobalPub

### (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 nontrivial. 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

### (0x012d) sstSegMap

### (0x012e) sstSegName

### (0x012f) sstPreComp

### (0x0133) sstFileIndex

### (0x0134) sstStaticSym

## Hash table and sort table descriptions