SYM File Format
Version 3.4

Note

This document describes a file format that is still changing. Since we are not ready to commit to this format for all time, this document is for informational purposes only. Don't ship anything that depends on it without asking DTS first.

Despite the author's continuing threats of accuracy (however temporary it may be) and completeness there are still some remarkable rough edges here. Questions and constructive criticism are encouraged. If you find any glaring typos, inconsistencies, nebulousities or mistakes, please let me know.

Table of Contents

Revision History

Summary of Changes from MPW 3.0 to MPW 3.1

Nano-summary:

The changes described below make 'Version 3.1' Sym files incompatible with 'Version 3.0A1.2' Sym files. The 3.1 linker will generate only 3.1 Sym files. The 3.1 SADE (actually I think they call it the "1.1" SADE, just to be confusing) will read only 3.1 Sym files.

DISK_SYMBOL_HEADER_BLOCK (DSHB)

- The version string in the DSHB has changed to "`Version 3.1`" from "`VERSION 3.0A1.2`". Note the change from upper to lower case.

- The DTI_OBJECT_COUNT field in the DISK_TABLE_INFO structure is now a LONGINT (some of the 16-bit counts were overflowing).

STORAGE_CLASS_ADDRESS (SCA)

- A new storage kind, '`STORAGE_KIND_WITH`', has been added to support WITH and implied - WITH scopes in Pascal and object-oriented languages.

RESOURCE_TABLE_ENTRY (RTE)

- Added **rte_res_size** field, the size of the resource.

CONTAINED_VARIABLES_TABLE_ENTRY (CVTE)

- Significant changes for allowing "logical addresses". Please refer to the section on CVTEs for more information. Logical addresses are described in the OMF documentation for the LocalID2 record.

CONTAINED_STATEMENTS_TABLE_ENTRY (CSNTE)

- Removed the **csnte_statement_number** field. The **csnte_mte_index** field is now the first record in the record's statement entry variant.

Type Information

- The argument to the TTE typecode has been changed from a hard-coded 16-bit word to a normal type information scalar.

Summary of Changes from MPW 3.1 to MPW 3.2

- Version string changed to `"\pVersion 3.2"`.

- MODULES_TABLE_ENTRY
  - The fields `mte_def_fref` and `mte_def_end` have been eliminated.
  - fields `mte_csnte_idx_1` and `mte_csnte_idx_2` were changed to unsigned longs.

- `CONTAINED_STATEMENTS_TABLE_ENTRY`
  - The fields csnte_.mte_offset was changed to unsigned long.

Summary of Changes from MPW 3.2 to MPW 3.3
(fall, 1992)

- Version string changed to `"\pVersion 3.3"`.
- MODULES_TABLE_ENTRY
  - field `mte_cvte_index` changed from an unsigned short to an unsigned long

Summary of Changes from MPW 3.3 to MPW 3.4
(May, 1994)

- Version string changed to `"\pVersion 3.4"`.
- Constants `END_OF_LIST`, `FILE_NAME_INDEX`, `SOURCE_FILE_CHANGE`, and `MAXIMUM_LEGAL_INDEX` changed to `UNSIGNED LONG`
- FILE_REFERENCE
  - Field `fref_frte_index` extended to `UNSIGNED LONG`
- DISK_TABLE_INFO
  - Fields `dte_first_page` and `dte_page_count` extended to `UNSIGNED LONG`
- DISK_SYMBOL_HEADER_BLOCK
  - Fields `dshb_hash_page` and `dshb_root_mte` extended to `UNSIGNED LONG`
- RESOURCE_TABLE_ENTRY
  - Fields `rte_mte_first` and `rte_mte_last` extended to `UNSIGNED LONG`
- MODULES_TABLE_ENTRY
  - Fields `mte_parent`, `mte_cmte_index`, `mte_clte_index` and `mte_ctte_index` extended to `UNSIGNED LONG`
  - Field `mte_cvte_index` extended to `UNSIGNED LONG` (effective with Version 3.3 SYM format)
- CONTAINED_MODULES_TABLE_ENTRY
  - Fields `mte_index` and `cmte_end_of_list` extended to `UNSIGNED LONG`
- CONTAINED_VARIABLES_TABLE_ENTRY
  - Fields `change`, `tte_index`, and `cvte_end_of_list` extended to `UNSIGNED LONG`
- CONTAINED_LABELS_TABLE_ENTRY
  - Fields `change`, `mte_index`, and `clte_end_of_list` extended to `UNSIGNED LONG`
- CONTAINED_STATEMENTS_TABLE_ENTRY
  - Fields `change`, `mte_index`, and `csnte_end_of_list` extended to `UNSIGNED LONG`
- FILE_REFERENCE_TABLE_ENTRY
  - Fields `name_entry`, `mte_index`, and `frte_end_of_list` extended to `UNSIGNED LONG`
- FRTE_INDEX_TABLE_ENTRY
  - Fields `frte_index` and `fite_end_of_list` extended to `UNSIGNED LONG`
- CONTAINED_TYPES_TABLE_ENTRY
  - Fields `change`, `tte_index`, and `ctte_end_of_list` extended to `UNSIGNED LONG`
- NAME_TABLE_ENTRY
  - Additional special entry added to handle strings with length > 255 characters.

Introduction

The MPW linker produces a symbolic file which provides SYM file consumers with debugging information about a tool or application. The linker reads symbolic OMF records from the object files used to build the executable file and uses that symbolic information, along with the moral equivalent of a link map, to build the Sym file. SYM file consumers use the Sym file to display source information and access variables, types and labels.

Related Documents and Tools

DebugDataTypes.p (header file for Pascal)

DebugDataTypes.h (header file for C)

> **These are the actual interfaces used by SYM file consumers and producers; they contain some anachronisms, but still represent the ultimate arbiter for the constant and record definitions.**

MPW 3.0 Object Module Format (OMF Document)

> **This document describes the OMF records used to build the Sym file. Also includes information on typecodes, and a few notes on the symbolic OMF semantics.**

SADE Reference Manual

DumpSym

> **This is an MPW tool available from DTS. It dumps the contents of Sym files.**

A note about the 3.3 format

The version 3.3 SYM file format definition was never officially sanctioned by Apple, nor was it ever supported by an Apple product. Since it has been in widespread use for the last couple of years, though, I felt that we should bump the version number beyond it to help differentiate between it and this updated specification.

Abbreviations

The Sym file contains a number of structures that are referred to by abbreviations and acronyms. Here is a summary of terms used in this document:

Term	Expansion	Description
CLTE	Contained Labels Table Entry	Information about labels in a scope
CMTE	Contained Modules Table Entry	Information about nested scopes and modules
CONST	Constant pool	Storage for >4 byte constants (e.g. strings, floats)
CSNTE	Contained Statements Table Entry	Source location information
CTTE	Contained Types Table Entry	Information about types in a scope
CVTE	Contained Variables Table Entry	Information about variables in a scope
DSHB	Disk Symbol Header Block	Sym file header structure
DTI	Disk Table Info	Sym file table descriptor (contained in DSHB)
FITE	FRTE Index Table Entry	Occurence of a source file
FREF	File Reference	Reference to a source location by (FRTE, offset)
FRTE	File Reference Table Entry	Maps between module and source file location
HTE	Hash Table Entry	Hash table for NTE lookup
MTE	Module Table Entry	Describes a module or scope
NTE	Names Table Entry	Hashed name table entry
RTE	Resource Table Entry	Maps from a resource to a range of MTEs
SCA	Storage Class Address	Describes how to access an addressable entity
TINFO	Type Information	Actual type information, pointed to by TTEs
TTE	Type Table Entry	Index to type information, from a type number

Future Directions

The Sym file and OMF will provide support for object-oriented languages (C++ and Object Pascal). This will affect the OMF and Sym file format in as-yet undetermined ways (for instance, multiple inheritance means that an MTE can be in several scopes at once, not just a simple lexical scope).

In C++ variables can have non-block scope (where scope starts in the middle of a block), which the current Sym file structures don't handle efficiently. Similarly, the current structures don't gracefully handle variables which move from register to register.

There will be changes to make the Sym file information smaller (i.e. compress MTEs).

To maintain compatibility with the version 3.2 SYM definition, the original 3.2 structure definitions included in the interface files have been retained, but their names have been suffixed with "_v32". This was done to allow software components to support both the old and the new definitions, if they so wish. For those readers who wish to simply rebuild their component using the new interfaces to allow their component to support the new format, the new interfaces have had the original structure definitions updated to conform to the following descriptions.

During this period of transition, it is suggested that components dealing with SYM files support both the current 3.2 format as well as the new 3.4 format. To do so, the component should use the 3.4 format data structures internally, and translate to/from the previous format when necessary. For SYM file producers, an appropriate warning should be issued when the conversion of a 32-bit value to a 16-bit value would cause overflow of the 16-bit data field. The default with "-sym on" or "-sym full" should be the generation of version 3.4-format SYM files.

SYM file producers that are command-line driven should standardize on a common additional argument to the existing -sym option. I suggest "3.4", as in "-sym 3.4". "Off", "On", and "Full" are already in use to produce 3.2 format SYM files. "Big" is already in use by the MakeSym tool to allow it to use memory differently when producing a SYM file - therefore this argument should not be used to specify the production of the new SYM format file. To support future compatibility, I also suggest implementing a "-sym 3.2" option, to specify generation of 3.2-format SYM files.

Non-command-line driven producers can either supply some type of preference setting, or simply produce the new SYM format and hope that consumers will be updated in time to be able to read the new format.

SYM file consumers should be able to be implemented in such a way as to make the decision at runtime which SYM file format is being read, based on the contents of the version string embedded within the SYM file.

Constants and Common Records

This section describes some common constants and records that are used by data structures defined later in this document.

Types

In the Pascal declarations that follow, the UNSIGNED type is really an INTEGER that is treated as if it were unsigned. MPW Pascal does not support 16-bit unsigned values (at least, not well), and it is frequently necessary to "trick" the compiler into doing the right thing when manipulating such values.

```
TYPE
   UNSIGNED = INTEGER;
```

Symbol Scope

The following constants indicate the visibility of a module, label or variable (disregarding its lexical scope):

```
CONST
   SYMBOL_SCOPE_LOCAL   =  0; { not visible outside lexical scope }
   SYMBOL_SCOPE_GLOBAL  =  1; { exported, visible to other scopes }
```

Storage Classes

A STORAGE_CLASS_ADDRESS record describes how get to a variable. The record has the following fields:

```
STORAGE_CLASS_ADDRESS = RECORD
   SCA_KIND:   SignedByte; { distinguish local/value/reference }
   SCA_CLASS:  SignedByte; { how to get a base address         }
   SCA_OFFSET: LONGINT;    { offset from the base address      }
END;
```

The sca_kind field distinguishes local variables from call-by-reference and call-by-value parameters, and takes on one of the following values:

```
CONST
   STORAGE_KIND_LOCAL      =  0; { a simple local variable             }
   STORAGE_KIND_VALUE      =  1; { a call-by-value parameter           }
   STORAGE_KIND_REFERENCE  =  2; { a call-by-reference (VAR) parameter }
   STORAGE_KIND_WITH       =  3; { describes a "WITH" logical address  }
```

If a variable's kind is STORAGE_KIND_LOCAL **it is a simple local variable (probably A6-relative or static); likewise, if a variable's kind is** STORAGE_KIND_VALUE**, it is a call-by-value procedure or function parameter; in any case the variable is at the address determined by the sca_class field.**

If a variable's kind is STORAGE_KIND_REFERENCE**, the value at the address determined by the sca_class (see below) is a pointer to the actual value.**

If a variable's kind is STORAGE_KIND_WITH**, the debugger uses the variable's type information to construct logical addresses to named fields in the type. That is, named fields in the variable's type are treated as variables themselves.**

The sca_class field describes how to obtain a base address for the variable, and takes on one of the following values:

```
CONST
   STORAGE_CLASS_REGISTER     =  0;    { value is in a register         }
   STORAGE_CLASS_A5           =  1;    { relative to A5                 }
   STORAGE_CLASS_A6           =  2;    { relative to A6                 }
   STORAGE_CLASS_A7           =  3;    { relative to A7                 }
   STORAGE_CLASS_ABSOLUTE     =  4;    { relative to location zero      }
   STORAGE_CLASS_CONSTANT     =  5;    { this is the value itself       }
   STORAGE_CLASS_BIGCONSTANT  =  6;    { CONST pool index of the value  }
   STORAGE_CLASS_RESOURCE     =  99;   { the RTE index of a resource    }
```

The sca_offset field contains the offset of the variable from the base address. For registers, it indicates the register number (see the OMF document for information on register numbers). For global variables and local variables on a stack frame the offset field specifies an offset from an address register. For small constants, the it is the constant value itself. For big constants it is the CONST pool index of the constant.

For STORAGE_CLASS_RESOURCE **the sca_offset field is the RTE index of a resource that actually holds the data. However,** STORAGE_CLASS_RESOURCE **is not currently supported by any MPW language processors, the linker or the OMF (it's only mentioned here for completeness).**

Constants that Distinguish Variants

These constants are used to identify variants in lists of table entries. (Tables containing variant records are CMTE, CVTE, CSNTE, CLTE, CTTE, FRTE, and FITE.) The first field in the table entry is a 32-bit integer, which may take on one of the following values:

```
CONST
   END_OF_LIST          =  $FFFFFFFF;    { end of the list            }
   FILE_NAME_INDEX      =  $FFFFFFFE;    { indicates a file name      }
   SOURCE_FILE_CHANGE   =  $FFFFFFFE;    { indicates new source file  }
   {  anything else  }     { typically a table index  }
```

Since this document uses Pascal syntax for record declarations, *dummy constants* are used to distinguish the variant in the record declaration. Don't be mislead — the dummy's actual value does not matter, and in fact this schmutz does not appear in the C version of the DebugDataTypes header file.

The File Reference (FREF) Record

A file reference describes a source file location. The fields are:

```
FILE_REFERENCE = RECORD
   fref_frte_index:  LONGINT;   { File Reference Table index           }
   fref_offset:      LONGINT;   { absolute offset into the source file }
END;
```

The fref_frte_index field contains the index of a FRTE that is a FILE_NAME_INDEX variant, from which can be extracted the source file's name and modification date (see the FRTE definition below). If the fref_frte_index field is zero, there is no source file (because a table index of zero is a dummy).

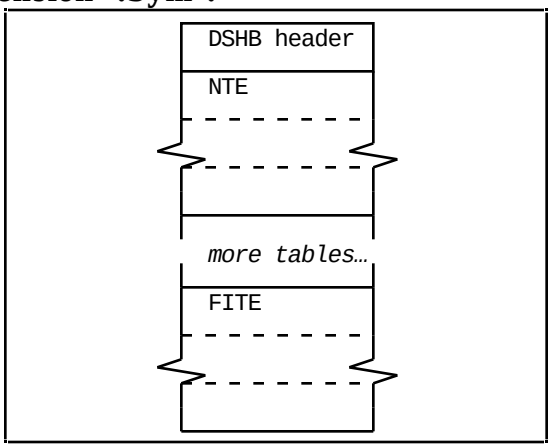
The fref_offset field contains an offset into the source file.

Sym File Layout

A Sym file is organized into tables; each table is made up of pages. Except for the DSHB (which occupies the first page) the tables may appear in any order. The location and size of each table is specified by the DSHB.

The page size is also specified by the DSHB. The linker enforces a minimum page size of 1K; it currently writes 2K pages. The table entries contained within pages cannot cross page boundaries. (EXCEPTION: the entries in the Name Table [NTE] can cross page boundaries.) Pages are numbered from zero.

The Sym file has a file type of 'MPSY'. The name of the Sym file is normally that of the executable, with the extension ".Sym".



The Disk Table Info (DTI) Record

DISK_TABLE_INFO (DTI) records appear only within the DSHB. They describe the location and size (in pages) of tables within the Sym file. The DTI fields are:

```
DISK_TABLE_INFO = RECORD
   dti_first_page:   LONGINT;   { table's first page       }
   dti_page_count:   LONGINT;   { # pages in the table     }
   dti_object_count: LONGINT;   { # objects in the table   }
END;
```

The dti_first_page field indicates the first page of the table.

The dti_page_count field indicates the number of pages in the table.

If the table is empty, dti_page_count is zero and dti_first_page is unpredictable.

The dti_object_count field contains the maximum legal index into the table. Object indices range from 0 to dti_object_count; the 0th entry is a zero-filled dummy that should never be accessed. In other words, an object index of K *means* K, not K-1.

Tables are indexed in two ways; by byte offset (a 32-bit number representing either the absolute file seek position or the table-relative seek position) and by table object index (a 32-bit number that is multiplied by the "sizeof" a table entry to yield a table-relative seek position). Here is a list of tables by "index flavor":

Tables Where the Index Is an Object Number	Tables Where the Index Is a "Seek Position"
FRTE	NTE (multiply index by 2)
CMTE	TINFO (index is file — not table — relative)
RTE	CONST (index is table-relative)
CVTE	
MTE	
CSNTE	
CLTE	
CTTE	
TTE	
FITE	

Many tables have variants; the first word of such records is a discriminator that can be a normal table index, or an "out of band" constant such as END_OF_LIST or SOURCE_FILE_CHANGE. For example, a portion of the FRTE list might look like:

Data	Pidgin FRTE Description
$**FFFFFFFE**, $00002110, $80272361	frte_name_entry = FILE_NAME_INDEX, frte_nte_index = 8464, frte_mod_date = $80272361;
$**0000002A**, $0000012C, $xxxxxxxx	frte_mte_index=42, frte_file_offset=300;
$**0000002B**, $00000177, $xxxxxxxx	frte_mte_index=43, frte_file_offset=375;
$**0000002C**, $000001D1, $xxxxxxxx	frte_mte_index=44, frte_file_offset=465;
$**FFFFFFFF**, $xxxxxxxx, $xxxxxxxx	END_OF_LIST;

The first entry is a FILE_NAME_INDEX **variant. The second through fourth variants are "normal" entries, a longword of each of which is unused. The last variant indicates the end of the list of FRTEs.**

Sym File Header (DSHB)

The DISK_SYMBOL_HEADER_BLOCK (DSHB) occupies the first page in the Sym file. It contains information about the physical layout of the Sym file, including the Sym file version (which will change in the future), the page size, the executable file's type and creator, and the locations of all of the tables. The fields are:

```
DISK_SYMBOL_HEADER_BLOCK = RECORD
   dshb_id:          STRING[31]; {* Version information          *}
   dshb_page_size:   INTEGER;    {* Size of the pages/blocks     *}
   dshb_hash_page:   LONGINT;    {* Disk page for the hash table *}
   dshb_root_mte:    LONGINT;    {* MTE index of the program root *}
   dshb_mod_date:    LONGINT;    {* executable's mod date        *}

   dshb_frte:     DISK_TABLE_INFO;  {* Per TABLE information     *}
   dshb_rte:      DISK_TABLE_INFO;
   dshb_mte:      DISK_TABLE_INFO;
   dshb_cmte:     DISK_TABLE_INFO;
   dshb_cvte:     DISK_TABLE_INFO;
   dshb_csnte:    DISK_TABLE_INFO;
   dshb_clte:     DISK_TABLE_INFO;
   dshb_ctte:     DISK_TABLE_INFO;
   dshb_tte:      DISK_TABLE_INFO;
   dshb_nte:      DISK_TABLE_INFO;
   dshb_tinfo:    DISK_TABLE_INFO;
   dshb_fite:     DISK_TABLE_INFO;
   dshb_const:    DISK_TABLE_INFO;

   dshb_file_creator:   OSType;  {* executable's creator       *}
   dshb_file_type:      OSType;  {* executable's file type     *}
END;
```

The dshb_id field is a Pascal string that indicates the Sym file version; currently `'Version 3.4'`**. Previous versions are identified with different version numbers within the string. See the interface files for specific values.**

The dsbh_page_size field indicates the size of a page. The MPW Linker writes 2K pages and enforces a minimum page size of 1024, but SYM file consumers should be able to deal with any page size large enough to hold the largest record. No symbolic record may cross a page boundary (with the exception of the Name Table); this is a potential problem only for the TINFO and CONST tables, whose entries could exceed an arbitrary fixed page size. [The linker's strategy is to pick a page size empirically determined to be "big enough" for most programs, and to supply a command line option (-pg *page_size_in_bytes*) so the page size can be increased if necessary].

The dshb_hash_page field indicates the page that contains the Hash Table Entries (HTEs). The size of the hash table (4K) is known, so a DTI is not necessary.

The dshb_root_mte field contains the MTE index of the root scope.

The dshb_mod_date field contains the modification date of the executable file. Likewise, the dshb_file_creator and dshb_file_type fields contain the executable file's type and creator.

The remaining fields, dshb_*xxxx*, are Disk Table Info (DTI) records (see above) which indicate the location and size of each table.

The Resource (RTE) Table

All code and data resides in resources, even global data resides in a dummy resource of type 'gbld' with id 0. Each RTE describes a resource (its type, id and name) and the range of MTEs the resource contains. The debugger uses RTEs to quickly translate a segment+offset address to a range of MTEs.

The fields are:

```
RESOURCE_TABLE_ENTRY = RECORD
   rte_ResType:      ResType;    { resource type                          }
   rte_res_number:   INTEGER;    { resource id                            }
   rte_nte_index:    LONGINT;    { the resource's name                    }
   rte_mte_first:    LONGINT;    { first MTE contained in the resource    }
   rte_mte_last:     LONGINT;    { last MTE contained in the resource     }
   rte_res_size:     LONGINT;    { the size of the resource               }
END;
```

The rte_ResType and rte_res_number fields specify the resource's type and id.

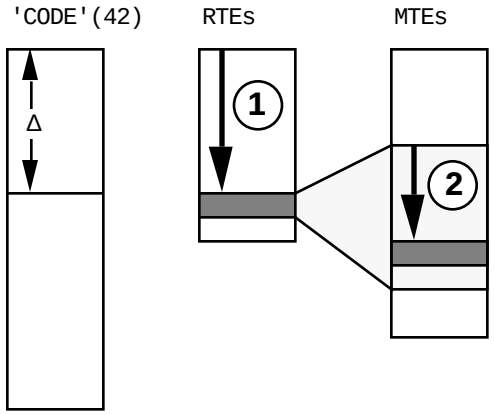
The rte_nte_index field specifies the NTE index of the resource's name.

The rte_mte_first and rte_mte_last fields specify the range of MTEs that belong to the resource. The MTE range is used to narrow the search for MTEs when it is necessary to translate a resource-and-offset to a scope or a source location, since the MTEs are ordered by segment offset.

The rte_res_size field specifies the size of the resource. Note that MTEs contained in a resource are not guaranteed to completely cover the resource (e.g. padding or data initialization information follow the last MTE in a resource).

To obtain an MTE index from a resource+offset, (1) search the RTE list for an RTE with a matching resource type and id, then (2) search the range of MTEs "owned" by that RTE until the first MTE that brackets the resource offset is found, using the mte_res_offset and mte_size fields.

To obtain the MTE whose scope corresponds to a resource+ offset, find the MTE that most narrowly contains the offset.



The Module (MTE) Table

An MTE represents a *scope* which lexically contains variables, labels, types, source statements and other MTEs. MTEs are used to define both *vacuous* scopes that are not associated with code (such as Pascal compilation UNITs, C source files, or C "curly-brace" scopes), and non-vacuous (or *meaty*) scopes which are associated with Module records and have addresses.

MTEs are grouped by segment, and ordered by segment offset.

The fields in an MTE are:

```
MODULES_TABLE_ENTRY = RECORD
   mte_rte_index:    UNSIGNED;          { which resource the MTE is in   }
   mte_res_offset:   LONGINT;           { offset into the resource       }
   mte_size:         LONGINT;           { size of the MTE                }
   mte_kind:         SignedByte;        { kind of MTE                    }
   mte_scope:        SignedByte;        { MTE's visibility               }
   mte_parent:       LONGINT;           { index of parent MTE            }
   mte_imp_fref:     FILE_REFERENCE;    { implementation source location }
   mte_imp_end:      LONGINT;           {   end of implementation        }
   mte_nte_index:    LONGINT;           { MTE's name                     }
   mte_cmte_index:   LONGINT;           { MTEs contained by this MTE     }
   mte_cvte_index:   LONGINT;           { variables contained by this MTE}
   mte_clte_index:   LONGINT;           { labels contained by this MTE   }
   mte_ctte_index:   LONGINT;           { types contained by this MTE    }
   mte_csnte_idx_1:  LONGINT;           { CSNTE index of first statement }
   mte_csnte_idx_2:  LONGINT;           { CSNTE index of last statement  }
END;
```

The mte_rte_index, mte_res_offset and mte_size fields specify the MTE's location and extent in a resource. If the MTE is not associated with any code, then mte_rte_index is zero.

The mte_kind field can take on the following values (corresponding to the values used by the OMF):

```
CONST
   MODULE_KIND_NONE        = 0;  { unknown module type                  }
   MODULE_KIND_PROGRAM     = 1;  { the root MTE has this kind           }
   MODULE_KIND_UNIT        = 2;  { transparent scope, must be GLOBAL    }
   MODULE_KIND_PROCEDURE   = 3;  { meaty MTE representing a procedure   }
   MODULE_KIND_FUNCTION    = 4;  { meaty MTE representing a function    }
   MODULE_KIND_BLOCK       = 6;  { unnamed scope                        }
```

The root MTE has MODULE_KIND_PROGRAM; compilation units have MODULE_KIND_UNIT, "meaty" MTEs associated with code have MODULE_KIND_PROCEDURE or MODULE_KIND_FUNCTION, and anonymous block scopes (such as curly-brace scopes in C) have MODULE_KIND_BLOCK. Code modules with no symbolic information have MODULE_KIND_NONE.

The mte_scope field contains SYMBOL_SCOPE_LOCAL or SYMBOL_SCOPE_GLOBAL, depending on the module's visibility. Modules of kind

UNIT are always SYMBOL_SCOPE_GLOBAL, and modules of kind BLOCK (which are generated from the OMF records BlockBegin and BlockEnd) are always SYMBOL_SCOPE_LOCAL.

The linker puts procedure or function MTEs with SYMBOL_SCOPE_GLOBAL into two CMTE lists; once in their parent (lexical) scope's list (so the correct contained variables, types, labels and MTEs are accessable), and again in the root MTE's list (so that searching for a global procedure or function is fast).

The mte_parent field specifies the MTE index of the MTE's parent lexical scope. The root MTE (which is generated by the linker) has a parent value of zero. UNITs and compilation units that specified a parent ID of zero in the OMF are hung from the root MTE.

The mte_imp_fref and mte_imp_end fields specify the source file and range of the MTE's implementation, take from an OMF ModuleBegin/Implementation record.

The mte_nte_index field contains the Names Table Entry index of the module's name, or zero if the MTE does not have a name.

The mte_cmte_index field contains the starting index of a list of the MTEs that the MTE lexically contains. By lexical scope we mean that the child MTEs have access to the variables, types, labels and modules that this module contains.

The mte_cvte_index, mte_clte_index and mte_ctte_index fields specify the starting index of a list of the variables, labels or types scoped in the MTE.

The mte_csnte_idx_1 field contains the index of the first source statement (CSNTE) that is considered part of the MTE; the mte_csnte_idx_2 field contains the index of the MTE's last source statement. Only MTEs associated with code have valid CSNTE indices. MTEs of MODULE_KIND_BLOCK do not have statements (their CSNTE indices are zero); instead it is necessary to locate the nearest parent scope with a non-zero CSNTE index. See the section Contained Statement Table for more information.

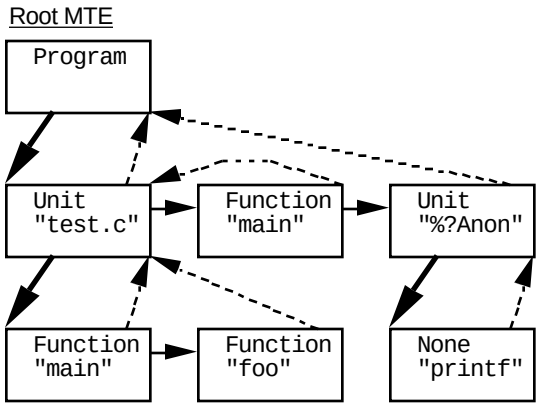
When the linker encounters a global module that does not have symbolic information, it constructs an MTE of MODULE_KIND_NONE for the module anyway, and places the MTE in the special "UNIT" scope called "%?Anon", which is the last child of the root MTE. (The mte_rte_index, mte_res_offset, mte_size, mte_kind, mte_scope, mte_parent and mte_nte_index fields are valid; all other fields will be zero). This allows users to reference library functions (e.g. `printf`) by name, even though there is no source, variable or type information for such modules.

For example, given the following C source in the file "test.c"**:**

```
/* file test.c */
main() {
   printf("Hi!\n");
   foo();
}

static int foo() {
   exit(42);
}
```

The MTE tree would look like:



(Parent-scope links are indicated by the dashed lines). The root MTE points to the Unit scope for the file, "test.c"**, which in turn contains an MTE for the functions** main **and** foo**. The function** main **is global, so it is contained again by the root MTE (note that there is only one MTE for** main**; it is simply referenced in two CMTE lists — see below). The function** printf **comes from a library that does not have symbolic information, so** printf **is contained in the Unit** "%?Anon"**, and has a module kind of "None".**

The Contained Modules (CMTE) Table

A CMTE list indicates the MTEs that are contained by an MTE. The CMTE table indicates nested scoping, and also allows efficient access to nested scopes (e.g. procedures and compilation units) by name. The table consists of terminated lists of MTE and NTE indices. The fields are:

```
CONTAINED_MODULES_TABLE_ENTRY = RECORD
   CASE INTEGER OF
   0: {DUMMY}
   (
      cmte_mte_index:   LONGINT;   { MTE that is contained    }
      cmte_nte_index:   LONGINT;   { the contained MTE's name }
   );

   END_OF_LIST:
   (
      cmte_end_of_list: LONGINT;   { indicates end of CMTE list  }
   );
END;
```

The cmte_mte_index field contains the index of an MTE that is contained. The cmte_nte_index field contains a copy of the MTE's nte index, so the MTE itself does not have to be accessed to get the MTE's name.

Each list is terminated with an END_OF_LIST variant; the value of cmte_end_of_list is END_OF_LIST.

The Contained Variables (CVTE) Table

A CVTE list indicates the variables that are contained by an MTE. A list consists of a SOURCE_FILE_CHANGE variant, followed by any number of CVTE entries and SOURCE_FILE_CHANGE variants, terminated by an END_OF_LIST variant. The order of the variable entries is not guaranteed.

```
CONST
   kCVTE_SCA         = 0;     { flags CVTE that has an SCA        }
   kCVTE_BIG_LA      = 127;   { flags an LA in Big Constant pool  }
   kCVTE_LA_MAX_SIZE = 13;    { max #bytes of LAs that fit        }

CONTAINED_VARIABLES_TABLE_ENTRY = RECORD
   CASE INTEGER OF
   SOURCE_FILE_CHANGE:
   (
      cvte_file_change: LONGINT;         { = SOURCE_FILE_CHANGE  }
      cvte_fref:        FILE_REFERENCE;  { the new source file   }
   );

   0: {≠ to SOURCE_FILE_CHANGE or END_OF_LIST, a variable entry:}
   (
      cvte_tte_index:   LONGINT;   { the variable's type            }
      cvte_nte_index:   LONGINT;   { the variable's name            }
      cvte_file_delta:  INTEGER;   { increment from previous source }

      cvte_scope: SignedByte;      { Scope of the variable          }
      cvte_la_size:  SignedByte;   { Size of logical address info   }

   CASE SignedByte {really 'cvte_la_size'} OF
     { Variable's address is described by a STORAGE_CLASS_ADDRESS }
     kCVTE_SCA:
     (
      cvte_location: STORAGE_CLASS_ADDRESS;
     );

     { Variable's address is described by a logical address }
     1:
     (
      cvte_la: ARRAY[0..kCVTE_LA_MAX_SIZE-2] OF SignedByte; { LA bytes}
      cvte_la_pad:   SignedByte; { pad, actually part of the array }
      cvte_la_kind:  SignedByte; { eqv. to cvte_location.sca_kind }
      cvte_la_xxxx:  SignedByte; { reserved for future use  }
     );

     { Variable's address is described by a BIG logical address   }
     kCVTE_BIG_LA:
     (
      cvte_big_la:   LONGINT;    { LA data offset in constant pool   }
      cvte_big_la_kind: SignedByte; { eqv. to cvte_location.sca_kind }
     );
   );

   END_OF_LIST:
   (
      cvte_end_of_list: LONGINT;   { indicates end of CVTE list  }
   );
END;
```

A SOURCE_FILE_CHANGE variant appears first in the list, or whenever the cvte_file_delta field exceeds the range of a signed integer.

VARIABLE_DEFINITION variants have the following fields:

The cvte_tte_index field contains the type of the variable. It is between 0 and 99 for primitive types, and greater than or equal to 100 for a type table (TTE) reference.

The cvte_nte_index field contains the NTE index of the variable's name.

The cvte_file_delta field contains a signed 16-bit offset from the previous source location, specifying the source location of the variable's declaration. If this field overflows, or a new source file is required, a SOURCE_FILE_CHANGE variant should be emitted.

The cvte_scope field contains SYMBOL_SCOPE_LOCAL or SYMBOL_SCOPE_GLOBAL, depending on the variable's visibility.

The value contained in the cvte_la_size field indicates the variant that follows:

cvte_la_size = kCVTE_SCA:
The cvte_location field contains a description of the "addressing mode" used to access the variable. See the section on Storage Classes for more information.

cvte_la_size ≤ kCVTE_LA_MAX_SIZE:
The cvte_la field contains an "in-situ" logical address that is up to kCVTE_LA_MAX_SIZE bytes long. See the documentation on the LocalID2 OMF record for a description of logical addresses.

The cvte_pad field is actually the last byte of the array cvte_la; it is necessary in the MPW Pascal declaration (but not in the C declaration) because Pascal does not permit odd-aligned fields following odd-sized arrays. The C declaration is much clearer.

The cvte_la_kind field contains the variable's kind, the equivalent of the sca_kind field in a STORAGE_CLASS_ADDRESS record.

The cvte_la_xxxx field is reserved for future use.

cvte_la_size = kCVTE_BIG_LA:
The cvte_big_la field contains the Big Constant Pool offset of a logical address whose size exceeds kCVTE_LA_MAX_SIZE. Like all big constants, the logical address is preceeded by a 16-bit size.

The cvte_big_la_kind field contains the variable's kind, the equivalent of the sca_kind field in a STORAGE_CLASS_ADDRESS record.

The Contained Labels (CLTE) Table

A CLTE list indicates the labels contained by an MTE. A list consists of a SOURCE_FILE_CHANGE variant, followed by any number of CLTE entries and SOURCE_FILE_CHANGE variants, terminated by an END_OF_LIST variant. The labels are sorted by source location.

```
CONTAINED_LABELS_TABLE_ENTRY = RECORD
   CASE INTEGER OF
   SOURCE_FILE_CHANGE:
   (
      clte_file_change: LONGINT;   { = SOURCE_FILE_CHANGE          }
      clte_fref:  FILE_REFERENCE;  { the new source file           }
   );

   0: {≠ to SOURCE_FILE_CHANGE or END_OF_LIST, a label entry:       }
   (
      clte_mte_index:   LONGINT;   { the module the label is in    }
      clte_mte_offset:  LONGINT;   { the label's offset in the MTE }
      clte_nte_index:   LONGINT;   { the label's name              }
      clte_file_delta:  INTEGER;   { increment from previous source }
      clte_scope:       INTEGER;   { scope of the label            }
   );

   END_OF_LIST:
   (
      clte_end_of_list: LONGINT;   { indicates end of CLTE list    }
   );
END;
```

A SOURCE_FILE_CHANGE variant appears first in the list, whenever the clte_file_delta field exceeds the range of a signed integer, or when the actual source file changes.

Label entries have the following fields:

> **The clte_mte_index field contains the index of the MTE the label appears in (lexical scoping).**
>
> **The clte_mte_offset field contains the label's offset into the MTE. (Note: This has always [or, at least since the 3.1 definition] been defined here in the specification as a LONGINT. The interface files have historically defined it as an UNSIGNED SHORT. As of this 3.4 definition, the interfaces are being corrected.)**
>
> **The clte_nte_index field contains the NTE index of the label's name.**
>
> **The clte_file_delta field contains a signed 16-bit offset from the previous source location, specifying the source location of the label's declaration. If this field overflows, or a new source file is required, a SOURCE_FILE_CHANGE variant should be emitted.**
>
> **The clte_scope field contains SYMBOL_SCOPE_LOCAL or SYMBOL_SCOPE_GLOBAL, depending on the label's visibility.**

Labels that are SYMBOL_SCOPE_GLOBAL are doubly contained; once in the lexical scope (for access to variables, types, labels and MTEs), and again in the root MTE's CLTE list, for efficient access by name.

The Contained Statement Table (CSNTE)

The Contained Statement Table is used to map from code to source location and back again. Each "meaty" MTE has an associated CSNTE list. Any vacuous (scope-only) MTEs within a meaty MTE do not have valid CSNTE indices, and it is necessary to locate the parent "meaty" MTE in order to do statement mapping (see below).

A CSNTE entry takes the form:

```
CONTAINED_STATEMENTS_TABLE_ENTRY = RECORD
   CASE INTEGER OF
   SOURCE_FILE_CHANGE:
   (
      csnte_file_change:   LONGINT;   { = SOURCE_FILE_CHANGE  }
      csnte_fref:    FILE_REFERENCE;   { the new source file   }
   );

   0: {≠ to SOURCE_FILE_CHANGE or END_OF_LIST, a statement entry:}
   (
      csnte_mte_index:  LONGINT;   { the MTE the statement is in }
      csnte_file_delta: INTEGER;    { delta from previous src loc }
      csnte_mte_offset: LONGINT;   { code location, offset into MTE }
   );

   END_OF_LIST:
   (
      csnte_end_of_list:   LONGINT;   { indicates end of stmt list  }
   );
END;
```

A SOURCE_FILE_CHANGE variant appears first in the list, whenever the csnte_file_delta field exceeds the range of a signed integer, or when the actual source file changes. This variant establishes a source file and initial offset; the offset is advanced by subsequent statement entries.

Statement entries have the following fields:

The csnte_statement_number field contains the statement number for the SYM file consumer's use. This is currently just a number that is incremented by one for each CSNTE entry.

The csnte_file_delta field contains the statement's offset from the previous source location.

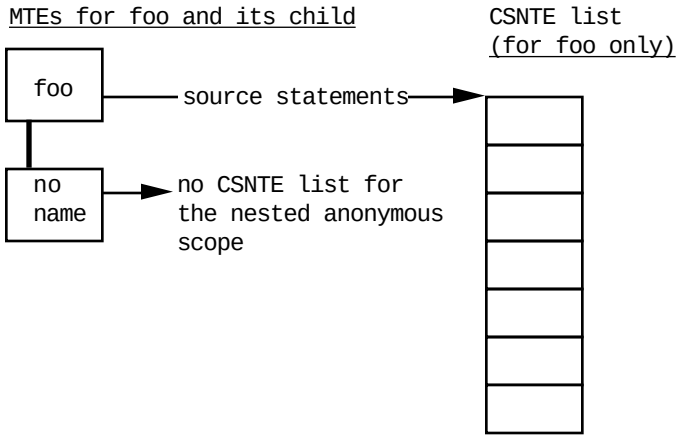
The csnte_mte_index and csnte_mte_offset fields specify the statement's code location (an MTE and an offset into the MTE). In some cases this delta can be zero; for selection purposes SADE treats statements whose code deltas are zero as part of the following statement. (Note: the csnte_mte_offset had previously been defined as an UNSIGNED by this document, but has been defined as UNSIGNED LONG in the interfaces. It should really be a 32-bit value, so this document has now been corrected.)

An END_OF_LIST variant follows the last statement belonging to the meaty MTE.

For example, given the following source code:

```
void foo()
{
   statements…
   {
      statements in "curly brace" scope…
   }
   more statements…
}
```

The MTE and CSNTE relationship would be something like:



If the debugger finds itself in a statement-less scope (the "curly-brace" scope in this example), it must chase the chain of parent MTEs, looking for an MTE with statements. If the search encounters the root MTE, the scope does not have statement information.

Sometimes compilers need to emit statements with code deltas of zero. For example:

```
if (spanishInquisition)
      return;
```

```
TST.W spanishInquisition
BNE.W some_RTS_instruction
```

The return statement must be considered part of the if because there is no program location that represents only the "return" state; it's impossible to break on the return statement without examining the processor's flags.

SADE treats consecutive statements with zero code deltas as a single statement, because the statements look like they have the same address.

The Name (NTE) Table

The name table is a list of Pascal strings that (for historical reasons) start on word boundaries. NTE indices are table-relative word offsets (i.e. the index must be multiplied by two to get a byte offset from the base of the table). The order of the strings is by hash chain; clumps of strings are terminated by a magic symbol of length 1 whose text is the character zero. Null strings (strings with length zero) indicate the end of a page (the chain is continued on the next page). Each NTE entry is distinct (there are no duplicate strings in the table).

Starting with version 3.4, each Pascal string in the table is also terminated with a NULL byte, thereby making it a C string as well. This was done to make it easier to implement a SYM file consumer in C and/or C++.

The first valid NTE index is 1, so the first two bytes of the NTE table are unused.

To get from an NTE index to a (page, page_offset) pair, do the following computation:

```
page_number := (NTE_index * 2) / DSHB.dshb_page_size +
   DSHB.dshb_nte.dti_first_page;
page_offset := (NTE_index * 2) MOD DSHB.dshb_page_size;
```

Starting with version 3.4, strings in the Name Table are limited to 254 characters each. This was done so that a length byte of 255 can signal a special entry in the table. This special entry (nte_ext_type = 0) is used to store strings with lengths greater than 254 characters (the historical limitation of Pascal strings). This special entry has the following format:

```
NTE_EXT_ENTRY = RECORD
   nte_ext_magic:    Byte;    {* Must = 255 ($FF)                    *}
   nte_ext_type:     Byte;    {* Type of extension, currently $00    *}
   nte_ext_length:   UNSIGNED;      {* the string length *}
   nte_ext_text:     PACKED ARRAY[0..1] OF SignedByte;
                               {* the string contents *}
   nte_ext_term:     SignedByte;    {* terminating NULL byte *}
END;
```

The nte_ext_magic field must be set to 255 ($FF), and is used as a semaphore to the SYM file consumer that this particular entry in the Name Table is actually a special extended Name Table entry.

The nte_ext_type field is used to identify which type of extension entry this particular entry is. For the 3.4 SYM format, this byte must be 0, which means that the entry is a string with length > 255 characters.

The nte_ext_length field contains the unsigned, 16-bit length of the name string. (This is analogous to the Pascal string length byte.)

The nte_ext_text field is an unbounded array used to hold the contents of the name string.

The nte_ext_term field is the null terminator which allows the string to be referenced as a C-style string. (Note that this field does not exist in the C header file, since the null terminator is implicitly defined in normal language usage.)

The Hash (HTE) Table

The hash table is shared between the linker and the debugger. The hash table consists of 1024 longword entries. Each entry contains the NTE index of a head of a chain of names in the NTE table; the chain is terminated by a magic symbol (see the section on NTEs).

The hash function is:

```
HashSize EQU   1024  ; must be a power of 2
;
;    Hash a Pascal string to a value in the range 0 … HashSize-1
;
;  Passed:   A0 -> Pascal string
;  Result:   D0.W = hash table index
;  Destroys: A0, D1
;
      CLR.W    D0             ; result := 0
      CLR.W    D1             ;
      MOVE.B   (A0)+,D1       ; D1 = length of string
      BEQ.S    @1             ; if length is zero, we're done
      MOVE.B   D1,D0          ; D1 = DBRA length of string
      SUBQ.W   #1,D1          ;
      ASL.W    #8,D0          ; length goes in high byte
@2:   ROR.B    #3,D0          ;   rotate current value
      ADD.B    (A0)+,D0       ;   add next character
      DBRA     D1,@2          ; (for all characters)
@1:   AND.W    #HashSize-1,D0 ; range for hash table index
```

The File Reference (FRTE) Table

The File Reference Table is used to map from a source file and source file offset to an MTE. Each source file has a FRTE list, which consists of MTE indices ordered by MTE source file offset. The fields in a FRTE are:

```
FILE_REFERENCE_TABLE_ENTRY = RECORD
   CASE INTEGER OF
      FILE_NAME_INDEX:
      (
         frte_name_entry:  LONGINT;   { = FILE_NAME_INDEX           }
         frte_nte_index:   LONGINT;   { name of the source file     }
         frte_mod_date:    LONGINT;   { the source file's mod date  }
      );

      0: {≠ FILE_NAME_INDEX and ≠ END_OF_LIST, a FRTE entry:   }
      (
         frte_mte_index:   LONGINT;   { the MTE's index             }
         frte_file_offset: LONGINT;   { the MTE's source file offset}
      );

      END_OF_LIST:
      (
         frte_end_of_list: LONGINT;   { = END_OF_LIST   }
      );
END;
```

Each source file's FRTE list starts with a FILE_NAME_INDEX variant. The frte_name_entry field indicates the variant and contains the constant FILE_NAME_INDEX. The frte_nte_index field contains the NTE index of the source file's name. The frte_mod_date field contains the source file's modification date, as indicated by the OMF.

Zero or more MTE references follow the FILE_NAME_INDEX variant. The frte_mte_index field contains the index of the MTE that begins at the source location specified by the frte_file_offset field. These MTE reference entries are ordered by increasing frte_file_offset.

Each source file's FRTE list is terminated with an END_OF_LIST variant.

The FRTE Index (FITE) Table

The FITE table provides quick access to the FRTE list by source file name. The order of the FITEs is not guaranteed (they are not sorted). The fields are:

```
FRTE_INDEX_TABLE_ENTRY = RECORD
   CASE INTEGER OF
      0: {DUMMY}
      (
         fite_frte_index:  LONGINT;   { index of src file's FRTE list  }
         fite_nte_index:   LONGINT;   { source file's name }
      );

      END_OF_LIST:
      (
         fite_end_of_list: LONGINT;   { = END_OF_LIST   }
      );
END;
```

For historical reasons the FITE table is terminated with an END_OF_LIST item, even though there is an accurate object count in the DTI.

The fite_frte_index field contains the index of the source file's FRTE list.

The fite_nte_index field contains the NTE index of the name of the source file corresponding to the FRTE list.

The Contained Types (CTTE) Table

A Contained Types list indicates the types that are contained by an MTE. A list consists of a SOURCE_FILE_CHANGE variant, followed by a list of type references and SOURCE_FILE_CHANGE variants, terminated by an END_OF_LIST variant. The fields are:

```
CONTAINED_TYPES_TABLE_ENTRY = RECORD
   CASE INTEGER OF
      SOURCE_FILE_CHANGE:
      (
         ctte_file_change: LONGINT;   { = SOURCE_FILE_CHANGE     }
         ctte_fref:  FILE_REFERENCE;  { type's source location   }
      );

      0: {≠ to SOURCE_FILE_CHANGE or END_OF_LIST, a type entry:}:
      (
         ctte_tte_index:   LONGINT;   { the type number }
         ctte_nte_index:   LONGINT;   { the type's name }
         ctte_file_delta:  INTEGER;   { delta from last source loc  }
      );

      END_OF_LIST:
      (
         ctte_end_of_list: LONGINT;   { = END_OF_LIST   }
      );
END;
```

A SOURCE_FILE_CHANGE variant appears first in the list, whenever a new source file is required, or when the ctte_file_delta field exceeds the range of an integer.

Type reference entries have the following fields:

> **The ctte_tte_index field contains a primitive type number (0 to 99) or a TTE table index (100 to SOURCE_FILE_CHANGE-1).**
>
> **The ctte_nte_index field contains the NTE table index of the type's name; the type name is also contained in the TINFO information, but is duplicated here for efficiency.**
>
> **The ctte_file_delta field contains the 16-bit signed source location offset from the previous definition.**

The Type Table (TTE)

The TTE Table is used to indirectly reference the actual type information. The TTE table is a vector of LONGINTs; each entry contains the file-relative seek position of the type information in the TINFO table. Since there is no type information for primitive types (TTE indices 0 to 99) the first TTE entry corresponds to type 100. That is, you should subtract 100 from the type index before indirecting through the TTE table. The dti_object_count field for the TTE table includes the first 100 "dummy" types in its count.

The Type Information (TINFO) Table

The TINFO table contains the actual type information. Each entry starts on a word boundary, and starts with a record of the form:

```
TYPE_INFORMATION_ENTRY = RECORD
   tinfo_nte_index:  LONGINT;       { the type's name }
   tinfo_physical_size: UNSIGNED;   { size of the type information   }

   CASE UNSIGNED OF
   0:
   (           { 16-bit logical size   }
      tinfo_short_logical_size:  UNSIGNED;
      tinfo_short_typecodes:  ARRAY [0..8000] OF SignedByte;
   );

   $8000:
   (           { 32-bit logical size   }
      tinfo_long_logical_size:   LONGINT;
      tinfo_long_typecodes:   ARRAY [0..8000] OF SignedByte;
   );
END;
```

The tinfo_nte_index field contains the NTE index of the type's name.

The tinfo_physical_size field contains the size (in bytes) of the typecodes that follow the header record.

If bit 15 of tinfo_physical_size is zero, the following word (tinfo_short_logical_size) contains the 16-bit logical ("sizeof") size of the type.

If bit 15 of tinfo_physical_size is one, the following longword (tinfo_long_logical_size) contains the 32-bit logical ("sizeof") size of the type.

Typecodes (described in the OMF document) follow the logical size field.

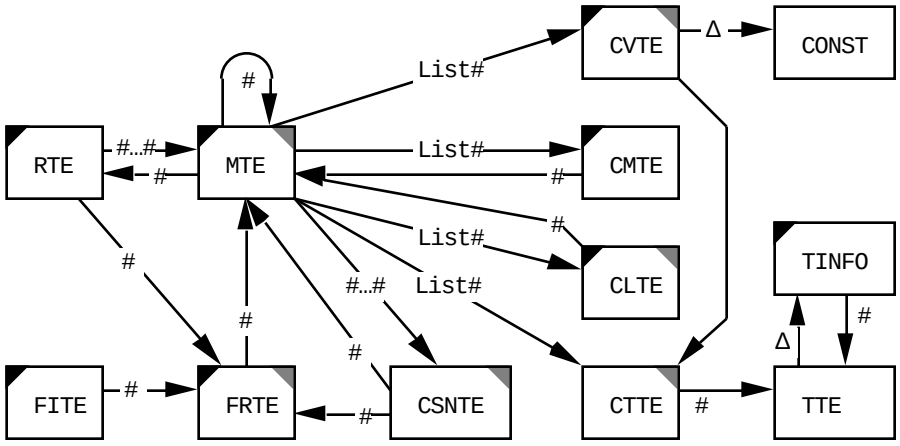
The Constant Pool (CONST)

The constant pool is used to contain values which take more than four bytes (a longword) to represent (e.g. floating point numbers and strings). Entries in the constant pool are referenced by a table-relative byte index (not by record number, since the entries vary in size). The format of a CONST pool entry is:

Entries start on word boundaries, and do not cross page boundaries.

Relationships Between the Tables

Here is a table that summarizes the links and dependencies between the tables:



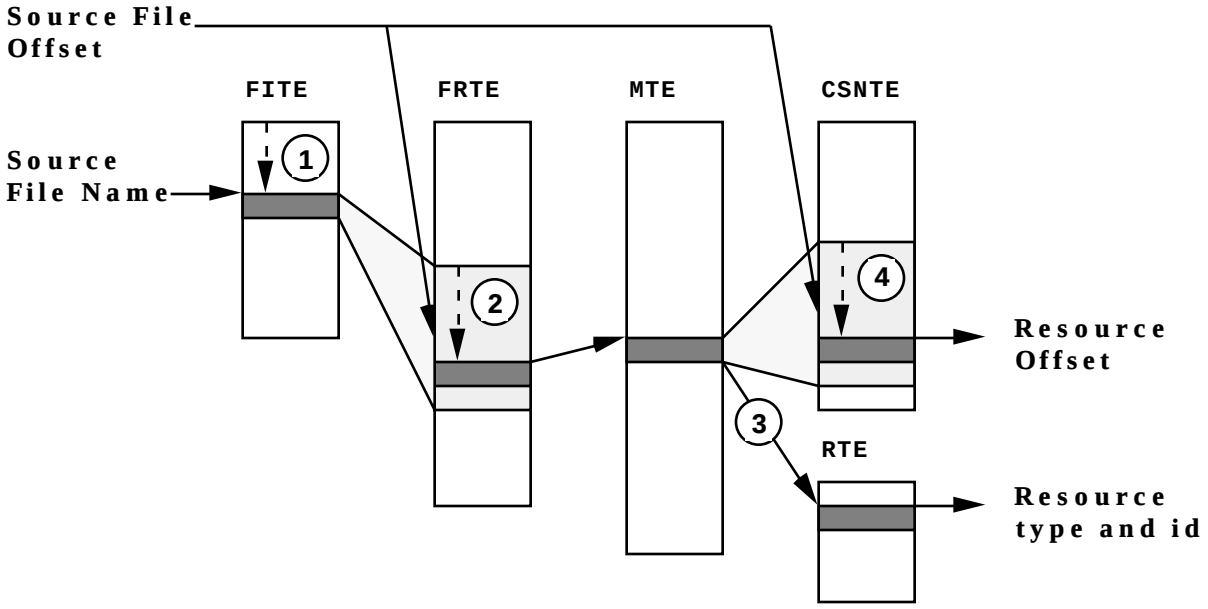
◤	has NTE index
#	table entry number
List#	table entry # of start of a list
Δ	file or table-relative seek index
#…#	a range of table indices
◥	has source file location (FRTE)

NTE and FRTE indices are so ubiquitous that for the most part they've been represented with symbols in the corner of each table. Not shown are the record variants that actually make the references.

Using the Sym File

SourceToAddr Strategy

Given a source file name and a file offset, the problem is to find the corresponding code address (a resource type, id and offset). A debugger could use the following algorithm:



[1] Find the source file

Search the FITE table linearly for a matching file name. If a matching FITE is found, it will contain the index of a FRTE list;

[2] Find the module

Search the FRTE list linearly for an entry which brackets the source offset. If a matching FRTE is found, it will contain the MTE index of the module it represents;

[3] Get the resource

The MTE contains an RTE index, which in turn gives the resource type and id;

[4] Find the statement

The MTE also specifies a range of statement (CSNTE) entries sorted by source file offset, with corresponding MTE (code) offsets. Search the CSNTE list linearly for the statement that is nearest to the source file offset and extract the corresponding MTE offset.

[5] Wrap up

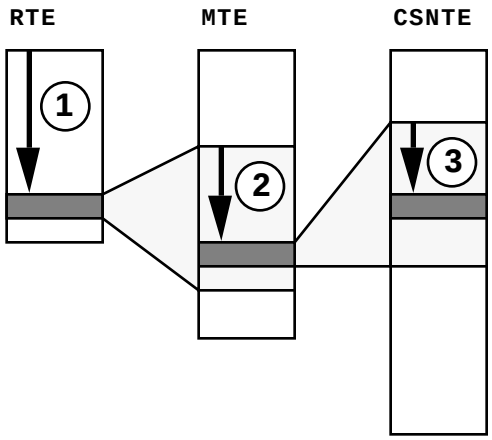
Adding the MTE's resource offset to the statement's MTE offset gives the final resource offset.

The basic idea is to narrow the range of the necessary linear searches; none of the tables can be searched with a binary algorithm, for instance, because source file change information is interspersed with other entries.

Translating from a source offset to a scope (MTE) is similar, except that the debugger must find the "narrowest" MTE whose definition or implementation FREF and 'end' field brackets the source location.

AddrToSource Strategy

Given a resource and resource offset, the problem is to find the corresponding statement. The following algorithm could be used:



[1] Find RTE

Search the RTE table linearly for an entry which matches the resource's type and id;

[2] Find MTE

The RTE "owns" a contiguous set of MTEs, ordered by resource offset. Search the RTE's range of MTEs for an MTE that contains the resource offset;

[3] Find the Statement

Search the MTE's list of contained statements for the statement nearest to the resource offset.

Users of the Sym File

The following table defines the (current) producers and consumers of .SYM files.

Producers	Consumers
Link	SADE
ILinkToSym	SourceBug
MakeSym	VoodooMonkey
Metrowerks	M.D. for PowerPC
Symantec C++ for PowerPC	M.D. for 68k
	Metrowerks Debugger
	The Debugger from Jasik
	DumpSym
	PrintProff