MPW 68K Object File Format

11/5/96

Introduction

This document describes the structure of MPW 68K object files. These files are created by various language processors (e.g. Asm, SC, Pascal) and the MPW Lib tool, and are read by the MPW linker, Lib, and the DumpObj tool.

MPW 68K object files have file type `'OBJ '` and creator `'MPS '`.

Note: The MPW linker validates only the file type, since applications other than MPW may create MPW-compatible object files.

An object file consists of a sequence of object file records. The records described in the remainder of this document are located in the data fork of the object file.

There are currently 22 types of object file records which are divided into two groups, code records and symbolic records.

Code Records

There are currently 12 types of code records:

- The first record in the file must be a **First** record**.**
- The last record in the file must be a **Last** record**.**
- One-byte **Pad** records are used to maintain word alignment.
- **Comment** records allow comments to be included in the file.
- **Dictionary** records associate names with unique IDs.
- **Module** records define code and data modules.
- **EntryPoint** records define entry points in code and data modules.
- **Size** records specify the sizes of modules.
- **Contents** records specify the contents of modules.

- **Reference** and **ComputedReference** records specify locations in modules that contain references to other modules or entry points.

- **XData** records define data modules specific to the CFM-68K runtime architecture.

Definitions

A **module** is a contiguous region of memory that contains code or static data. A module is the smallest unit of memory that is included or removed by the Linker. An **entry point** is a location (offset) within a module. (The module itself is treated as an entry point with an offset of 0.) A **segment** is a named collection of modules.

All modules, entry points, segments and symbolic records have a unique 16-bit ID that's assigned by the compiler, assembler or Lib. An **ID** is a file-relative number that identifies a specific module, entry point, segment or other entity within a single object file. If an ID has a name associated with it, that name is specified in a Dictionary record. If no Dictionary record exists for it, an ID is considered to be **anonymous**.

Modules and entry points may be local or external.

- A **local** module, entry point or segment can be referenced only from within the file where it is defined.

- An **external** module, entry point, or segment can be referenced from different files. In addition to an ID, each external module or entry point defined or referenced in an object file must also have a unique name (a string identifier) that identifies it across files.

Local modules and entries need not have unique names, and an external segment may have the same name as an external module or entry point.

Note: Although the names need not be unique, the IDs of these different objects must be unique. There must be multiple Dictionary entries even though the names are the same. If symbolic debugging records are generated, then ModuleBegin records which correspond to Module records must also share the same ID.

The CFM-68K runtime architecture defines a third level of visibility, called "exported". An **exported** module can be referenced not only from different files, but also from different fragments. Only external modules can be exported.

At any given point in an object file there can be one **current code module** and one **current data module**. The beginning of a new code or data module is indicated by a Module record. The current code and data modules are further defined by EntryPoint, Size, Contents, Reference, and ComputedReference records—these records can occur in any order after the Module record. In each of these intra-module records, a flag bit indicates whether the record refers to the current code module or the current data module, thereby permitting the interleaving of code and data records by compilers.

Code and data records may be arbitrarily interleaved. For example, the following record sequence declares three code modules and one data module; data module 2's scope extends until the next data module, across an arbitrary number of code modules.

```
Module(Code, ID=1)
Contents, Size, EntryPoint and Reference records for module 1
Module(Data, ID=2)
Contents, Size, EntryPoint and Reference records for modules 1 and 2
Module(Code, ID=3)
Contents, Size, EntryPoint and Reference records for modules 2 and 3
Module(Code, ID=4)
Contents, Size, EntryPoint and Reference records for modules 2 and 4
```

Note: A segment is declared implicitly, by specifying a segment ID in a code-type Module record.

Symbolic Records

There are currently 10 types of symbolic records:

- **Filename** records specify the compiler or assembler source files.
- **SourceStatement** records specify correspondence between generated code and source statements.
- **ModuleBegin** and **ModuleEnd** records declare named scopes (typically associated with UNITs, files or functions).
- **BlockBegin** and **BlockEnd** records declare "block" scopes contained within modules.
- **LocalID** and **LocalID2** records declare identifiers scoped within modules or blocks.
- **LocalLabel** records specify correspondence between generated code, source statements, and label identifiers.
- **LocalType** records define type information for local identifiers and functions.

Scoping of Symbolic Information

All symbolic records contain a parent ID field that specifies the scope to which the record applies. These records may be emitted in any order, in any part of the object file. Whether the code records and the symbolic records are interleaved, nested or completely separated is left to the discretion of the language implementor.

Executable objects (functions and procedures) are scoped lexically, while variables are scoped according to their visibility. To see why this is so, consider the following example in C (Pascal UNITs are similar):

```
/* example.c */
static int static_var;
int public_var;
static local_func() { … }
public_func() { … }
```

If `public_func` were contained by the root scope then it would be impossible to access any file-level static variables (e.g. `static_var`) from within a breakpoint in the function. Even though the Module record for `public_func` has its **external** bit set, the ModuleBegin for the function must specify the source file as the parent scope.

On the other hand, the variable `public_var` must be visible to procedures outside the file scope, so it is necessary to specify the root as its parent.

The records generated for the above example might be:

First(version=3)
Dictionary(1, "example.c")
Filename(1, *modificationDate*)

Dictionary(2, "example.c")
ModuleBegin(moduleID=2, parentID=0, fileID=1, kind=Unit)
  Dictionary(3, "static_var")
  LocalID(parentID=2, fileID=1, ID=3, *type, etc.*)
  Dictionary(4, "public_var")
  LocalID(parentID=0, fileID=1, ID=4, *type, etc.*)
  Dictionary(5, "local_func")
  ModuleBegin(moduleID=5, parentID=2, fileID=1, kind=Function)
  ModuleEnd(moduleID=5)
  Dictionary(6, "public_func")
  ModuleBegin(moduleID=6, parentID=2, fileID=1, kind=Function)
  ModuleEnd(moduleID=6)
ModuleEnd(moduleID=2, *fileOffset*)

Module(ID=5)
Contents, references and entry points for module 5
Likewise, source statement information
Module(ID=6, flags=external)
Contents, references and entry points for module 6
Likewise, source statement information
Last

The symbolic records may appear in any order (ModuleEnds preceding ModuleBegins, if necessary), interspersed with the code records (which *do* have order dependencies). There are a few items of interest in this example:

- A parent ID of 0 indicates the root.

- The ID referred to by a Filename record cannot also be referred to by a ModuleBegin record. Languages (such as C) which have file scoping will need to produce two Dictionary records that have the same name but different IDs; one for the file name, the other for the file-level scope.
- ModuleBegin records are not emitted for data modules; LocalID records are used instead.

ModuleBegin Implementation/Declaration Semantics

(The terms “Implementation” and “Declaration” refer to ModuleBegin records with the **isDeclaration** bit respectively 0 or 1).

If a module has only an Implementation, the linker assumes the Declaration and Implementation source locations are the same. The first Declaration encountered is used, except that a Declaration in the same object file as the first Implementation will override any previous Declaration. Any Declarations or Implementations following the first Implementation are ignored (anything contained in such an ignored scope is also ignored). It is legal to have a Declaration without an Implementation or a Module.

Module scope information is supplied solely by Implementation records; Declaration records may be nested, but the nesting is ignored since the Declaration information does not affect mapping between the source and the executable code.

Local variables (e.g. parameter variables) and types should be attached to either the Implementation or Declaration, but not both.

Object File Records

Each record type is represented by a diagram such as the following:

The first box illustrates the record. Each square block represents a byte. The first byte indicates the **record type**, in this case, 10. The **flags** byte is expanded in the second box. The **record size** is a signed, 16-bit integer that indicates the total length of the record (including the record type byte, flags

byte, and record size field). Hence, any one object file record is limited to 32767 bytes. (This is not a limit on the size of the module, because partial contents can be placed in several records.)

The second box represents the flag bits. In this example, they are interpreted as follows:

Bit	Meaning
0	0 indicates code, and 1 indicates data
1–2	must be 0
3	0 indicates short and a 1 indicates long
4–5	0 indicates 32 bits, 1 indicates 16 bits, and 2 indicates 8 bits
6	must be 0
7	1 indicates a difference computation

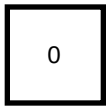
Note: All unspecified bits must be 0.

In the remainder of this document, names in bit-fields will be specified in numeric order. For instance, in the case of the text “short/long”, the tags “short” and “long” are understood to be respectively 0 and 1.

The records have been defined so that:

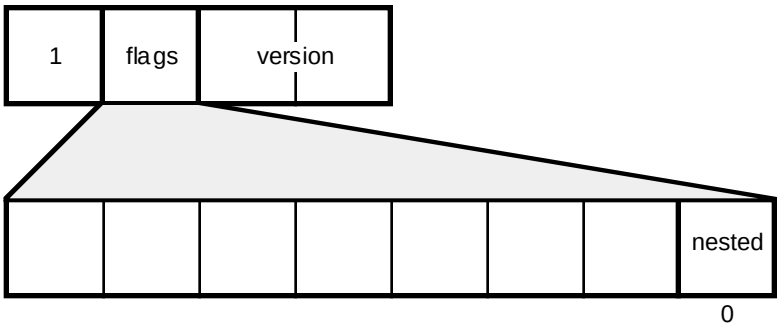
- All 16-bit and 32-bit fields are word-aligned in the file;
- Fixed-size records do not have a **record size** field;
- All variable-length records have a **record size** field.

Pad Record



A Pad record is a single byte that is always zero. A Pad record follows any record whose length is an odd number of bytes, in order to maintain word alignment. (Other than Pad records, all records are word-aligned.)

First Record



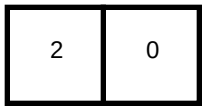
The first record in an object file must be a First record.

If the **nested** bit in the **flags** field is 1, then the linker interprets all references to undefined ID-name pairs as external references. If the nested bit is 0, the linker will try to match the name of an undefined symbol with a local name before treating the undefined symbol as external.

The **version** field contains a version number. The following values are defined:

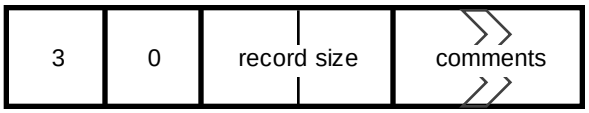
Value	Description
$0001	Object files generated by MPW 2.x.
$0002	Object files generated by MPW 3.0 and MPW 3.1.
	Object files generated by MPW 3.2 that do not contain symbolic records.
$0003	Object files generated by MPW 3.2 that do contain symbolic records.
	Classic 68K runtime object files generated by MPW 3.3 or later.
$0104	CFM-68K runtime object files generated by MPW 3.4 or later.

Last Record



The last record in an object file must be a Last record.

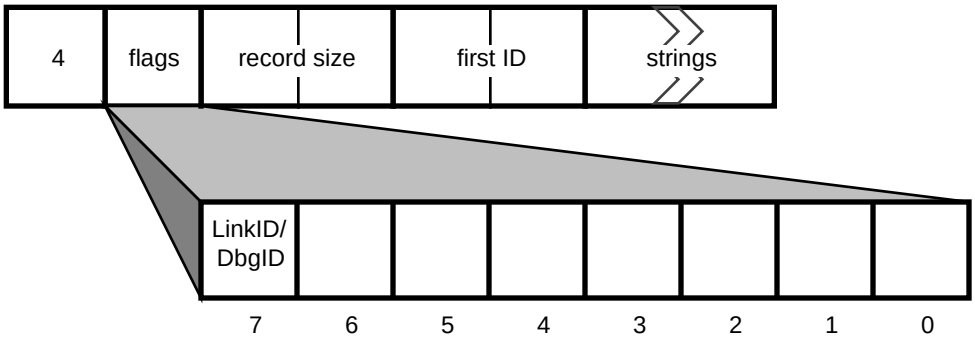
Comment Record



A Comment record allows comments to be included in an object file.

The **record size** field specifies the total number of bytes in the record.

Dictionary Record



A Dictionary record associates a name with an ID (or several names with several IDs).

At most one Dictionary record may appear for a given ID in a single object file.

The **record size** field specifies the total number of bytes in the record.

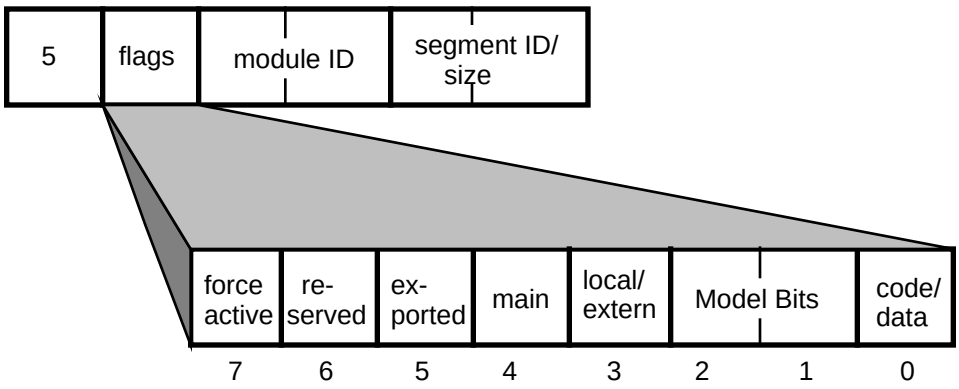
The **strings** field contains one or more names, each of which is preceded by an unsigned length byte.

The first name in the strings field is associated with the ID given in the **firstID** field. The second name is associated with **firstID** + 1, and so on.

The Dictionary record for an ID must appear before the Module, EntryPoint or XData record that defines the ID, but need not appear before Reference or ComputedReference records that refer to the ID. If an ID has no Dictionary record or has a name with a length of zero, it's considered anonymous.

The **LinkID/DbgID** flag bit is used to differentiate code identifiers from symbolic debugging identifiers.

Module Record



A Module record associates an ID with a module, and establishes that module as the "current" code or data module. All EntryPoint, Size, Contents, Reference and ComputedReference records combine to define a code or data module. If the module has a name, the Dictionary record for the **module ID** must appear before the Module record.

Modules may contain either **code** or **data**:

- For code modules, the **segment ID** field specifies the segment in which the code is placed. Segments may be named or anonymous. Named segments are treated as external; anonymous segments are local. (If the segment is named, the Dictionary record specifying the name must appear before the segment ID can be used in a Module record.)

- For data modules, a nonzero **size** field specifies the size of the module. In this case Size or Contents records are unnecessary. (The size of a module can also be specified by a Size record, or implicitly by the offset of the last byte in a Contents record.)

Bits 1 and 2 are the **Model Bits**. These 2 bits can have the following values:

Value	Situation
0	Universal (all modules except…)
1	16-bit Object Pascal objects
2	32-bit Object Pascal objects
3	reserved

All modules should be Universal unless the following is true:

- a module uses the sizeof a Pascal object
- a module references any field of a Pascal object
- a module creates a Pascal object

Modules may be either **local** or **external**. (Local modules may be anonymous.)

A code module flagged as **main** becomes the execution starting point of the program. A data module flagged as main becomes the main program data area, just below the location pointed to by A5. At most one main code module or entry point and one main data module may appear in an object file.

External data modules may also be **exported**. If a module is exported, the linker will not strip that module even if it is not referenced by any other module and is not the main module. Code modules and local data modules can not be exported. Functions are exported by marking their corresponding XVector exported (see the XData record description).

Note: Bit 5 should be set to 0 for classic 68K runtime object files.

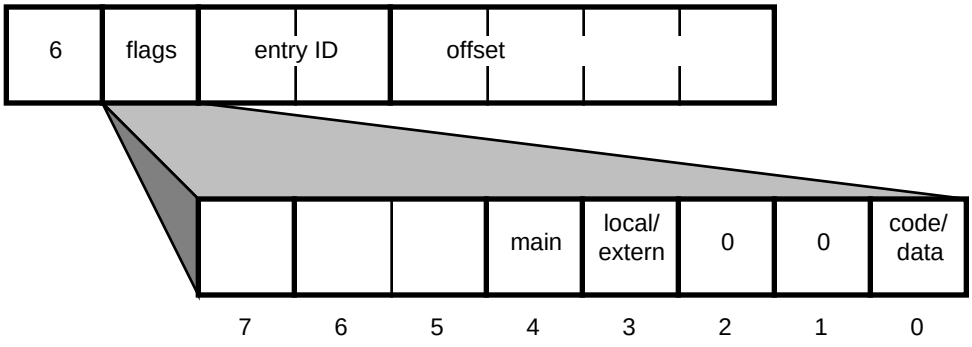
Bit 6 is reserved for internal use only.

If a code or data module has the **force active** bit set, then the linker will not strip that module even if it is not referenced by any other module and is not the main module.

References to a module are considered to be references to the first byte of the module.

Note: The linker ensures that modules are aligned on word or longword boundaries.

EntryPoint Record



An EntryPoint record associates an ID with an entry point. The entry point is in the current code or data module, as indicated by bit 0 of the **flags** field. If the entry point has a name, the Dictionary record for the **entry ID** must appear before the EntryPoint record.

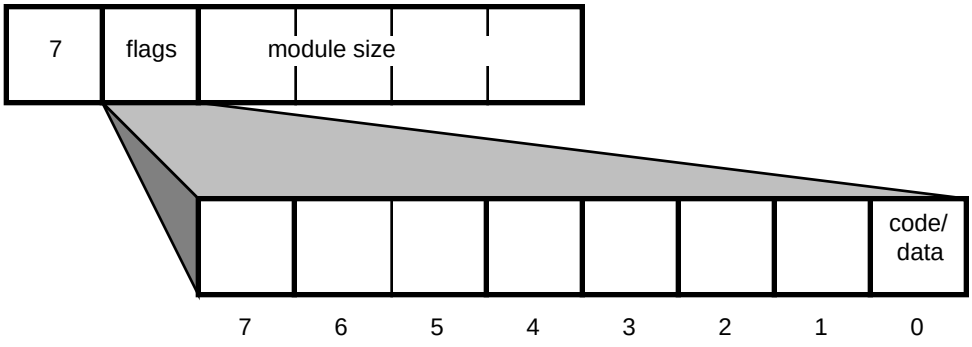
The **offset** field gives the byte offset of the entry point relative to the beginning of the module. The offset of an entry point may be outside the module (for example, a virtual base for an array).

An entry point may be defined for either a **code** or a **data** module.

Entry points may be either **local** or **external**. (Local entry points may be anonymous.)

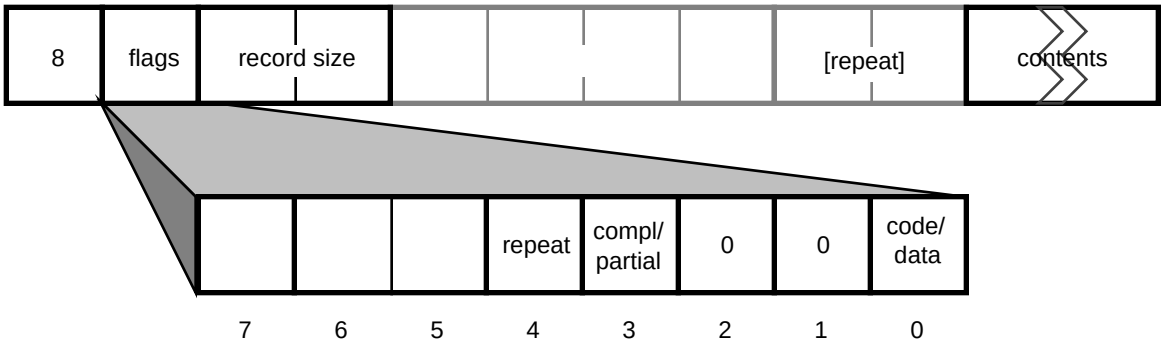
A code entry point flagged as **main** becomes the execution starting point of the program. At most one main code module or entry point may appear in an object file.

Size Record



A Size record specifies the size of the current code or data module in bytes. The size of a module may also be specified in a Contents record, or (for data modules) in the Module record. If more than one size is specified, the largest size given is taken as the size of the module.

Contents Record



Contents records specify the contents of the current code or data module.

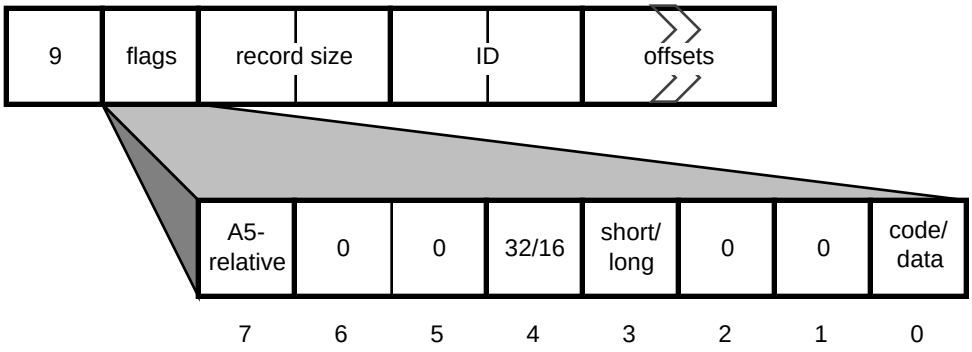
The **record size** field specifies the total number of bytes in the record.

Either **complete** or **partial** contents may be specified. If partial contents are specified, the first four bytes of the **contents** field specify the offset of the contents from the beginning of the module.

The **contents** may be either the bytes to be placed in the module, or a 2-byte **repeat** count followed by the bytes to be repeated. (If both an offset and a repeat count are specified, the offset comes first.)

Multiple Contents records per module are permitted, in any order. The offset of the last byte for which contents are specified determines the module's total size. (Size specifications may also appear in the Module record, and in Size records—if more than one size is specified, the largest size given is taken as the size of the module.)

Reference Record



A Reference record specifies a list of references to an ID. The references are from the current code or data module, and may be to either code or data.

The **record size** field specifies the total number of bytes in the record.

The **ID** field specifies the module or entry point being referenced.

The **offsets** field specifies a list of byte offsets from the beginning of the current code or data module. These offsets are either **short** (16 bits) or **long** (32 bits) and indicate the locations within the current code or data module that require modification. The modified locations may be either **32** or **16** bits long. The **A5-relative** flag indicates whether A5-relative addressing should be used.

Multiple references to the same or overlapping locations are permitted.

References fall into four categories: from code to code, from code to data, from data to code, and from data to data.

Code-to-Code References

- **A5-relative flag = 0**
  - **location size = 16 bits**

    The linker selects either PC-relative or A5-relative addressing. The immediately preceding 16-bit word must contain a `JMP`, `JSR`, `LEA` or `PEA` instruction, and is modified to indicate either PC-relative or A5-relative addressing. If the referenced code module or entry point and the current code module are in the same segment, the PC-relative offset of the referenced module or entry point is added to the contents of the specified locations. If they are in different segments, the A5-relative offset of the jump-table entry associated with the referenced module or entry point is added to the specified locations.

  - **location size = 32 bits**

    Same as above but run-time operations are also required. If the referenced code module or entry point and the current code module are in the same segment, the memory address of the segment must be added to the contents of the specified locations at run-time. If they are in different segments, the actual value of A5 must be added to the contents of the specified locations at run-time.

- **A5-relative flag = 1**
  - **location size = 16 bits**

    The A5-relative offset of the jump-table entry associated with the referenced code module or entry point is added to the contents of the specified locations (except when linking a stand-alone code resource). The linker forces PC-relative addressing if linking a stand-alone code resource. The immediately preceding 16-bit word must contain a `JMP`, `JSR`, `LEA` or `PEA` instruction, and is modified to indicate PC-relative addressing. The PC-relative offset of the referenced module or entry point is added to the contents of the specified locations.

  - **location size = 32 bits**

    The A5-relative offset of the jump-table entry associated with the referenced code module or entry point is added to the contents of the specified locations. No instruction editing is performed. (Note that this requires a run-time operation that adds the actual value of A5 to the contents of the specified locations.)

Code-to-Data References

- **A5-relative flag = 0**

  The A5-relative offset of the referenced data module or entry point is added to the contents of the specified locations, which must be 32 bits. (Note that this requires a run-time operation that adds the actual value of A5 to the contents of the specified locations.)

- **A5-relative flag = 1**

  The A5-relative offset of the referenced data module or entry point is added to the contents of the specified locations, which may be either 16 or 32 bits. No instruction editing is performed. (32-bit A5-relative addressing is available for the 68020, but not for the 68000.)

Data-to-Code References

- **A5-relative flag = 0**

  The A5-relative offset of the jump-table entry associated with the referenced code module or entry point is added to the contents of the specified locations, which must be 32 bits. (Note that this requires a run-time operation that adds the actual value of A5 to the contents of the specified locations.)

- **A5-relative flag = 1**

  The A5-relative offset of the jump-table entry associated with the referenced code module or entry point is added to the contents of the specified locations, which may be either 16 or 32 bits.

Data-to-Data References

- **A5-relative flag = 0**

  The A5-relative offset of the referenced data module or entry point is added to the contents of the specified locations, which must be 32 bits. (Note that this requires a run-time operation that adds the actual value of A5 to the contents of the specified locations.)

- **A5-relative flag = 1**

  The A5-relative offset of the referenced data module or entry point is added to the contents of the specified locations, which may be either 16 or 32 bits.

Source	Target	16/32 bits	A5 Rel.	Meaning
Code	Code	16	0	Edit instruction at link time to PC-relative if in same segment; otherwise add A5-offset of target's jump table entry.
		32	0	Same as above but requires run-time addition of either the segment's memory address or A5.
		16	1	Add A5-offset of target's jump table entry at link time. Force PC-relative for stand-alone code resources.
		32	1	Add A5-offset of target's jump table entry at link time. Requires run-time addition of A5.
Code	Data	16	0	Not allowed.
		32	0	Add A5-offset of target at link time. Requires run-time addition of A5.
		16/32	1	Add A5-offset of target at link time.
Data	Code	16	0	Not allowed.
		32	0	Add A5-offset of target's jump table entry at link time. Requires run-time addition of A5.
		16/32	1	Add A5-offset of target's jump table entry at link time.
Data	Data	16	0	Not allowed.
		32	0	Add A5-offset of target at link time. Requires run-time addition of A5.
		16/32	1	Add A5-offset of target at link time.

Note: Edited or forced instructions must be `JMP`, `JSR`, `LEA` or `PEA`.

Example

This MPW assembly language example exercises many of the possible modes of fixups. Note that further instruction editing is done by the linker (for instance, the "PC-relative" `JSR` to `PROC2` below will be forced A5-relative when the linker realizes that `PROC2` is in a different segment).

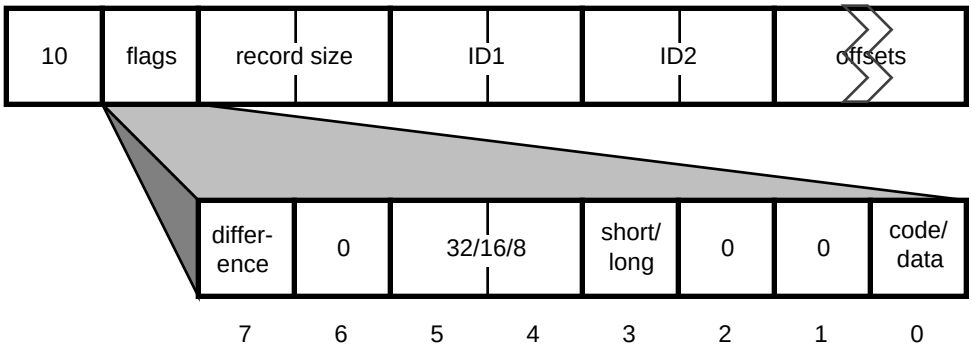
```
        SEG     'SEG1'
PROC    MAIN
        IMPORT  PROC1:CODE, PROC2:CODE, DATA1:DATA
        IMPORT  _DATAINIT   ;
        JSR     _DATAINIT   ; DO DATA INITIALIZATION

        CODEREFS FORCEJT    ; FORCE A5-RELATIVE:
        JSR     PROC1       ; CODE-TO-CODE, A5=1, SAME SEGMENT
        JSR     PROC2       ; CODE-TO-CODE, A5=1, DIFFERENT SEGMENT
        LEA     DATA1,A0    ; CODE-TO-DATA, A5=1

        CODEREFS NOFORCEJT  ; FORCE PC-RELATIVE:
        JSR     PROC2       ; CODE-TO-CODE, A5=0, DIFFERENT SEGMENT
                            ;   (FORCED A5-RELATIVE BY LINKER)
        JSR     PROC1       ; CODE-TO-CODE, A5=0, SAME SEGMENT
        ENDMAIN

DATA1   RECORD
        IMPORT  PROC1:CODE, DATA2:DATA

        DATAREFS RELATIVE   ; FORCE A5-RELATIVE
        DC.L    PROC1       ; DATA-TO-CODE, A5=1, 32-BIT THROUGH A5
        DC.W    PROC1       ; DATA-TO-CODE, A5=1, 16-BIT THROUGH A5
        DC.L    DATA1       ; DATA-TO-DATA, A5=1, 32-BIT THROUGH A5
        DC.W    DATA1       ; DATA-TO-DATA, A5=1, 16-BIT THROUGH A5

        DATAREFS ABSOLUTE   ; FORCE ABSOLUTE
        DC.L    PROC1       ; DATA-TO-CODE, A5=0, 32-BIT
        DC.L    DATA1       ; DATA-TO-DATA, A5=0, 32-BIT
        ENDR                ;

PROC1   PROC    EXPORT
        ENDPROC

        SEG     'SEG2'
PROC2   PROC    EXPORT
        ENDPROC
        END
```

C’s pointer initialization makes heavy use of the 32-bit Data-to-Code and Data-to-Data reference modes, as in:

```
extern int func();
int (*fp)() = func;      /* Data-to-Code, 32-bit, run-time addition of A5 */
int (**pfp) = &fp;       /* Data-to-Data, 32-bit, run-time addition of A5 */
```

The A5-relative Data-to-Code and Data-to-Data reference modes could be used for saving space if the application has a large number of pointers to data or code (a dispatch table, for instance).

ComputedReference Record



A ComputedReference record specifies a list of computed references based on two specified IDs. The references are from the current code or data module, and may be to either code or data.

The **record size** field specifies the total number of bytes in the record.

The **ID1** and **ID2** fields specify the modules or entry points being referenced. If **ID1** specifies a code module or entry point, **ID2** must also be a code module or entry point in the same segment. If **ID1** is a data module or entry point, **ID2** must also be a data module or entry point.

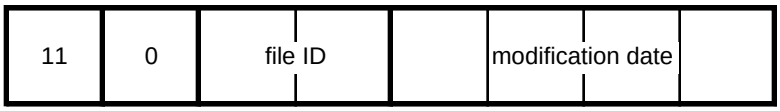
The only computation provided is **difference** (that is, bit 7 must be set).

The **offsets** field specifies a list of byte offsets from the beginning of the current code or data module. These offsets are either **short** (16 bits) or **long** (32 bits) and indicate the locations within the current code or data module that require modification. The modified locations may be either **32**, **16**, or **8** bits (a 0 in bits 4 and 5 indicates 32, 1 indicates 16, and 2 indicates 8). No instruction editing is performed.

The value of the address of **ID1** minus the address of **ID2** is added to the contents of the specified locations.

Multiple references to the same or overlapping locations are permitted.

Filename Record

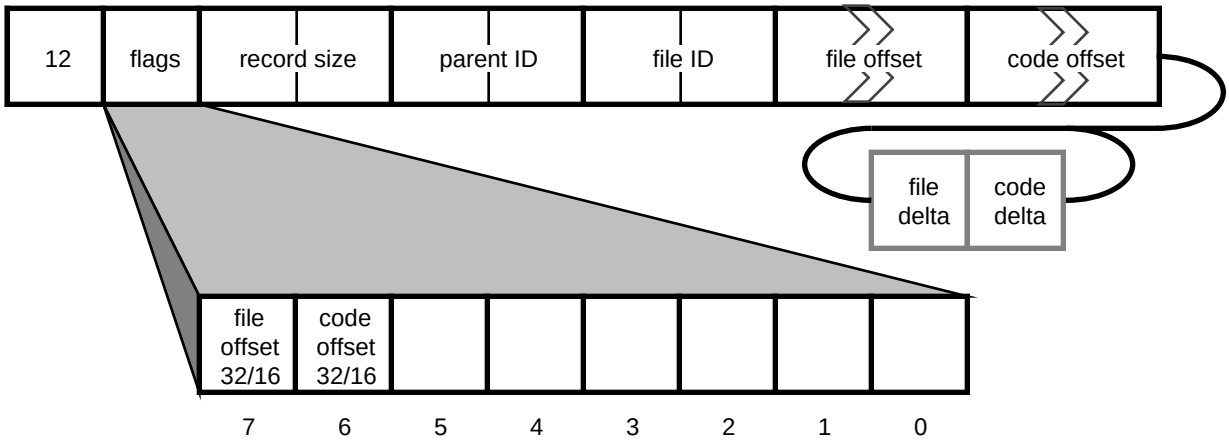


A Filename record specifies the source file from which the object file was generated. There can be multiple Filename records (i.e. one for the “primary” source file as well as one for each included file).

The **file ID** field contains the dictionary ID associated with the source file's name. A symbolic Dictionary record must exist for the file ID, i.e. files can't be anonymous. Although the file ID may be used by other records (e.g. SourceStatement records) prior to the appearance of the Filename record, a Filename record must exist for every file ID.

The **modification date** is the source file's modification date. This can be used by debuggers to help verify that the source file being displayed corresponds to the object file.

SourceStatement Record



A SourceStatement record specifies the correspondence between generated code and source file statements. The meaning of "statement" is defined by the language and the compiler writer.

The **record size** field specifies the total number of bytes in the record.

The **parent ID** specifies the scoping entity containing the statement.

The **file ID** specifies the source file.

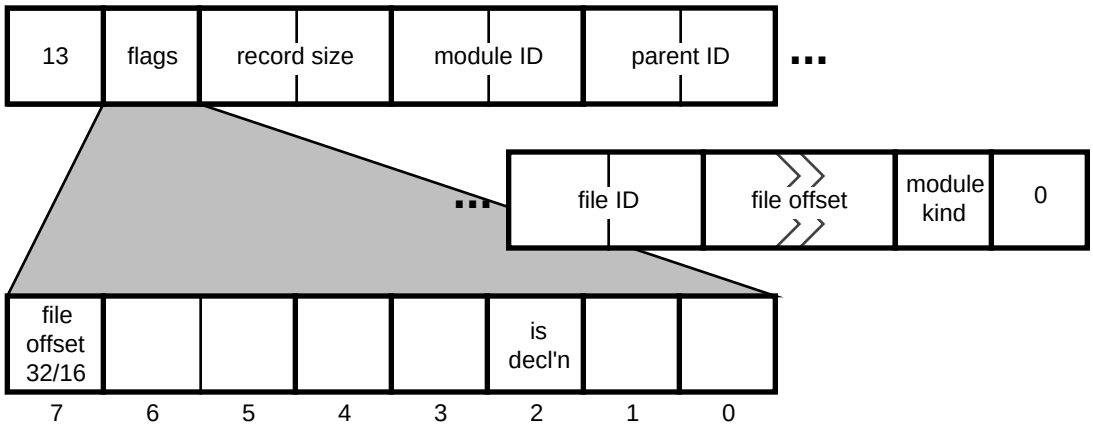
The **file offset** and **code offset** specify the source file offset and code or data module offset for the first statement specified by this source statement record. These fields may be either **16** or **32**-bit signed values. The file offset is the 0-relative byte offset in the file specified by the file ID. The code offset is the byte offset from the beginning of the code or data module for the first byte of code or data corresponding to the statement whose offset is specified by the file offset.

Each additional statement following the one specified by the file and code offset fields is specified by a **file delta** and **code delta** field. The deltas represent the *difference* between adjacent file and code offsets starting with the one specified by the file and code fields. These deltas are in the range from 0 to 255.

If the subsequent statement cannot be expressed with these offsets then a new SourceStatement record should be emitted with new beginning offsets.

All of the SourceStatement records for a module must be emitted in order of increasing source code offset.

ModuleBegin Record



The ModuleBegin record supplies symbolic information for a module.

The **is declaration** bit in the flags field provides a way to specify source location information for a module's declaration, if the module's declaration and implementation are separated in the source code (e.g. a Pascal `FORWARD` or `INTERFACE` declaration).

The **record size** field specifies the total number of bytes in the record.

The **module ID** associates an ID with the record. This ID must be the same value used in the Module record. It is through this ID that the connection is made between the debugger object file stream for a module and the standard object module stream.

Note: There doesn't have to be a Module record associated with the ModuleBegin record. If there isn't a Module record with the same ID then the scope declared by the ModuleBegin record is treated as a wrapper that doesn't have any code, but that can contain other symbolic objects. This is usually the case when the **module kind** field is **program** or **unit** .

The **parent ID** specifies the scoping entity that contains the module. An ID of 0 indicates global scope. This field is ignored if the **is declaration** bit is set (parentage information is not extracted from a declaration).

The **file ID** specifies the source file.

The **file offset** specifies the source file offset for the module header. This field may be either **16** or **32** bits and is the 0-relative byte offset in the file specified by the **file ID**.

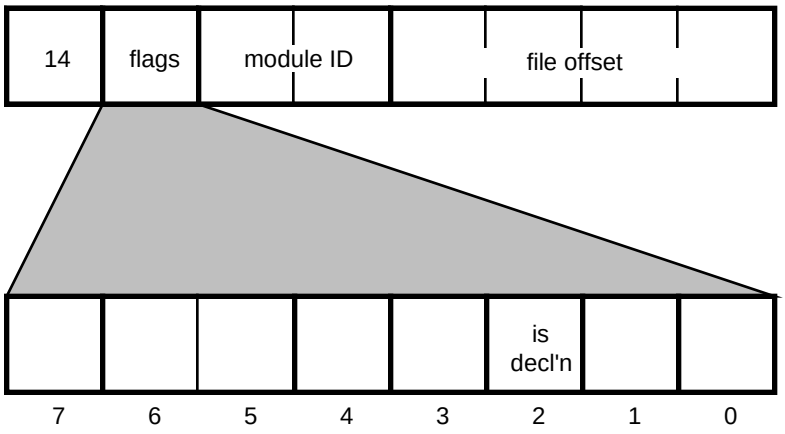
The **module kind** field indicates the kind of the module as follows:

Value	Module Kind
0	none
1	*reserved*
2	unit
3	procedure
4	function
5	data module
6…15	*reserved*

A Pascal program's program level should be treated as a **unit** (module kind 2).

The byte following the **module kind** field is reserved.

ModuleEnd Record

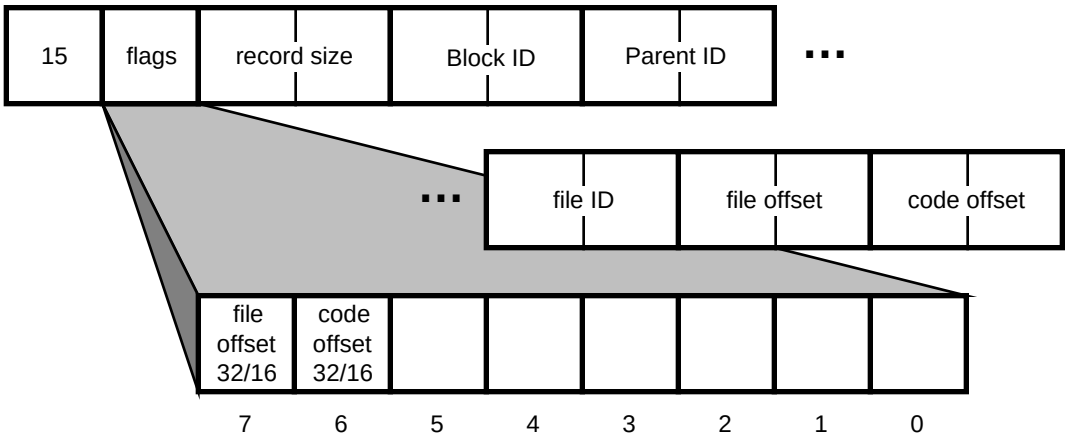


The ModuleEnd record is associated with a ModuleBegin record by the **module ID** field.

The **file offset** specifies the last byte in the source file that is to be considered part of the module.

The **is declaration** bit specifies whether the file offset reflects the end of the module's implementation or declaration.

BlockBegin Record



The BlockBegin record declares a nested scope. Usually these scopes don't have names, but a dictionary ID can be associated with the block scope *for symbolic naming purposes only*; the ID can't also be associated with another object (such as an entry point).

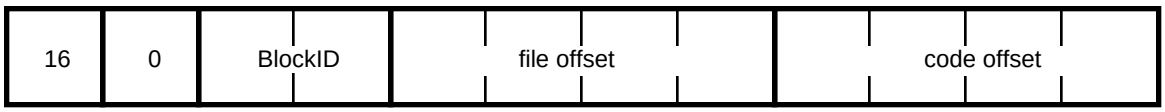
The **record size** field specifies the total number of bytes in the record.

The **parent ID** specifies the scoping entity (e.g. block or module) containing the block. It's illegal to specify a parent ID of 0, or to specify a parentage chain that doesn't (eventually) include a {ModuleBegin, Module} pair; there must be a linkable object in the chain.

The **file ID** and **file offset** specify the source file and source file offset of the first statement within the block.

The **code offset** specifies the offset of the first instruction in the block. The offset is relative to the module that the block appears in, and does not depend on any intervening scopes between the BlockBegin record and the module it appears in.

BlockEnd Record



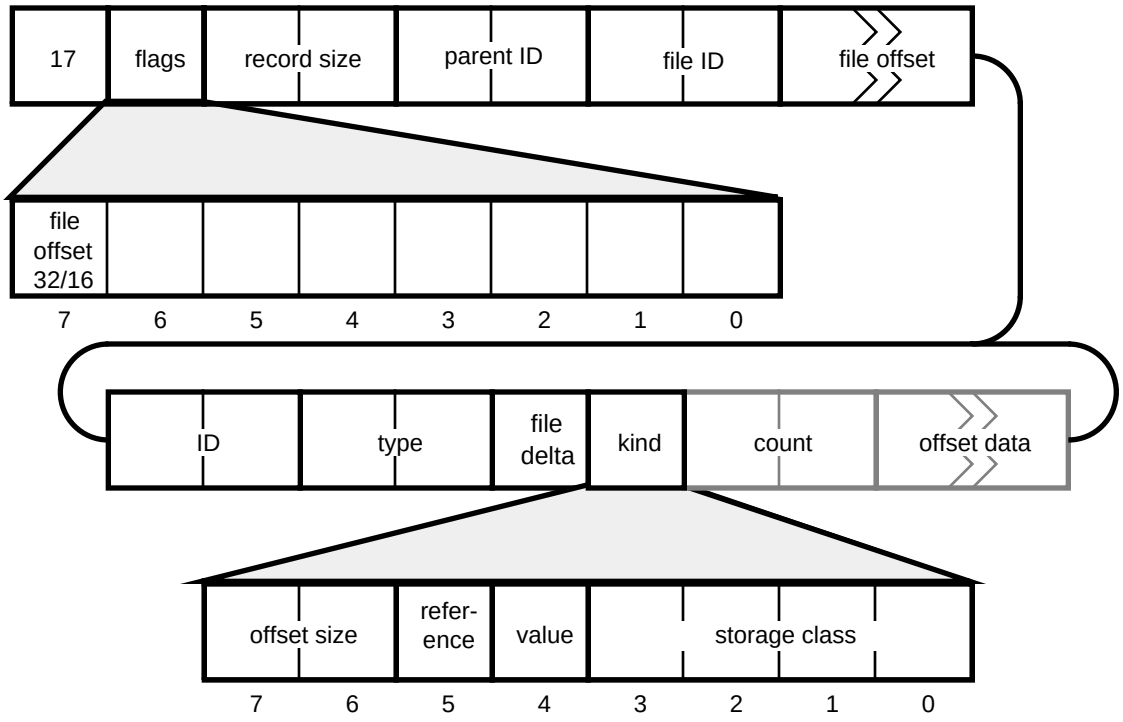
The BlockEnd record indicates the end of a block. There must be a BlockEnd record for each BlockBegin record.

The BlockEnd record is associated with a BlockBegin record by the **Block ID** field.

The **file offset** specifies the last byte in the source file that is to be considered part of the block.

The **code offset** specifies the offset within the containing module of the first instruction not to be treated as part of the block.

LocalID Record



LocalID records specify formal parameters to procedures and functions as well as local identifiers (and their types) declared within modules or blocks.

The **record size** field specifies the total number of bytes in the record.

The **parent ID** specifies the scoping entity (e.g. module or block) containing the identifier.

The **file ID** specifies the source file.

The **file offset** specifies the source file offset for the first identifier specified by this record. This field may be either a **16** or **32** bit signed value giving the 0-relative byte offset in the file specified by the **file ID**.

Following the initial file offset one or more sets of 5 fields will provide information about local identifier(s).

The **ID** associates a dictionary ID with a formal parameter or a local variable. MPW Pascal uses the convention that a LocalID with the same name as the function is the function's return value slot; this is up to the implementor, since many languages do not have a "current return value".

The **type** is a type ID specifying the identifier's type. For primitive types this ID will range from 0 to 99; for non-primitive types it will be the ID (≥100) of the LocalType record describing the type.

The **file delta** byte specifies the identifier's source offset. (The first delta byte will usually be zero). If the delta for the variable is not in the range 0..255 then a new record must be started.

The **reference** and **value** bits in the **kind** byte take the following values:

reference	value	Meaning
0	0	variable is local
0	1	variable is call-by-value parameter
1	0	variable is call-by-reference parameter
1	1	*not permitted*

The **offset size** field in the **kind** byte indicates whether there is a byte offset field present, and if there is, how large it is. If it is 3, a 16-bit **count** field following the **kind** byte specifies the number of bytes that follow the count (used for representing floating point or string constants). If the count is odd, a pad byte should be added to the data.

offset size	Meaning
0	no offset field follows
1	2-byte offset field
2	4-byte offset field
3	variable size offset field (preceded by 16-bit count)

The **storage class** field in the **kind** byte indicates the identifier's storage class as follows:

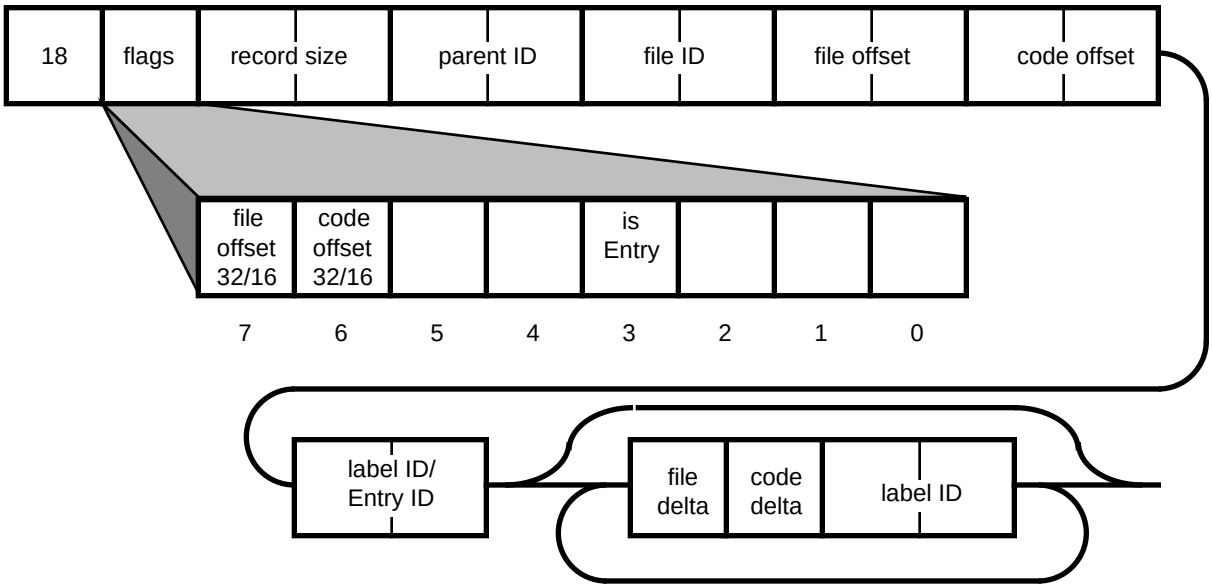
storage class	Meaning	***offset* is the…**
0	register	register number
1	A5-relative	ID of the module or entry-point corresponding to this variable
2	A6-relative	A6-relative offset
3	A7-relative	A7-relative offset
4	absolute	absolute address
5	constant	value of the constant
6…15	*reserved*	*(reserved for future use)*

A7-offsets (storage class 3) are encoded as offsets from the top of the stack (usually the return address) *prior* to executing a `LINK` instruction.

Register numbers are encoded as integers indicating the specific register as follows:

Value	Register	Meaning
0..7	D0..D7	Data registers
8..15	A0..A7	Address registers
16	CCR	Condition code register
17	SR	Status register
18	USP	User stack pointer
19	MSP	Master stack pointer
20	SFC	Source function code register
21	DFC	Destination function code register
22	CACR	Cache control register
23	VBR	Vector base register
24	CAAR	Cache address register
25	ISP	Interrupt stack pointer
26	PC	Program counter
27		*reserved*
28	FPCR	Floating-point control register
29	FPSR	Floating-point status register
30	FPIAR	Floating-point instruction address register
31		*reserved*
32..39	FP0..FP7	Floating-point data registers
40..50		*reserved*
51	PSR	PMMU status register
52	PCSR	PMMU cache status register
53	VAL	PMMU validate access level register
54	CRP	PMMU CPU root pointer register
55	SRP	PMMU supervisor root pointer register
56	DRP	PMMU DMA root pointer register
57	TC	PMMU translation control register
58	AC	PMMU access control register
59	SCC	PMMU stack change control register
60	CAL	PMMU current access level register
61..62	TT0..TT1	MC68030 transparent translation registers
63		*reserved*
64..71	BAD0..BAD7	PMMU breakpoint acknowledge data registers
72..79	BAC0..BAC7	PMMU breakpoint acknowledge control registers

LocalLabel Record



LocalLabel records give the correspondence between generated code, source statements, and label identifiers.

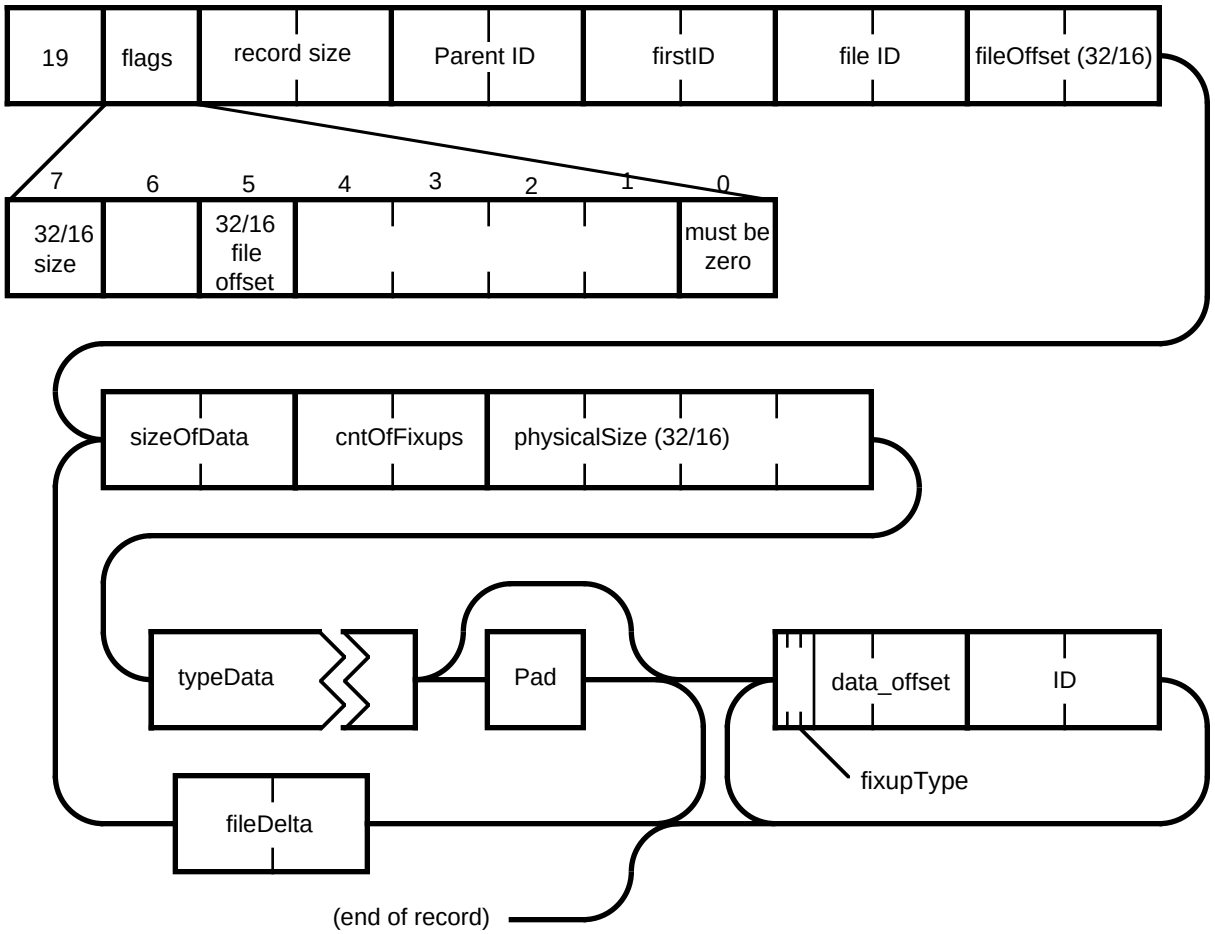
These records are similar to SourceStatement records.

The **flags**, **record size**, **parent ID, file ID**, **file offset**, and **code offset** fields are identical to those fields in the SourceStatement record (see description above).

The **isEntry** bit should be set if the label's ID is the same as that of an entry point in the module (the label's initial code offset should match the entry point's module offset).

The **file delta** and **code delta** are also encoded as they are in SourceStatement records, but here an additional **label ID** field is supplied with each file and code delta pair. The **label ID** associates a dictionary ID with a label identifier. The *first* label ID in the LocalLabel record has no file or code delta fields associated with it since that ID is at the location specified by the file and code offset fields.

LocalType Record



LocalType records associate type declarations with type IDs (or several types with several IDs). Type IDs are used to define types referred to by LocalID, LocalID2 and other LocalType records.

The **record size** field specifies the total number of bytes in the record.

The **parent ID** specifies the scoping entity (e.g. module or block) containing the identifier.

The **firstID** field specifies the first type ID declared by the record, '**firstID** + 1' is associated with the second type declared, and so on. The first type ID should be ≥ 100; IDs 0 through 99 are treated as "primitive" types and should never be defined by a LocalType record. A type ID is associated with a name by a symbolic Dictionary entry with a corresponding ID.

The **file ID** specifies the source file. If the **file ID** is 0 then the type declarations in the record are not associated with any source code, and the **fileOffset** and **fileDelta** fields are ignored (though they must appear).

The **fileOffset** field specifies the offset of the type declaration in the source file. This is a 16-bit or 32-bit unsigned value, as determined by the **32/16 file offset** bit in the **flags** field.

The **32/16 size** bit in the **flags** field specifies whether the **physicalSize** fields of all type declarations in the record are 16 or 32 bits.

Bit 0 of the **flags** field must be 0.

Type declarations immediately follow the **fileOffset** field, and extend to the end of the record. For each declaration the following fields appear:

The **sizeOfData** field specifies the size of the type data in bytes. (If this field is odd, a pad byte of zero must follow the type data).

The **cntOfFixups** field specifies the number of type ID and dictionary ID fixups that follow the type data.

The **physicalSize** field is either 16 or 32 bits (see definition of the **flags** field) and specifies the type's physical representation size, in bytes (e.g. the value returned by C's `sizeof` operator). This field may be inaccurate for packed types and sliced arrays, and is meaningless for `ProcOf` types.

The type data follows the **physicalSize** record. The format of the type data is described in Appendix A. If the size of the type data is odd then a pad byte of zero must be appended to it. The pad byte is not included in the size of the type data.

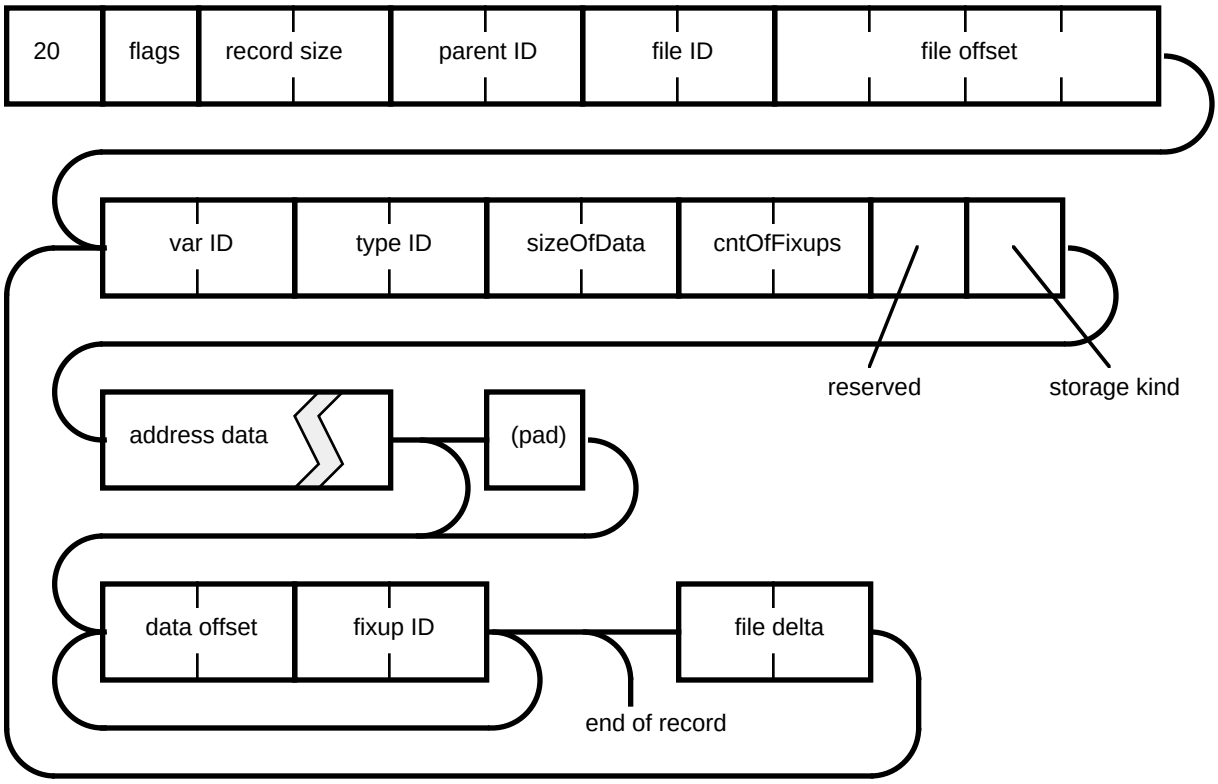
Following the type data is the fixup list, consisting of **cntOfFixups** entries. Each fixup entry is two words; an offset into the original type data followed by an ID to translate. The ID is translated and the result is inserted into the type data at the specified offset. The upper three bits of the fixup offset indicate the kind of translation to be made; the remaining bits are the type data offset. The translation kinds are:

bit 15	bit 14	bit 13	Translation Treatment
0	0	0	Insert the TTE index of the ID (which must be a LocalType or a number from 0 to 99) as a 16-bit word
0	0	1	*reserved*
0	1	0	Insert a scalar based on the ID's type: Module — insert MTE index of the module LocalID — insert CVTE index of the variable Segment — insert RTE index of the segment Source file — insert FRTE index of the source file LocalType — insert TTE index of the type
0	1	1	*reserved*
1	0	0	Insert NTE index of the ID, as a scalar
1	0	1	*reserved*
1	1	0	*reserved*
1	1	1	*reserved*

(MTEs, CVTEs, RTEs, FRTEs and TTEs and so forth are part of the SYM file produced by the linker, and are described in the document 3.4 Sym File Format.

The **fileDelta** field indicates the source-file offset of the next type declaration as a *signed* 16-bit integer. This field does not appear following the last type declaration in the record. If the source offset will not fit in 16 bits, then a new LocalType record should be started.

LocalID2 Record



The LocalID2 record is an extension of the LocalID record. It allows more "addressing modes" than the LocalID record, but it does not completely duplicate the LocalID record (for instance, it does not support definition of constants).

The **record size**, **parent ID**, **file ID** and **file offset** fields appear as they do in the LocalID record, except that the **file offset** field is always 32 bits.

The rest of the record contains one or more sub-records that each describe a local variable. For each sub-record the following fields appear:

var ID is the ID of the variable. There will normally be a Dictionary entry associated with the variable, to give it a name.

type ID is either the ID of a LocalType or a primitive type number in the range 0 to 99.

sizeOfData specifies the size of the logical address data that follows.

Exceeding 13 bytes of logical address information is expensive in terms of link time and SYM file space utilization. If possible, don't go over this limit. (13 bytes is sufficient to describe A5-relative and A6-relative variables, as well as handle and pointer-based variables).

cntOfFixups specifies the number of fixups to be applied to the logical address data (see below).

The **storage kind** field contains extra symbolic information about the variable. The low byte of this field is copied to the `SCA_KIND` field of the `STORAGE_CLASS_ADDRESS` record in the SYM file. The field can take on the following values (all other values are reserved):

```
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 parameter }
   STORAGE_KIND_WITH         = 3;   { a "WITH" scope to search      }
```

If **storage kind** is `STORAGE_KIND_REFERENCE` then the logical address refers to a *pointer* to the actual parameter variable. (The debugger will append a logical address opcode of %01101001 to indirect to the actual value).

If **storage kind** is `STORAGE_KIND_WITH`, the debugger uses the variable's type information to construct logical addresses to named fields in the type. The variable itself can be anonymous, but it should be of type RecordOf or UnionOf, or a pointer (or a pointer to a pointer, etc.) to one of those types.

address data is a vector of logical address bytecodes (described below). If **sizeOfData** is odd, a **pad** byte of zero must follow the address data.

Fixups are used to modify the address byte-codes to reflect information that is available only at link-time. **cntOfFixups** specifies the number of fixup pairs that follow the address data. The **data offset** word contains flags (in the upper four bits) and an offset into the address data (in the lower 12 bits). The **fixup ID** word contains an ID that is translated; the kind of translation to be performed depends on the flags in the offset word:

Upper Four Bits	Kind of Fixup
0000	32-bit segment-offset fixup. The ID must refer to a code or data module or an entry point associated with one. Add the module or entry point's 32-bit segment offset to the specified longword in the address information. If the ID refers to a dead module (one that was stripped by dead-code analysis) then the longword in the address information is not modified.
0001	Resource fixup. The ID must refer to a code or data module or an entry point. Six bytes in the address information are affected; the first four bytes are replaced with the resource type of the segment that the ID appears in (e.g. 'CODE'); the last two bytes are replaced with the segment's resource ID.
0010 … 1111	*reserved for future use*

The record may end following the last fixup pair.

If there is another sub-record, the **file delta** field contains the next variable's source location delta.

Logical Address Bytecodes

A logical address is a sequence of postfix bytecodes terminated with an EndOfPostfix code. During evaluation there is a stack available for address calculations. The result is on the top of the stack when EndOfPostfix is reached. The stack is typed (see below).

The bytecodes are interpreted by examining their high four bits. The low four bits form common modifiers.

bits 7654	bits 3210	Arguments	Opcode Description
0000	0000		EndOfPostFix. Execution terminates.
	0001 *through* 1111		*Reserved*
0001			Load. The low four bits have the following interpretations:
	0000	<b>	Register. The byte following specifies a register. A *reference* to the register is loaded, not the actual *contents* of the register; an indirect is necessary to get a register's value. Canonical register numbering is used (see the LocalID record).
	0001	<b>	Byte. The following byte is sign-extended and pushed onto the stack.
	0010	<w>	Word. The following word is sign-extended and pushed.
	0011	<l>	Long. The following longword is pushed.
	0100	<l> <w>	The following four bytes are a resource type, the next two bytes are the resource number. The address of the resource is pushed.
	01xx	<b,w,l>	Trap. The byte, word or long is interpreted as a trap number.
	1xxx		Immediate. The 3-bit value is pushed without sign extension.
0010			Add. The meaning of the low four bits is as above, except for:
	0000		TOS. The top two stack entries are popped, added, and the result pushed.
	0001	<b>	Byte.
	0010	<w>	Word.
	0011	<l>	Long.
	01xx		*reserved*
	1xxx		Immediate.

0011		Subtract. Essentially identical to Add:
	0000	TOS.
	0001 <b>	Byte.
	0010 <w>	Word.
	0011 <l>	Long.
	01xx	*reserved*
	1xxx	Immediate.
0100		Multiply. Essentially identical to Add:
	0000	TOS.
	0001 <b>	Byte.
	0010 <w>	Word.
	0011 <l>	Long.
	01xx	*reserved*
	1xxx	Immediate.
0101	xxxx	Push the contents of the register named by the low four bits of this opcode. The numbering is traditional: 0..7--> D0..D7 8..15--> A0..A7
0110		General indirect / contents of. This opcode fetches an operand from memory, possibly replacing the old TOS.
	0xxx	The low three bits indicate a register D0..D7. The low 16 bits of the register are sign-extended and pushed.
	1000	(TOS).W The two bytes pointed to by the TOS are fetched and sign extended, replacing the TOS.
	1xxx	(TOS).L Repeated fetch. The longword pointed to by the TOS is fetched, replacing the TOS. This operation is repeated 'xxx' times.
0111 *through* 1111		*Reserved*

The evaluation stack is typed. For instance, the Load Register opcode pushes a reference to the specified register. To obtain the contents of the VBR, for example, the following opcodes would be used:

```
0001 0000  00010111     ; Load Register VBR                2 bytes
0110 1000               ; Indirect ==> contents of VBR     1 byte
```

Thus it's possible to do arithmetic with addresses, resources and traps (e.g. push a resource and add an offset to it).

Examples

A typical A5-relative or A6-relative variable could be addressed with the following opcodes:

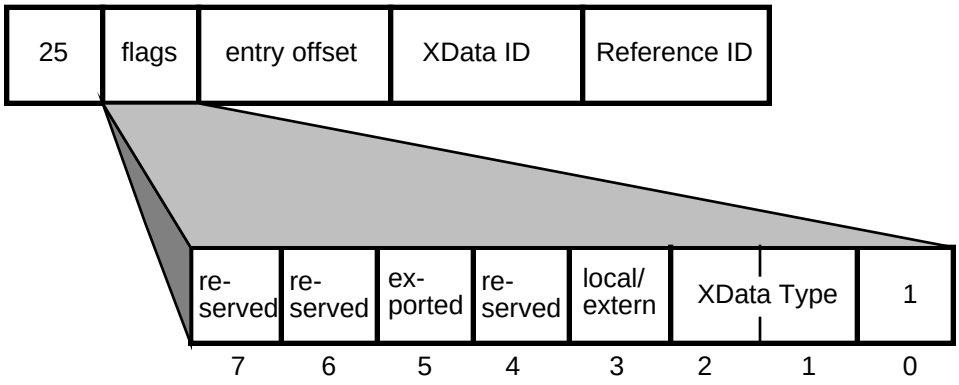
```
0001 0011  <lw>     ; Push 32-bit A5 offset                5 bytes
0101 1101           ; Push A5                              1 byte
0010 0000           ; Add TOS                              1 byte
0000 0000           ; EndOfPostfix                         1 byte
                    ;                                      8 bytes total
```

A variable contained in a relocatable block could be addressed with:

```
0101 1101           ; Push A5                              1 byte
0010 0011  <lw>     ; Add 32-bit A5-offset of ptr to handle 5 bytes
0110 1010           ; Indirect twice ==> address of block  1 byte
0010 0011  <lw>     ; Add offset of variable within block  5 bytes
0000 0000           ; EndOfPostFix                         1 byte
                    ;                                      13 bytes total
```

Similarly, a pointer-based variable could be addressed by doing one less indirection (%01101001 instead of %01101010).

XData Record



The XData record defines a data module specific to the CFM-68K runtime architecture. XData records are independent of the current code and data module streams. There are no Contents, Size or Reference records associated with an XData record. All of the necessary information is contained within the XData record itself.

The **XData ID** field contains the ID associated with the data module. If the data module is named, the Dictionary record for the ID must appear before the XData record.

Bit 0 of the **flags** byte should be ignored. XData records do **not** affect the current code or data module.

Bits 1 and 2 of the **flags** byte define the type of CFM-68K data module that this record represents. These 2 bits can have the following values:

Value	Type of CFM-68K Data Module
0	XVector
1	XPointer
2	XDataPointer
3	reserved

XData Type = 0 (XVector)

The **Reference ID** field contains the ID of the code module to which the XVector refers.

The **entry offset** field specifies an offset into the referenced code module. The XVector will point to the start of the referenced code module plus the specified offset.

XVectors may be either **local** or **external**. The local/extern bit should be set to the same value as the local/extern bit of the referenced code module.

XVectors may be flagged as **exported**. If an XVector is exported, the linker will not strip it even if it is not referenced by any other module and is not the main module. Only external XVectors can be exported.

XData Type = 1 (XPointer)

The **Reference ID** field contains the ID of the XVector to which the XPointer refers.

The **entry offset** field is not used.

XPointers may be either **local** or **external**. The local/extern bit should be set to the same value as the local/extern bit of the referenced XVector.

The **export** bit must be 0.

XData Type = 2 (XDataPointer)

The **Reference ID** field contains the ID of the data module to which the XDataPointer refers.

The **entry offset** field is not used.

XDataPointers may be either **local** or **external**. The local/extern bit should be set to the same value as the local/extern bit of the referenced data module.

The **export** bit must be 0.

XData Record Usage

The section covers what language translators should generate when defining a code module, referencing a cross-fragment code module, defining a data module, or referencing a cross-fragment data module.

Defining a Code Module

When defining a code module, the following should occur. In some circumstances one or more of these items may be removed by optimization.

1. Dictionary entries should be generated for:
    a. the code module
    b. the code module's XVector
    c. the code module's XPointer
    d. the code module's internal entry point
2. The Module record should be generated.
3. XData records should be generated for:
    a. the XVector (XData Type = 0)
    b. the XPointer (XData Type = 1)
4. An EntryPoint record for the internal entry point should be generated.
5. The module references should be generated.
6. The module contents should be generated.

With the CFM-68K runtime architecture, many code modules have an internal entry point that is not at the immediate start of the module, but rather 2 bytes into the module. This entry point is used by internal callers (i.e. callers from within the same code fragment), and skips an instruction which performs A5 register switching. External or indirect calls to the code module go through the XVector and are directed to the start of the module since, in these cases, the A5 switching is needed. As an optimization, the A5 switching instruction may not be present. Therefore, internal calls cannot simply branch 2 bytes into the module. To solve this problem, the internal entry point for a code module is designated with an EntryPoint record. In addition to generating an EntryPoint record when defining a code module, the internal entry point must be used for all direct calls to the code module. That is, when calling a code module using an in-fragment calling sequence (i.e. a simple `JSR` or `BSR` instruction) the target should be the module's internal entry point not the module itself.

Referencing a Cross-Fragment Code Module

When referencing a cross-fragment code module, all that is required is a Dictionary entry for the XPointer name. The ID for that entry is then used by the reference.

Defining a Data Module

The data module definition is unchanged, with the exception of the new flags bit to designate export status.

Referencing a Cross-Fragment Data Module

When referencing a cross-fragment data module, a Dictionary entry for the XDataPointer is needed, as well as an actual XDataPointer (an XData record with an XData Type of 2). Duplicate XDataPointers found in separate compilation units are silently ignored by the linker.

CFM-68K Naming Conventions

Consider the following definitions in the examples below.

```
void MyProc (void)
int MyData;
```

By convention, XVectors are named the same as the function they refer to except prefixed with "_%". For example, the XVector for `MyProc` would be named `_%MyProc`.

XPointers are named the same as the XVector they refer to except prefixed with "_@". For example, the XPointer for `_%MyProc` would be named `_@MyProc`.

XDataPointers are named the same as the data module they refer to except prefixed with "_#". For example, the XDataPointer for `MyData` would be named `_#MyData`.

A code module's internal entry point has the same name as the module, except prefixed with "_$". For example, the internal entry point for `MyProc` would be named `_$MyProc`.

Appendix A- Type Interpretation

The goals of Type Interpretation are to support the interpretation of SADE debugger variables by type, map from name to type, and map from type information to name. Operators, such as array indexing, indirection, field name, are applied to types to yield another type and address information. The type information should be complete. By complete is meant that the debugger should have a minimum amount of knowledge about how data for any particular type is stored. At most, the debugger should apply word alignment criteria to types within structured types. Type information should be easy for compilers to emit and for the linker to extract from the Object Module Format. The storage of type information should be compact, yet fast and easy to expand. High level language interpreters for type information should not be prohibitively expensive.

Overview

SADE type information is contained in word aligned variable length data structures called **Type Table Entries**, abbreviated to **TTE**. These Type Table Entries are contiguously numbered (starting at 100, as the indices 0 .. 99 are reserved) and accessed via an indexing table of four byte disk addresses. An index into this table is called a **TTE Index**. The Type Table Entries are aligned on word (two byte) boundaries. No global/local scope information is contained in the type table; scoping is via the Modules table and its Contained Modules, Types, etc. entries. A picture of a TTE is given later in this document.

The **TypeCodes** portion of a TTE contains size information and an interpretative representation of a type. The exact form for the TypeCodes is described below. The paradigm used for the SADE type mechanism is types as functions. All types are either basic types or functions with types and integral constants as arguments. The realization of this paradigm is prefix code: an operator followed by operands.

Type values are either scalar types or composite types. Instances of a scalar type can be ordered, while composite types cannot necessarily be ordered. Scalar types are further divided into integral and non-integral scalar types. Integral types can be mapped to the set of integers.

Type Functions

Following are the definitions for the type functions and their arguments. Except for two cases, type function arguments are either other type functions, encoded scalars, or instances of a scalar type (see ScalarOf). In one exception, *ConstantOf()*, one argument is a sequence of uninterpreted bytes. In the

other, *TTE()*, the argument is an unaligned two bytes interpreted as an unsigned word. Except in these two cases, the convention used is to prefix arguments with S if it is a scalar or T if it is a Type or an instance of a Type.

BasicType(SType)

This function returns a ground type, one which cannot be composed of other types† . The argument is an integer in the range 0 .. 99. By convention, the empty type, called *void*, is represented by BasicType(0).

TTE(SType)

The type, and also the name for the type, is found at the SType'th entry in the Type Table. This aliasing function allows the association of a name to a type, or the factoring out of a shared anonymous type into one common entry, saving table space.

PointerTo(Ttype)

The argument is a type. The value of the function is pointer to that type. PointerTo(void) is a generic pointer, equivalent to the C type (void *).

ScalarOf(Ttype, Svalue)

The type returned by the ScalarOf function names an instance of given scalar type. The ScalarOf function is usually referred to by RecordOf or EnumerationOf

NamedTypeOf(Snte, Ttype)

The scalar is an index for a Name Table Entry. That entry gives the name for the type Ttype. This mechanism is used to give names which are local to a type, such as record and union field names.

ConstantOf(Ttype, Slength, byte...)

Similar to ScalarOf, except that the constant can be of a non-integral type, such as floating point constants or composite type constants. The Ttype is the type of the constant, the Slength is the number of bytes in the constant and the unencoded bytes following are the bytes comprising that type.

† This is not quite true. Due to historical reasons, strings are considered a ground type instead of a derived type.

EnumerationOf(Tbase, Slower, Supper, Snelements, Ttype...)

Names an enumeration type. Tbase names the underlying scalar type the elements of the enumeration are drawn from and determines the storage size of the Enumeration. Usually, they are drawn from BasicType(SignedWord). The Slower and Supper are the lower and upper bounds of the enumeration. Snelements is the number of Ttype elements named as part of the enumeration. Snelements can be less than Supper -Slower +1 if the enumeration is sparse, as is possible with C enums.

VectorOf(Tindex, Telement)

The function takes two arguments. The first is the index type, the scalar type the vector indices are drawn from. The second argument is the type of the vector elements. The value of the function is ARRAY [Tindex] OF Telement. C arrays are encoded by having Tindex be a ScalarOf(), where the value of the scalar is the number of elements in the array. If the scalar value is 0 then the size of the array is indeterminate.

RecordOf(Snfields, Soffset & Ttype...)

Returns a type composed of a linear sequence of types. Snfields is the number of types in the composite type. The argument types are pairs of a scalar and a type. The Soffset scalar gives the offset to that element from the beginning of the type. The Ttype is the type of that element. The offsets are byte offsets. The representation details are discussed in the following section.

UnionOf(Ttag, Soffset, Snfields, Tvariant & Ttype...)

The Union function could be built from the RecordOf function. Ttag is the scalar type of the tag . For C and other languages whose unions do not have a tag, Ttag is the ground type void. Soffset is the offset to the first variant from the start of the union. If there is no tag variable, then Soffset will be 0. Otherwise, Soffset is the size of the tag variable plus any required alignment. The Snfields is the number of variants in the union. The Tvariant and Ttype pairs define one element of the union. The Tvariant is an instance of the Ttag and names the variant. If Ttag is void, Tvariant is void. Ttype is the field of the union.

SubRangeOf(Tbase, Tlower, Tupper)

Names a subrange of the scalar type, Tbase. The bounds of the subrange are given by the Tlower and Tupper values.

SetOf(Tbase)

Names a set type. The set is composed of elements drawn from scalar type Tbase.

ProcOf(SClass, TReturn, SArgc, TArg...)

Names a procedure type. SClass is the *class* of procedure. The class of the procedure defines how arguments are passed to it and values are returned from it. TReturn is the type of the return value, if *void* then the ProcOf names a procedure instead of a function. SArgc is the number of arguments to the procedure. The TArg are the arguments to the procedure. The argument order is the order of declaration, as it appears in the source text. How the TArgs are actually passed, on the stack, Pascal or C, in registers, is as per the SClass.

ValueOf(Ttype, Scvte, Smte)

The scalar value, of type Ttype, can be had by fetching the variable whose CVTE index is Scvte and whose containing module's MTE index is given by Smte. Smte is required because the variable named by the Scvte might be in a register or relative to A6, requiring a debugger to find the proper stack frame. Ttype will be the same as the type in the CVTE; it is duplicated because we do not want to reference the CVTE just to find the type.

ArrayOf(Telement, Sorder, Sndim, Tbound1, …)

The ArrayOf type descriptor is used to describe monolithic arrays, those which cannot or should not be described with VectorOf functions. The Telement is the type of each array element. Sorder describes how the address of an element is computed from the array indices. Sndim is the number of dimensions of the array, and Tbound1… are the indexing types, usually SubRangeOf types.

Sorder can have one of two values. If 0, then the address of array(i,j,k) is computed by:

(i-*lowerBound*(Tbound1)) +
(j-*lowerBound*(Tbound2)) * *span*(Tbound1) +
(k-*lowerBound*(Tbound3)) * *span*(Tbound2) * *span*(Tbound1)

If 1, then the address is computed by:

(i-*lowerBound*(Tbound1)) * *span*(Tbound2) * *span*(Tbound3) +
(j-*lowerBound*(Tbound2)) * *span*(Tbound1) +
(k-*lowerBound*(Tbound3))

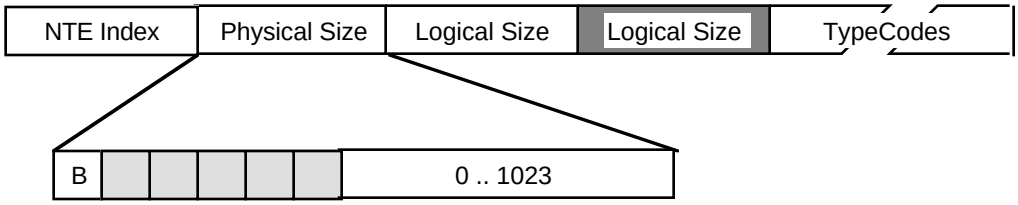
where:

lowerBound(Type)	= lower bound of the subrange or enumeration
upperBound(Type)	= upper bound of the subrange or enumeration
span(Type)	= *upperBound*(Type) - *lowerBound*(Type)

Note the difference in the order of the element type and index type from that in the VectorOf function. This was done to keep the variable number of bounds as the last arguments to the ArrayOf function.

The Type Table Entry

The SADE symbol table, created by the linker from information in the object files, stores type information in a Type Table Entry. The Type Table Entry is a word aligned variable length data structure of the form:



NTE Index

This is a four byte field, giving the name of the type via an index into the Name Table. The index can be 0, meaning that this is an anonymous type.

Physical Size

This two byte field defines the number of bytes taken up by the TypeCodes field. It does not include the Physical Size or Logical Size fields. The maximum size of a TypeCodes field is 1023 bytes. The other bits are flag bits. Of these flag bits, only one, the **B** bit, is defined. The other bits are reserved for future use and should be set to 0. The B bit applies to the Logical Size and means *Big*. When set, the Logical Size field will be four bytes instead of two, allowing the description of very large data structures such as Fortran arrays.

Logical Size

The Logical Size field is either two bytes or four bytes wide. If the B bit in the Physical Size field is 0, then the Logical Size is two bytes long. If set, then the field is four bytes long. In either case, the value of the field is the number of bytes required to store the type.

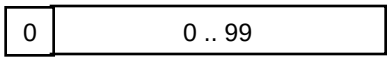
TypeCodes

Following the header information are the TypeCodes themselves. This is a sequence of *Physical Size* bytes of type information. The TypeCodes are described in the next section.

Representation of TypeCodes

Type function codes are single bytes. The Basic Type function has the MSB of the byte 0 and the basic type number encoded in the lower 7 bits. The Type Composition Function has a 1 in the MSB and the function code in the lower 6 bits. The next-to-MSB is a flag indicating the type of offset encountered in the arguments to that type. If 0, then offsets are byte offsets. If 1, then offsets are bit offsets from the beginning of the type.

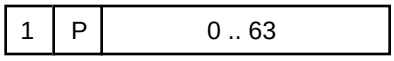
Basic Type



The byte names a Basic Type in the range 0 .. 99. The standard values for the Basic Types are as follows:

Value	Basic Type
0	No type
1	Pascal string
2	unsigned long word
3	signed long word
4	extended (10 bytes)
5	Pascal boolean (1 byte)
6	unsigned byte
7	signed byte
8	character (1 byte)
9	character (2 bytes)
10	unsigned word
11	signed word
12	singled
13	double
14	extended (12 bytes)
15	computational (8 bytes)
16	C string
17	as-is string

Type Composition Function



The TypeCode is in the lower 6 bits. The P bit means the type is PACKED. The default, P=0, means that the type is unpacked. The values for the TypeCode are:

Value	TypeCode
1	TTE
2	PointerTo
3	ScalarOf
4	ConstantOf
5	EnumerationOf
6	VectorOf
7	RecordOf
8	UnionOf
9	SubRangeOf
10	SetOf
11	NamedTypeOf
12	ProcOf
13	ValueOf
14	ArrayOf

The SClass for ProcOf types has the following meanings:

Value	Meaning
1	Undefined
2	Pascal
3	C with fixed arguments
4	C with variable number of arguments

Undefined

The calling conventions of the function are undefined.

Pascal

The calling convention follows Pascal. The return value or a pointer to the return value is placed on the stack and is then followed by a fixed number of arguments. The called procedure/function cleans up the stack before returning to the caller.

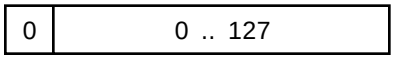
C with fixed arguments

C calling convention. The caller removes any arguments that were passed. The arg count is the number that was specified in the prototype for the C function or in the function definition.

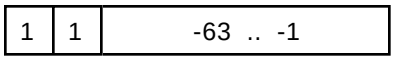
C with variable number of arguments

Similar to the above, except that the argument count is the minimum number of arguments to pass to the function. It is suggested that compilers emit a ProcOf(3, *TReturn*, 0) for functions referenced without prototypes or function definitions, as that is the most general case. This is especially useful for declarations of pointers to function.

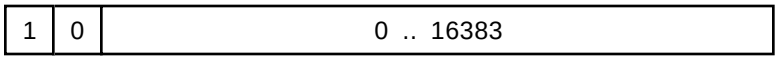
Representation of Scalars



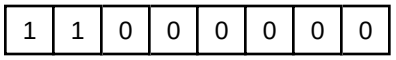
A constant in the range 0 .. 127 fits into a single byte. The high bit is 0, marking this as a small integer.



If the two high bits of a byte are 11, then the byte represents a small negative integer in the range -63 .. -1. The value -64, with the lower six bits 0, is used as switch to long. See below.



If the two high bits of a byte are 10, then a larger integer in the range 0 .. 16383 is specified by appending the next byte to the 10 byte and stripping off the high two bits of the word. The constant is not aligned to word boundaries.



4 bytes of longint

A byte whose value is -64 is used for expansion past the preceding bounds. The four bytes following (not aligned to word boundaries) are the longint.

Type Representation Examples

In these examples, it is assumed that “Integer” is the name of a BasicType of SignedWord.

Pascal source

```
TYPE
      VHSelect    = (v, h);

      Point       =
        RECORD
            CASE Integer OF
              0: (v: Integer;
                  h: Integer);
              1: (vh: ARRAY [VHSelect] OF Integer);
        END;
```

```
Rect        =
  RECORD
      CASE Quux: Integer OF
        0:(top:         Integer;
           left:        Integer;
           bottom:      Integer;
           right:       Integer);
        1:(topLeft:     Point;
           botRight:    Point);
  END;
```

A possible compilation into TTE

The representation of type information in the Object Module must be less compact than in the Type Table so that the linker will not have to interpret the type information, just resolve linkages, fold anonymous types into types referring to them, and emit the compacted information. The differences are:

- TTE(x) are replaced by the dictionary IDs of the entries.
- BasicType(x) are replaced by dictionary IDs less than 100.

In the following, dictionary numbers are distinguished from integers. BasicType(n) should be viewed as having been replaced by the canonical dictionary for that type. The numbers in **BOLD** are shorthand for TTE(**nnn**). Do not emit them as plain old numbers into the object file, emit them as 0x81, **yyy**, where 0x81 is the TTE type composition function and **yyy** is the scalar representation of **nnn** in the typecodes (see above, “Representation of Scalars”).

Dict #	Name	Type Definition
3000	v	ScalarOf(**3020**, 0)
3010	h	ScalarOf(**3020**, 1)
3020	VHSelect	EnumerationOf(BasicType(0), 0, 1, 2, **3000**, **3010**)
3050	vh	VectorOf(**3020**, BasicType(**INTEGER**))
3052		ScalarOf(BasicType(**INTEGER**), 0)
3054		RecordOf(/* Record has two fields */ 2, /* First, at offset 0, is the "v: Integer"*/ /* Two bytes after it is "h: Integer"*/ 0, NamedTypeOf("v", BasicType(**INTEGER**)), 2, NamedTypeOf("h", BasicType(**INTEGER**))),
3056		ScalarOf(BasicType(**INTEGER**), 1)

3060	Point	UnionOf(/* Two variants, Integer selector, no tag */ 2, BasicType(**INTEGER**), **3052**, **3054,**/* 0: selects the record of "v" and "h" */ **3056**, **3050**/* 1: selects the array of two ints */)
3132		RecordOf(4, 0, NamedTypeOf("top", BasicType(**INTEGER**)), 2, NamedTypeOf("left", BasicType(**INTEGER**)), 4, NamedTypeOf("bottom", BasicType(**INTEGER**)), 6, NamedTypeOf("right", BasicType(**INTEGER**)))
3134		RecordOf(2, 0, NamedTypeOf("topLeft", **3060**), 4, NamedTypeOf("botRight", **3060**))
3140	Rect	UnionOf(/* Two variants, Integer selector, tag is "Quux", size 2 */ 2, NamedTypeOf("Quux", BasicType(**INTEGER**)), 2 **3052**, **3132**,/* 0: First variant is the record of 4 Integers */ **3056**, **3134**/* 1: Second variant is the record of 2 Points */)

Another possible compilation into TTE

The linker can compact the representation, folding anonymous entries into the type definitions referring to them. This would yield the following:

Dict #	Name	Type Definition
3100	v	ScalarOf(**3102**, 0)
3101	h	ScalarOf(**3102**, 1)
3102	VHSelect	EnumerationOf(BasicType(**SIGNEDWORD**), 0, 1, 2, **3100**, **3101**)
3105	vh	VectorOf(**3102**, BasicType(**INTEGER**))
3106	Point	UnionOf(/* Two variants, Integer selector, no tag */ 2, BasicType(**INTEGER**), /* 0: selects the record of "v" and "h" */ ScalarOf(BasicType(**INTEGER**), 0), RecordOf(/* Record has two fields */ 2, /* First, at offset 0, is the "v: Integer" */ 0, NamedTypeOf("v", BasicType(**INTEGER**)), /* Two bytes past the first, is "h: Integer" */ 2, NamedTypeOf("h", BasicType(**INTEGER**))), /* 1: selects the array of two ints */ ScalarOf(BasicType(**INTEGER**), 1),

		3105)
3114	Rect	UnionOf(/* Two variants, Integer selector, tag is "Quux", size 2*/ 2, NamedTypeOf("Quux", BasicType(**INTEGER**)), 2 /* 0: First variant is the record of 4 Integers */ ScalarOf(BasicType(**INTEGER**), 0), RecordOf(4, 0, NamedTypeOf("top", BasicType(**INTEGER**)), 2, NamedTypeOf("left", BasicType(**INTEGER**)), 4, NamedTypeOf("bottom", BasicType(**INTEGER**)), 6, NamedTypeOf("right", BasicType(**INTEGER**))), /* 1: Second variant is the record of 2 Points */ ScalarOf(BasicType(**INTEGER**), 1), RecordOf(2, 0, NamedTypeOf("topLeft", **3106**), 4, NamedTypeOf("botRight", **3106**)))

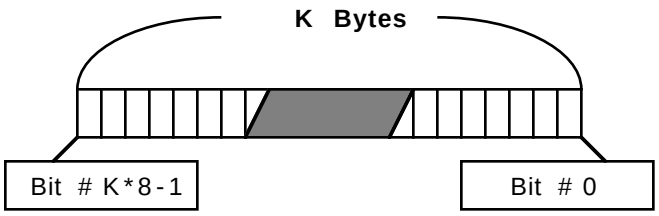
Type Interpretation and Packed Data

The previous section did not describe **PACKED** data, such as Pascal's packed arrays (actually vectors) and records and C's bit field structures . This proposal for describing packed data rests on the observation that it is not the composite types that are packed but rather components of the composite types. The solution is general enough to solve today's problem and be extensible to future language implementations.

When a type has the PACKED attribute, it is followed by packing information. The packing information describes the packing for that occurrence of the type. In the case of the *VectorOf* composing type, the packing information describes how a given number of elements are arranged within a fixed number of bytes.

Storage Framework

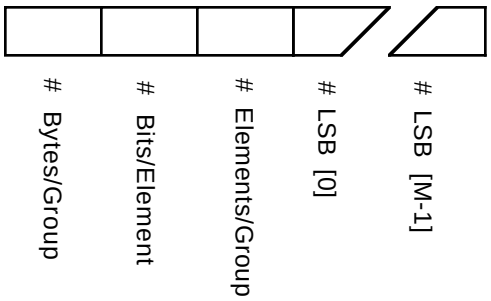
In general, an instance of a datatype is a sequence of bits. These bit sequences are usually aligned on address boundaries and are totally contained within an integral unit of addressable units of storage (bytes). Packing an instance of a datatype minimizes the amount of wasted bits, possibly at the expense of access time. Therefore, the packed type information assumes the following bit-level view of **K** bytes of storage, with the most significant bit (and byte) at the left and the least significant bit (byte) at the right:



A packed data field is represented by two scalars: the **MSB** and **LSB**. The scalars are represented as described in the TypeCodes section. This allows bitfields and packed data to be from 1 to 2**32-1 bits wide†. The number of bytes occupied by that packed data field is (MSB+7) DIV 8. LSB can be thought of as the shift factor and MSB-LSB+1 the number of bits in the mask.

A packed non-*VectorOf* type is followed by a single occurrence of MSB and LSB.

Packed vectors have a regular structure. A group of vector elements, M, will be packed into a number of bytes, N. The packing is regular, meaning that the bit field for Vector[p*M+k] is the same as that for Vector[k] except that it is p*N bytes after Vector[k]. Because of this regularity, the bit field information is in a different format:



The first scalar gives N, the number of bytes that the M vector elements fit into. The next scalar gives the width of the bit mask. The third scalar gives M, the number of elements in the repeating group. Following these three scalars are M scalars. These M scalars give the LSB for the 0'th to the M-1'th vector element. LSB[i]+# Bits/Elements-1 gives MSB[i].

---

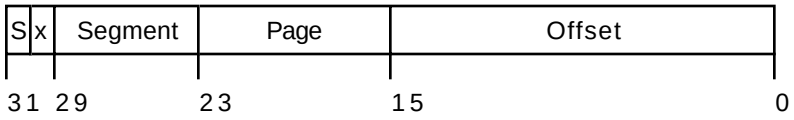
† Originally, it was thought that the packed data field could be represented by a single byte of mask and shift information. One nibble in the byte represented the field width (0 to 15 bits) in the word. The position of the field was represented by the other nibble as the number of bits to left shift the mask. However, one common case, the C bitfield with more than 15 bits, prevents a single byte representation for packed data. Since more than one byte is required to represent packed data, the MSB/LSB view was adopted.

MSB/LSB was chosen over MASK and SHIFT because one less addition is required to yield the number of bytes required for the data type.

Packed Data Example

The following example is in C and from Harbison and Steele. It is assumed to follow 68000 addressing in that the most significant bytes are at the lowest addresses. Also note that the example assumes that this particular C compiler packs fields into ints starting with the least significant bit of the int.

Example bitfield



C source

```
typedef struct {
              unsigned offset       : 16;
              unsigned page         : 8;
              unsigned segment      : 6;
              unsigned              : 1;  /* For future use */
              unsigned supervisor   : 1;
              } V_ADDR;
```

Possible Compilation into TTE

The fields above are not all packed fields. Only the segment and supervisor fields need to have the **PACKED** attribute applied to them. Those two fields are followed by the bit positions within the byte.

Dict #	Name	Type Definition
3107	V_ADDR	RecordOf(4, /* First is "offset". Note its byte offset */ 2, NamedTypeOf("offset", **UNSIGNEDINTEGER**), /* Next is "page". It is physically before "offset" */ 1, NamedTypeOf("page", **UNSIGNEDINTEGER**), /* Finally, "Segment" and "Supervisor" in byte 0*/ 0, **packed** NamedTypeOf("segment", **UNSIGNEDINTEGER**), 5, 0, 0, **packed** NamedTypeOf("supervisor", **UNSIGNEDINTEGER**), 7, 7)