# Tool Interface Standard (TIS)
# Portable Formats Specification

## Version 1.1

*TIS Committee*
**October 1993**

The TIS Committee makes no warranty for the use of these standards.

THE TIS COMMITTEE SPECIFICALLY DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, AND ALL LIABILITY, INCLUDING CONSEQUENTIAL AND OTHER INDIRECT DAMAGES, FOR THE USE OF THE SPECIFICATIONS AND THE INFORMATION CONTAINED IN IT, INCLUDING LIABILITY FOR INFRINGEMENT OF ANY PROPRIETARY RIGHTS. THE TIS COMMITTEE DOES NOT ASSUME ANY RESPONSIBILITY FOR ANY ERRORS THAT MAY APPEAR IN THE SPECIFICATIONS, NOR ANY RESPONSIBILITY TO UPDATE THE INFORMATION CONTAINED IN THEM.

The TIS Committee retains the right to make changes to these specifications at any time without notice.

Additional copies of this manual can be obtained from:

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

Or call: 1-800-548-4725

Refer to Intel Order Number 241597.

# Introduction

This Tool Interface Standards Portable Formats Specification, Version 1.1 is the result of the work of the TIS Committee--an association of members of the microcomputer industry formed to work toward standardization of the software interfaces visible to development tools for 32-bit Intel Architecture operating environments. Such interfaces include object module formats, executable file formats, and debug record information and formats.

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

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

TIS Portable Formats Specification, Version 1.1 and TIS Formats Specification for Windows™, Version 1.0 are the latest deliverables of the TIS Committee. They are based on existing, proven formats in keeping with the TIS Committee's goal to adopt, and when necessary, extend existing standards rather than invent new ones.

Within this Portable Formats specification are definitions for loadable, linkable, and debug formats, as listed in the following table. These tool interface standards will enhance the delivery of applications software for the 32-bit Windows, OS/2™ and UNIX® operating system environments, as well as for future systems.

| Tool Interface Type | Tool Interface Format | Industry Source |
|---|---|---|
| Loadable and Linkable | ELF (Executable and Linkable Format) | UNIX Systems Laboratories and UNIX International ABICC |
| Debug | DWARF (Debug With Arbitrary Record Format) | UNIX International |
| Linkable | OMF (Relocatable Object Module Format) | IBM Corporation, Intel Corporation, Microsoft Corporation, and PharLap Software Incorporated |

These, in conjunction with the Windows 32-bit formats, represent the tool interfaces currently agreed upon by TIS Committee members as TIS standards. In the future, the Committee expects to work on standardization efforts for tool interfaces in other areas that will benefit the microcomputer software industry, such as dump file formats, object mapping, and 64-bit operating environments.

# Table of Contents

# I

# Executable and Linkable Format (ELF)

# TIS Portable Formats Specification, Version 1.1
# ELF: Executable and Linkable Format

This document describes the Tool Interface Standards (TIS) portable object file format standard, ELF (Executable and Linkable Format). The original format specification was made available by USL (UNIX System Laboratories) as part of the ABI (Application Binary Interface). A committee in UNIX International, called the ABICC (ABI Coordinating Committee), approved this format specification for UNIX System V.

The TIS Committee formed an object format subcommittee to evaluate the widely available formats and to select one as the TIS standard. After studying many different formats, the committee concluded that ELF can be easily adopted across numerous 32-bit Intel Architecture environments and is therefore the best standard for a portable linkable and loadable format.

Some of the major reasons for selecting this format are the public nature of the specification and the fact that the PLSIG and ABICC standardization committees can enhance its formats. The PLSIG (Programming Language Special Interest Group) committee is able to propose modifications to the format for approval by the ABICC. Once the ABICC approves a modification, the specifications are enhanced accordingly. The TIS Committee intends to work through PLSIG to address issues of interest, such as compression, resource management, and future 64-bit architectures.

The TIS Committee created this ELF document by extracting the executable and linkable format information from the *System V Application Binary Interface* and the *System V Application Binary Interface Intel 386™ Processor Supplement* published by Prentice Hall (UNIX Press). This document is unchanged and is the same document that was released in TIS Portable Formats Specification, Version 1.0.

# I

# Executable and Linkable Format (ELF)

# Contents

## Preface

## 1 OBJECT FILES

## 2 PROGRAM LOADING AND DYNAMIC LINKING

## 3 C LIBRARY

## I Index

# Figures and Tables

# Preface

## ELF: Executable and Linking Format

The Executable and Linking Format was originally developed and published by UNIX System Laboratories (USL) as part of the Application Binary Interface (ABI). The Tool Interface Standards committee (TIS) has selected the evolving ELF standard as a portable object file format that works on 32-bit Intel Architecture environments for a variety of operating systems.

The ELF standard is intended to streamline software development by providing developers with a set of binary interface definitions that extend across multiple operating environments. This should reduce the number of different interface implementations, thereby reducing the need for recoding and recompiling code.

## About This Document

This document is intended for developers who are creating object or executable files on various 32-bit environment operating systems. It is divided into the following three parts:

- Part 1, ‘‘Object Files’’ describes the ELF object file format for the three main types of object files.
- Part 2, ‘‘Program Loading and Dynamic Linking’’ describes the object file information and system actions that create running programs.
- Part 3, ‘‘C Library’’ lists the symbols contained in `libsys`, the standard ANSI C and `libc` routines, and the global data symbols required by the `libc` routines.

**NOTE** References to X86 architecture have been changed to Intel Architecture.

# 1 OBJECT FILES

# Introduction

Part 1 describes the iABI object file format, called ELF (Executable and Linking Format). There are three main types of object files.

- A *relocatable file* holds code and data suitable for linking with other object files to create an executable or a shared object file.
- An *executable file* holds a program suitable for execution; the file specifies how `exec`(BA_OS) creates a program's process image.
- A *shared object file* holds code and data suitable for linking in two contexts. First, the link editor [see `ld`(SD_CMD)] may process it with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image.

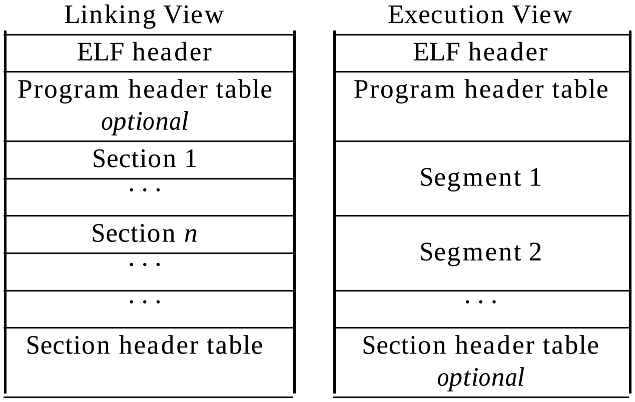
Created by the assembler and link editor, object files are binary representations of programs intended to execute directly on a processor. Programs that require other abstract machines, such as shell scripts, are excluded.

After the introductory material, Part 1 focuses on the file format and how it pertains to building programs. Part 2 also describes parts of the object file, concentrating on the information necessary to execute a program.

# File Format

Object files participate in program linking (building a program) and program execution (running a program). For convenience and efficiency, the object file format provides parallel views of a file's contents, reflecting the differing needs of these activities. Figure 1-1 shows an object file's organization.

**Figure 1-1: Object File Format**



An *ELF header* resides at the beginning and holds a ''road map'' describing the file's organization. *Sections* hold the bulk of object file information for the linking view: instructions, data, symbol table, relocation information, and so on. Descriptions of special sections appear later in Part 1. Part 2 discusses *segments* and the program execution view of the file.

A *program header table*, if present, tells the system how to create a process image. Files used to build a process image (execute a program) must have a program header table; relocatable files do not need one. A *section header table* contains information describing the file's sections. Every section has an entry in the table; each entry gives information such as the section name, the section size, etc. Files used during linking must have a section header table; other object files may or may not have one.

**NOTE** Although the figure shows the program header table immediately after the ELF header, and the section header table following the sections, actual files may differ. Moreover, sections and segments have no specified order. Only the ELF header has a fixed position in the file.

## Data Representation

As described here, the object file *format* supports various processors with 8-bit bytes and 32-bit architectures. Nevertheless, it is intended to be extensible to larger (or smaller) architectures. Object files therefore represent some control data with a machine-independent format, making it possible to identify object files and interpret their contents in a common way. Remaining data in an object file use the encoding of the target processor, regardless of the machine on which the file was created.

**Figure 1-2: 32-Bit Data Types**

| Name | Size | Alignment | Purpose |
|---|---|---|---|
| `Elf32_Addr` | 4 | 4 | Unsigned program address |
| `Elf32_Half` | 2 | 2 | Unsigned medium integer |
| `Elf32_Off` | 4 | 4 | Unsigned file offset |
| `Elf32_Sword` | 4 | 4 | Signed large integer |
| `Elf32_Word` | 4 | 4 | Unsigned large integer |
| `unsigned char` | 1 | 1 | Unsigned small integer |

All data structures that the object file format defines follow the ‘‘natural’’ size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 4-byte alignment for 4-byte objects, to force structure sizes to a multiple of 4, etc. Data also have suitable alignment from the beginning of the file. Thus, for example, a structure containing an `Elf32_Addr` member will be aligned on a 4-byte boundary within the file.

For portability reasons, ELF uses no bit-fields.

# ELF Header

Some object file control structures can grow, because the ELF header contains their actual sizes. If the object file format changes, a program may encounter control structures that are larger or smaller than expected. Programs might therefore ignore ‘‘extra’’ information. The treatment of ‘‘missing’’ information depends on context and will be specified when and if extensions are defined.

**Figure 1-3: ELF Header**

```
#define EI_NIDENT       16

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;
```

`e_ident` The initial bytes mark the file as an object file and provide machine-independent data with which to decode and interpret the file’s contents. Complete descriptions appear below, in ‘‘ELF Identification.’’

`e_type` This member identifies the object file type.

| Name | Value | Meaning |
|---|---|---|
| ET_NONE | 0 | No file type |
| ET_REL | 1 | Relocatable file |
| ET_EXEC | 2 | Executable file |
| ET_DYN | 3 | Shared object file |
| ET_CORE | 4 | Core file |
| ET_LOPROC | 0xff00 | Processor-specific |
| ET_HIPROC | 0xffff | Processor-specific |

Although the core file contents are unspecified, type `ET_CORE` is reserved to mark the file. Values from `ET_LOPROC` through `ET_HIPROC` (inclusive) are reserved for processor-specific semantics. Other values are reserved and will be assigned to new object file types as necessary.

e_machine This member's value specifies the required architecture for an individual file.

| Name | Value | Meaning |
|---|---|---|
| EM_NONE | 0 | No machine |
| EM_M32 | 1 | AT&T WE 32100 |
| EM_SPARC | 2 | SPARC |
| EM_386 | 3 | Intel 80386 |
| EM_68K | 4 | Motorola 68000 |
| EM_88K | 5 | Motorola 88000 |
| EM_860 | 7 | Intel 80860 |
| EM_MIPS | 8 | MIPS RS3000 |

Other values are reserved and will be assigned to new machines as necessary. Processor-specific ELF names use the machine name to distinguish them. For example, the flags mentioned below use the prefix EF_; a flag named WIDGET for the EM_XYZ machine would be called EF_XYZ_WIDGET.

e_version This member identifies the object file version.

| Name | Value | Meaning |
|---|---|---|
| EV_NONE | 0 | Invalid version |
| EV_CURRENT | 1 | Current version |

The value 1 signifies the original file format; extensions will create new versions with higher numbers. The value of EV_CURRENT, though given as 1 above, will change as necessary to reflect the current version number.

e_entry This member gives the virtual address to which the system first transfers control, thus starting the process. If the file has no associated entry point, this member holds zero.

e_phoff This member holds the program header table's file offset in bytes. If the file has no program header table, this member holds zero.

e_shoff This member holds the section header table's file offset in bytes. If the file has no section header table, this member holds zero.

e_flags This member holds processor-specific flags associated with the file. Flag names take the form EF_*machine_flag*. See ‘‘Machine Information’’ for flag definitions.

e_ehsize This member holds the ELF header's size in bytes.

e_phentsize This member holds the size in bytes of one entry in the file's program header table; all entries are the same size.

e_phnum This member holds the number of entries in the program header table. Thus the product of e_phentsize and e_phnum gives the table's size in bytes. If a file has no program header table, e_phnum holds the value zero.

e_shentsize This member holds a section header's size in bytes. A section header is one entry in the section header table; all entries are the same size.

e_shnum This member holds the number of entries in the section header table. Thus the product of e_shentsize and e_shnum gives the section header table's size in bytes. If a file has no section header table, e_shnum holds the value zero.

`e_shstrndx` This member holds the section header table index of the entry associated with the section name string table. If the file has no section name string table, this member holds the value `SHN_UNDEF`. See ‘‘Sections’’ and ‘‘String Table’’ below for more information.

## ELF Identification

As mentioned above, ELF provides an object file framework to support multiple processors, multiple data encodings, and multiple classes of machines. To support this object file family, the initial bytes of the file specify how to interpret the file, independent of the processor on which the inquiry is made and independent of the file’s remaining contents.

The initial bytes of an ELF header (and an object file) correspond to the `e_ident` member.

**Figure 1-4:** `e_ident[ ]` **Identification Indexes**

| Name | Value | Purpose |
|---|---|---|
| `EI_MAG0` | `0` | File identification |
| `EI_MAG1` | `1` | File identification |
| `EI_MAG2` | `2` | File identification |
| `EI_MAG3` | `3` | File identification |
| `EI_CLASS` | `4` | File class |
| `EI_DATA` | `5` | Data encoding |
| `EI_VERSION` | `6` | File version |
| `EI_PAD` | `7` | Start of padding bytes |
| `EI_NIDENT` | `16` | Size of `e_ident[]` |

These indexes access bytes that hold the following values.

`EI_MAG0` to `EI_MAG3`

A file’s first 4 bytes hold a ‘‘magic number,’’ identifying the file as an ELF object file.

| Name | Value | Position |
|---|---|---|
| `ELFMAG0` | `0x7f` | `e_ident[EI_MAG0]` |
| `ELFMAG1` | `’E’` | `e_ident[EI_MAG1]` |
| `ELFMAG2` | `’L’` | `e_ident[EI_MAG2]` |
| `ELFMAG3` | `’F’` | `e_ident[EI_MAG3]` |

`EI_CLASS` The next byte, `e_ident[EI_CLASS]`, identifies the file’s class, or capacity.

| Name | Value | Meaning |
|---|---|---|
| ELFCLASSNONE | 0 | Invalid class |
| ELFCLASS32 | 1 | 32-bit objects |
| ELFCLASS64 | 2 | 64-bit objects |

The file format is designed to be portable among machines of various sizes, without imposing the sizes of the largest machine on the smallest. Class ELFCLASS32 supports machines with files and virtual address spaces up to 4 gigabytes; it uses the basic types defined above.

Class ELFCLASS64 is reserved for 64-bit architectures. Its appearance here shows how the object file may change, but the 64-bit format is otherwise unspecified. Other classes will be defined as necessary, with different basic types and sizes for object file data.

EI_DATA Byte e_ident[EI_DATA] specifies the data encoding of the processor-specific data in the object file. The following encodings are currently defined.

| Name | Value | Meaning |
|---|---|---|
| ELFDATANONE | 0 | Invalid data encoding |
| ELFDATA2LSB | 1 | See below |
| ELFDATA2MSB | 2 | See below |

More information on these encodings appears below. Other values are reserved and will be assigned to new encodings as necessary.

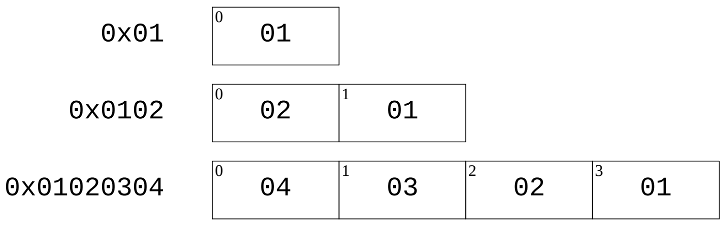
EI_VERSION Byte e_ident[EI_VERSION] specifies the ELF header version number. Currently, this value must be EV_CURRENT, as explained above for e_version.

EI_PAD This value marks the beginning of the unused bytes in e_ident. These bytes are reserved and set to zero; programs that read object files should ignore them. The value of EI_PAD will change in the future if currently unused bytes are given meanings.

A file's data encoding specifies how to interpret the basic objects in a file. As described above, class ELFCLASS32 files use objects that occupy 1, 2, and 4 bytes. Under the defined encodings, objects are represented as shown below. Byte numbers appear in the upper left corners.

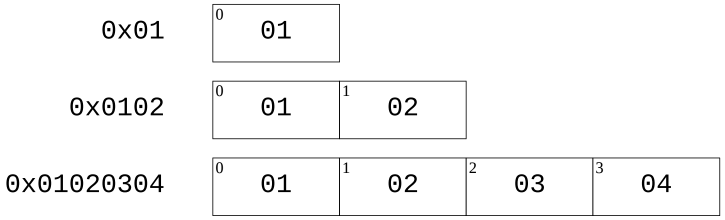
Encoding ELFDATA2LSB specifies 2's complement values, with the least significant byte occupying the lowest address.

**Figure 1-5: Data Encoding** ELFDATA2LSB



Encoding ELFDATA2MSB specifies 2's complement values, with the most significant byte occupying the lowest address.

**Figure 1-6: Data Encoding** ELFDATA2MSB



## Machine Information

For file identification in e_ident, the 32-bit Intel Architecture requires the following values.

**Figure 1-7: 32-bit Intel Architecture Identification,** e_ident

| Position | Value |
|---|---|
| e_ident[EI_CLASS] | ELFCLASS32 |
| e_ident[EI_DATA] | ELFDATA2LSB |

Processor identification resides in the ELF header's e_machine member and must have the value EM_386.

The ELF header's e_flags member holds bit flags associated with the file. The 32-bit Intel Architecture defines no flags; so this member contains zero.

# Sections

An object file's section header table lets one locate all the file's sections. The section header table is an array of Elf32_Shdr structures as described below. A section header table index is a subscript into this array. The ELF header's e_shoff member gives the byte offset from the beginning of the file to the section header table; e_shnum tells how many entries the section header table contains; e_shentsize gives the size in bytes of each entry.

Some section header table indexes are reserved; an object file will not have sections for these special indexes.

**Figure 1-8: Special Section Indexes**

| Name | Value |
|---|---|
| SHN_UNDEF | 0 |
| SHN_LORESERVE | 0xff00 |
| SHN_LOPROC | 0xff00 |
| SHN_HIPROC | 0xff1f |
| SHN_ABS | 0xfff1 |
| SHN_COMMON | 0xfff2 |
| SHN_HIRESERVE | 0xffff |

SHN_UNDEF This value marks an undefined, missing, irrelevant, or otherwise meaningless section reference. For example, a symbol ‘‘defined’’ relative to section number SHN_UNDEF is an undefined symbol.

NOTE Although index 0 is reserved as the undefined value, the section header table contains an entry for index 0. That is, if the e_shnum member of the ELF header says a file has 6 entries in the section header table, they have the indexes 0 through 5. The contents of the initial entry are specified later in this section.

SHN_LORESERVE This value specifies the lower bound of the range of reserved indexes.

SHN_LOPROC through SHN_HIPROC
Values in this inclusive range are reserved for processor-specific semantics.

SHN_ABS This value specifies absolute values for the corresponding reference. For example, symbols defined relative to section number SHN_ABS have absolute values and are not affected by relocation.

SHN_COMMON Symbols defined relative to this section are common symbols, such as FORTRAN COMMON or unallocated C external variables.

SHN_HIRESERVE This value specifies the upper bound of the range of reserved indexes. The system reserves indexes between SHN_LORESERVE and SHN_HIRESERVE, inclusive; the values do not reference the section header table. That is, the section header table does *not* contain entries for the reserved indexes.

Sections contain all information in an object file, except the ELF header, the program header table, and the section header table. Moreover, object files' sections satisfy several conditions.

- Every section in an object file has exactly one section header describing it. Section headers may exist that do not have a section.
- Each section occupies one contiguous (possibly empty) sequence of bytes within a file.
- Sections in a file may not overlap. No byte in a file resides in more than one section.
- An object file may have inactive space. The various headers and the sections might not ‘‘cover’’ every byte in an object file. The contents of the inactive data are unspecified.

A section header has the following structure.

**Figure 1-9: Section Header**

```
typedef struct {
        Elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
        Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

`sh_name` This member specifies the name of the section. Its value is an index into the section header string table section [see ‘‘String Table’’ below], giving the location of a null-terminated string.

`sh_type` This member categorizes the section’s contents and semantics. Section types and their descriptions appear below.

`sh_flags` Sections support 1-bit flags that describe miscellaneous attributes. Flag definitions appear below.

`sh_addr` If the section will appear in the memory image of a process, this member gives the address at which the section’s first byte should reside. Otherwise, the member contains 0.

`sh_offset` This member’s value gives the byte offset from the beginning of the file to the first byte in the section. One section type, `SHT_NOBITS` described below, occupies no space in the file, and its `sh_offset` member locates the conceptual placement in the file.

`sh_size` This member gives the section’s size in bytes. Unless the section type is `SHT_NOBITS`, the section occupies `sh_size` bytes in the file. A section of type `SHT_NOBITS` may have a non-zero size, but it occupies no space in the file.

`sh_link` This member holds a section header table index link, whose interpretation depends on the section type. A table below describes the values.

sh_info This member holds extra information, whose interpretation depends on the section type. A table below describes the values.

sh_addralign Some sections have address alignment constraints. For example, if a section holds a doubleword, the system must ensure doubleword alignment for the entire section. That is, the value of sh_addr must be congruent to 0, modulo the value of sh_addralign. Currently, only 0 and positive integral powers of two are allowed. Values 0 and 1 mean the section has no alignment constraints.

sh_entsize Some sections hold a table of fixed-size entries, such as a symbol table. For such a section, this member gives the size in bytes of each entry. The member contains 0 if the section does not hold a table of fixed-size entries.

A section header's sh_type member specifies the section's semantics.

**Figure 1-10: Section Types,** sh_type

| Name | Value |
|---|---:|
| SHT_NULL | 0 |
| SHT_PROGBITS | 1 |
| SHT_SYMTAB | 2 |
| SHT_STRTAB | 3 |
| SHT_RELA | 4 |
| SHT_HASH | 5 |
| SHT_DYNAMIC | 6 |
| SHT_NOTE | 7 |
| SHT_NOBITS | 8 |
| SHT_REL | 9 |
| SHT_SHLIB | 10 |
| SHT_DYNSYM | 11 |
| SHT_LOPROC | 0x70000000 |
| SHT_HIPROC | 0x7fffffff |
| SHT_LOUSER | 0x80000000 |
| SHT_HIUSER | 0xffffffff |

SHT_NULL This value marks the section header as inactive; it does not have an associated section. Other members of the section header have undefined values.

SHT_PROGBITS The section holds information defined by the program, whose format and meaning are determined solely by the program.

SHT_SYMTAB and SHT_DYNSYM

These sections hold a symbol table. Currently, an object file may have only one section of each type, but this restriction may be relaxed in the future. Typically, SHT_SYMTAB provides symbols for link editing, though it may also be used for dynamic linking. As a complete symbol table, it may contain many symbols unnecessary for dynamic linking. Consequently, an object file may also contain a SHT_DYNSYM section, which holds a minimal set of dynamic linking symbols, to save space. See ''Symbol Table'' below for details.

SHT_STRTAB The section holds a string table. An object file may have multiple string table sections. See ''String Table'' below for details.

SHT_RELA The section holds relocation entries with explicit addends, such as type Elf32_Rela for the 32-bit class of object files. An object file may have multiple relocation sections. See ''Relocation'' below for details.

SHT_HASH The section holds a symbol hash table. All objects participating in dynamic linking must contain a symbol hash table. Currently, an object file may have only one hash table, but this restriction may be relaxed in the future. See ''Hash Table'' in Part 2 for details.

SHT_DYNAMIC The section holds information for dynamic linking. Currently, an object file may have only one dynamic section, but this restriction may be relaxed in the future. See ''Dynamic Section'' in Part 2 for details.

SHT_NOTE The section holds information that marks the file in some way. See ''Note Section'' in Part 2 for details.

SHT_NOBITS A section of this type occupies no space in the file but otherwise resembles SHT_PROGBITS. Although this section contains no bytes, the sh_offset member contains the conceptual file offset.

SHT_REL The section holds relocation entries without explicit addends, such as type Elf32_Rel for the 32-bit class of object files. An object file may have multiple relocation sections. See ''Relocation'' below for details.

SHT_SHLIB This section type is reserved but has unspecified semantics. Programs that contain a section of this type do not conform to the ABI.

SHT_LOPROC through SHT_HIPROC
Values in this inclusive range are reserved for processor-specific semantics.

SHT_LOUSER This value specifies the lower bound of the range of indexes reserved for application programs.

SHT_HIUSER This value specifies the upper bound of the range of indexes reserved for application programs. Section types between SHT_LOUSER and SHT_HIUSER may be used by the application, without conflicting with current or future system-defined section types.

Other section type values are reserved. As mentioned before, the section header for index 0 (SHN_UNDEF) exists, even though the index marks undefined section references. This entry holds the following.

**Figure 1-11: Section Header Table Entry: Index 0**

| Name | Value | Note |
|---|---|---|
| sh_name | 0 | No name |
| sh_type | SHT_NULL | Inactive |
| sh_flags | 0 | No flags |
| sh_addr | 0 | No address |
| sh_offset | 0 | No file offset |
| sh_size | 0 | No size |

**Figure 1-11: Section Header Table Entry: Index 0** (continued)

| | | |
|---|---|---|
| `sh_link` | `SHN_UNDEF` | No link information |
| `sh_info` | `0` | No auxiliary information |
| `sh_addralign` | `0` | No alignment |
| `sh_entsize` | `0` | No entries |

A section header's `sh_flags` member holds 1-bit flags that describe the section's attributes. Defined values appear below; other values are reserved.

**Figure 1-12: Section Attribute Flags,** `sh_flags`

| Name | Value |
|---|---|
| `SHF_WRITE` | `0x1` |
| `SHF_ALLOC` | `0x2` |
| `SHF_EXECINSTR` | `0x4` |
| `SHF_MASKPROC` | `0xf0000000` |

If a flag bit is set in `sh_flags`, the attribute is ‘‘on’’ for the section. Otherwise, the attribute is ‘‘off’’ or does not apply. Undefined attributes are set to zero.

`SHF_WRITE` The section contains data that should be writable during process execution.

`SHF_ALLOC` The section occupies memory during process execution. Some control sections do not reside in the memory image of an object file; this attribute is off for those sections.

`SHF_EXECINSTR` The section contains executable machine instructions.

`SHF_MASKPROC` All bits included in this mask are reserved for processor-specific semantics.

Two members in the section header, `sh_link` and `sh_info`, hold special information, depending on section type.

**Figure 1-13: sh_link and sh_info Interpretation**

| sh_type | sh_link | sh_info |
|---|---|---|
| SHT_DYNAMIC | The section header index of the string table used by entries in the section. | 0 |
| SHT_HASH | The section header index of the symbol table to which the hash table applies. | 0 |
| SHT_REL<br>SHT_RELA | The section header index of the associated symbol table. | The section header index of the section to which the relocation applies. |
| SHT_SYMTAB<br>SHT_DYNSYM | The section header index of the associated string table. | One greater than the symbol table index of the last local symbol (binding STB_LOCAL). |
| other | SHN_UNDEF | 0 |

## Special Sections

Various sections hold program and control information. Sections in the list below are used by the system and have the indicated types and attributes.

**Figure 1-14: Special Sections**

| Name | Type | Attributes |
|---|---|---|
| .bss | SHT_NOBITS | SHF_ALLOC + SHF_WRITE |
| .comment | SHT_PROGBITS | none |
| .data | SHT_PROGBITS | SHF_ALLOC + SHF_WRITE |
| .data1 | SHT_PROGBITS | SHF_ALLOC + SHF_WRITE |
| .debug | SHT_PROGBITS | none |
| .dynamic | SHT_DYNAMIC | see below |
| .dynstr | SHT_STRTAB | SHF_ALLOC |
| .dynsym | SHT_DYNSYM | SHF_ALLOC |
| .fini | SHT_PROGBITS | SHF_ALLOC + SHF_EXECINSTR |
| .got | SHT_PROGBITS | see below |
| .hash | SHT_HASH | SHF_ALLOC |
| .init | SHT_PROGBITS | SHF_ALLOC + SHF_EXECINSTR |
| .interp | SHT_PROGBITS | see below |
| .line | SHT_PROGBITS | none |
| .note | SHT_NOTE | none |
| .plt | SHT_PROGBITS | see below |
| .rel*name* | SHT_REL | see below |

**Figure 1-14: Special Sections** (continued)

| | | |
|---|---|---|
| `.rela`*name* | `SHT_RELA` | see below |
| `.rodata` | `SHT_PROGBITS` | `SHF_ALLOC` |
| `.rodata1` | `SHT_PROGBITS` | `SHF_ALLOC` |
| `.shstrtab` | `SHT_STRTAB` | none |
| `.strtab` | `SHT_STRTAB` | see below |
| `.symtab` | `SHT_SYMTAB` | see below |
| `.text` | `SHT_PROGBITS` | `SHF_ALLOC` + `SHF_EXECINSTR` |

`.bss` This section holds uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. The section occupies no file space, as indicated by the section type, `SHT_NOBITS`.

`.comment` This section holds version control information.

`.data` and `.data1`
These sections hold initialized data that contribute to the program's memory image.

`.debug` This section holds information for symbolic debugging. The contents are unspecified.

`.dynamic` This section holds dynamic linking information. The section's attributes will include the `SHF_ALLOC` bit. Whether the `SHF_WRITE` bit is set is processor specific. See Part 2 for more information.

`.dynstr` This section holds strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries. See Part 2 for more information.

`.dynsym` This section holds the dynamic linking symbol table, as ''Symbol Table'' describes. See Part 2 for more information.

`.fini` This section holds executable instructions that contribute to the process termination code. That is, when a program exits normally, the system arranges to execute the code in this section.

`.got` This section holds the global offset table. See ''Special Sections'' in Part 1 and ''Global Offset Table'' in Part 2 for more information.

`.hash` This section holds a symbol hash table. See ''Hash Table'' in Part 2 for more information.

`.init` This section holds executable instructions that contribute to the process initialization code. That is, when a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called `main` for C programs).

`.interp` This section holds the path name of a program interpreter. If the file has a loadable segment that includes the section, the section's attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off. See Part 2 for more information.

`.line` This section holds line number information for symbolic debugging, which describes the correspondence between the source program and the machine code. The contents are unspecified.

`.note` This section holds information in the format that ‘‘Note Section’’ in Part 2 describes.

`.plt` This section holds the procedure linkage table. See ‘‘Special Sections’’ in Part 1 and ‘‘Procedure Linkage Table’’ in Part 2 for more information.

`.rel`*name* and `.rela`*name*

These sections hold relocation information, as ‘‘Relocation’’ below describes. If the file has a loadable segment that includes relocation, the sections’ attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off. Conventionally, *name* is supplied by the section to which the relocations apply. Thus a relocation section for `.text` normally would have the name `.rel.text` or `.rela.text`.

`.rodata` and `.rodata1`

These sections hold read-only data that typically contribute to a non-writable segment in the process image. See ‘‘Program Header’’ in Part 2 for more information.

`.shstrtab` This section holds section names.

`.strtab` This section holds strings, most commonly the strings that represent the names associated with symbol table entries. If the file has a loadable segment that includes the symbol string table, the section’s attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off.

`.symtab` This section holds a symbol table, as ‘‘Symbol Table’’ in this section describes. If the file has a loadable segment that includes the symbol table, the section’s attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off.

`.text` This section holds the ‘‘text,’’ or executable instructions, of a program.

Section names with a dot (`.`) prefix are reserved for the system, although applications may use these sections if their existing meanings are satisfactory. Applications may use names without the prefix to avoid conflicts with system sections. The object file format lets one define sections not in the list above. An object file may have more than one section with the same name.

Section names reserved for a processor architecture are formed by placing an abbreviation of the architecture name ahead of the section name. The name should be taken from the architecture names used for `e_machine`. For instance .FOO.psect is the psect section defined by the FOO architecture. Existing extensions are called by their historical names.

Pre-existing Extensions

`.sdata`
`.sbss`
`.lit8`
`.gptab`
`.conflict`
`.tdesc`
`.lit4`
`.reginfo`
`.liblist`

# String Table

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. One references a string as an index into the string table section. The first byte, which is index zero, is defined to hold a null character. Likewise, a string table's last byte is defined to hold a null character, ensuring null termination for all strings. A string whose index is zero specifies either no name or a null name, depending on the context. An empty string table section is permitted; its section header's `sh_size` member would contain zero. Non-zero indexes are invalid for an empty string table.

A section header's `sh_name` member holds an index into the section header string table section, as designated by the `e_shstrndx` member of the ELF header. The following figures show a string table with 25 bytes and the strings associated with various indexes.

| Index | +0 | +1 | +2 | +3 | +4 | +5 | +6 | +7 | +8 | +9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | `\0` | `n` | `a` | `m` | `e` | `.` | `\0` | `V` | `a` | `r` |
| 10 | `i` | `a` | `b` | `l` | `e` | `\0` | `a` | `b` | `l` | `e` |
| 20 | `\0` | `\0` | `x` | `x` | `\0` | | | | | |

**Figure 1-15: String Table Indexes**

| Index | String |
|---|---|
| 0 | *none* |
| 1 | `name.` |
| 7 | `Variable` |
| 11 | `able` |
| 16 | `able` |
| 24 | *null string* |

As the example shows, a string table index may refer to any byte in the section. A string may appear more than once; references to substrings may exist; and a single string may be referenced multiple times. Unreferenced strings also are allowed.

# Symbol Table

An object file's symbol table holds information needed to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 both designates the first entry in the table and serves as the undefined symbol index. The contents of the initial entry are specified later in this section.

| Name | Value |
|---|---|
| STN_UNDEF | 0 |

A symbol table entry has the following format.

**Figure 1-16: Symbol Table Entry**

```
typedef struct {
        Elf32_Word      st_name;
        Elf32_Addr      st_value;
        Elf32_Word      st_size;
        unsigned char   st_info;
        unsigned char   st_other;
        Elf32_Half      st_shndx;
} Elf32_Sym;
```

`st_name` This member holds an index into the object file's symbol string table, which holds the character representations of the symbol names. If the value is non-zero, it represents a string table index that gives the symbol name. Otherwise, the symbol table entry has no name.

**NOTE** External C symbols have the same names in C and object files' symbol tables.

`st_value` This member gives the value of the associated symbol. Depending on the context, this may be an absolute value, an address, etc.; details appear below.

`st_size` Many symbols have associated sizes. For example, a data object's size is the number of bytes contained in the object. This member holds 0 if the symbol has no size or an unknown size.

`st_info` This member specifies the symbol's type and binding attributes. A list of the values and meanings appears below. The following code shows how to manipulate the values.

```
#define ELF32_ST_BIND(i)   ((i)>>4)
#define ELF32_ST_TYPE(i)   ((i)&0xf)
#define ELF32_ST_INFO(b,t) (((b)<<4)+((t)&0xf))
```

st_other This member currently holds 0 and has no defined meaning.

st_shndx Every symbol table entry is ‘‘defined’’ in relation to some section; this member holds the relevant section header table index. As Figure 1-7 and the related text describe, some section indexes indicate special meanings.

A symbol’s binding determines the linkage visibility and behavior.

**Figure 1-17: Symbol Binding,** ELF32_ST_BIND

| Name | Value |
|---|---|
| STB_LOCAL | 0 |
| STB_GLOBAL | 1 |
| STB_WEAK | 2 |
| STB_LOPROC | 13 |
| STB_HIPROC | 15 |

STB_LOCAL Local symbols are not visible outside the object file containing their definition. Local symbols of the same name may exist in multiple files without interfering with each other.

STB_GLOBAL Global symbols are visible to all object files being combined. One file’s definition of a global symbol will satisfy another file’s undefined reference to the same global symbol.

STB_WEAK Weak symbols resemble global symbols, but their definitions have lower precedence.

STB_LOPROC through STB_HIPROC
Values in this inclusive range are reserved for processor-specific semantics.

Global and weak symbols differ in two major ways.

- When the link editor combines several relocatable object files, it does not allow multiple definitions of STB_GLOBAL symbols with the same name. On the other hand, if a defined global symbol exists, the appearance of a weak symbol with the same name will not cause an error. The link editor honors the global definition and ignores the weak ones. Similarly, if a common symbol exists (i.e., a symbol whose st_shndx field holds SHN_COMMON), the appearance of a weak symbol with the same name will not cause an error. The link editor honors the common definition and ignores the weak ones.
- When the link editor searches archive libraries, it extracts archive members that contain definitions of undefined global symbols. The member’s definition may be either a global or a weak symbol. The link editor does *not* extract archive members to resolve undefined weak symbols. Unresolved weak symbols have a zero value.

In each symbol table, all symbols with STB_LOCAL binding precede the weak and global symbols. As ‘‘Sections’’ above describes, a symbol table section’s sh_info section header member holds the symbol table index for the first non-local symbol.

A symbol's type provides a general classification for the associated entity.

**Figure 1-18: Symbol Types,** ELF32_ST_TYPE

| Name | Value |
|---|---|
| STT_NOTYPE | 0 |
| STT_OBJECT | 1 |
| STT_FUNC | 2 |
| STT_SECTION | 3 |
| STT_FILE | 4 |
| STT_LOPROC | 13 |
| STT_HIPROC | 15 |

STT_NOTYPE The symbol's type is not specified.

STT_OBJECT The symbol is associated with a data object, such as a variable, an array, etc.

STT_FUNC The symbol is associated with a function or other executable code.

STT_SECTION The symbol is associated with a section. Symbol table entries of this type exist primarily for relocation and normally have STB_LOCAL binding.

STT_FILE Conventionally, the symbol's name gives the name of the source file associated with the object file. A file symbol has STB_LOCAL binding, its section index is SHN_ABS, and it precedes the other STB_LOCAL symbols for the file, if it is present.

STT_LOPROC through STT_HIPROC
Values in this inclusive range are reserved for processor-specific semantics.

Function symbols (those with type STT_FUNC) in shared object files have special significance. When another object file references a function from a shared object, the link editor automatically creates a procedure linkage table entry for the referenced symbol. Shared object symbols with types other than STT_FUNC will not be referenced automatically through the procedure linkage table.

If a symbol's value refers to a specific location within a section, its section index member, st_shndx, holds an index into the section header table. As the section moves during relocation, the symbol's value changes as well, and references to the symbol continue to ‘‘point’’ to the same location in the program. Some special section index values give other semantics.

SHN_ABS The symbol has an absolute value that will not change because of relocation.

SHN_COMMON The symbol labels a common block that has not yet been allocated. The symbol's value gives alignment constraints, similar to a section's sh_addralign member. That is, the link editor will allocate the storage for the symbol at an address that is a multiple of st_value. The symbol's size tells how many bytes are required.

SHN_UNDEF This section table index means the symbol is undefined. When the link editor combines this object file with another that defines the indicated symbol, this file's references to the symbol will be linked to the actual definition.

As mentioned above, the symbol table entry for index 0 (STN_UNDEF) is reserved; it holds the following.

**Figure 1-19: Symbol Table Entry: Index 0**

| Name | Value | Note |
|---|---|---|
| st_name | 0 | No name |
| st_value | 0 | Zero value |
| st_size | 0 | No size |
| st_info | 0 | No type, local binding |
| st_other | 0 | |
| st_shndx | SHN_UNDEF | No section |

## Symbol Values

Symbol table entries for different object file types have slightly different interpretations for the st_value member.

- In relocatable files, st_value holds alignment constraints for a symbol whose section index is SHN_COMMON.
- In relocatable files, st_value holds a section offset for a defined symbol. That is, st_value is an offset from the beginning of the section that st_shndx identifies.
- In executable and shared object files, st_value holds a virtual address. To make these files' symbols more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation) for which the section number is irrelevant.

Although the symbol table values have similar meanings for different object files, the data allow efficient access by the appropriate programs.

# Relocation

Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. In other words, relocatable files must have information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. *Relocation entries* are these data.

**Figure 1-20: Relocation Entries**

```
typedef struct {
        Elf32_Addr      r_offset;
        Elf32_Word      r_info;
} Elf32_Rel;

typedef struct {
        Elf32_Addr      r_offset;
        Elf32_Word      r_info;
        Elf32_Sword     r_addend;
} Elf32_Rela;
```

`r_offset` This member gives the location at which to apply the relocation action. For a relocatable file, the value is the byte offset from the beginning of the section to the storage unit affected by the relocation. For an executable file or a shared object, the value is the virtual address of the storage unit affected by the relocation.

`r_info` This member gives both the symbol table index with respect to which the relocation must be made, and the type of relocation to apply. For example, a call instruction's relocation entry would hold the symbol table index of the function being called. If the index is `STN_UNDEF`, the undefined symbol index, the relocation uses 0 as the ''symbol value.'' Relocation types are processor-specific. When the text refers to a relocation entry's relocation type or symbol table index, it means the result of applying `ELF32_R_TYPE` or `ELF32_R_SYM`, respectively, to the entry's `r_info` member.

```
#define ELF32_R_SYM(i)    ((i)>>8)
#define ELF32_R_TYPE(i)   ((unsigned char)(i))
#define ELF32_R_INFO(s,t) (((s)<<8)+(unsigned char)(t))
```

`r_addend` This member specifies a constant addend used to compute the value to be stored into the relocatable field.

As shown above, only `Elf32_Rela` entries contain an explicit addend. Entries of type `Elf32_Rel` store an implicit addend in the location to be modified. Depending on the processor architecture, one form or the other might be necessary or more convenient. Consequently, an implementation for a particular machine may use one form exclusively or either form depending on context.

A relocation section references two other sections: a symbol table and a section to modify. The section header's `sh_info` and `sh_link` members, described in ''Sections'' above, specify these relationships. Relocation entries for different object files have slightly different interpretations for the `r_offset` member.

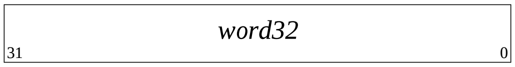
- In relocatable files, `r_offset` holds a section offset. That is, the relocation section itself describes how to modify another section in the file; relocation offsets designate a storage unit within the second section.
- In executable and shared object files, `r_offset` holds a virtual address. To make these files' relocation entries more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation).

Although the interpretation of `r_offset` changes for different object files to allow efficient access by the relevant programs, the relocation types' meanings stay the same.

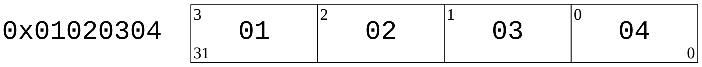
## Relocation Types

Relocation entries describe how to alter the following instruction and data fields (bit numbers appear in the lower box corners).

**Figure 1-21: Relocatable Fields**



*word32* This specifies a 32-bit field occupying 4 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the 32-bit Intel Architecture.



Calculations below assume the actions are transforming a relocatable file into either an executable or a shared object file. Conceptually, the link editor merges one or more relocatable files to form the output. It first decides how to combine and locate the input files, then updates the symbol values, and finally performs the relocation. Relocations applied to executable or shared object files are similar and accomplish the same result. Descriptions below use the following notation.

A This means the addend used to compute the value of the relocatable field.

B This means the base address at which a shared object has been loaded into memory during execution. Generally, a shared object file is built with a 0 base virtual address, but the execution address will be different.

G This means the offset into the global offset table at which the address of the relocation entry's symbol will reside during execution. See ''Global Offset Table'' in Part 2 for more information.

GOT This means the address of the global offset table. See ''Global Offset Table'' in Part 2 for more information.

L This means the place (section offset or address) of the procedure linkage table entry for a symbol. A procedure linkage table entry redirects a function call to the proper destination. The link editor builds the initial procedure linkage table, and the dynamic linker modifies the entries during execution. See ''Procedure Linkage Table'' in Part 2 for more information.

P This means the place (section offset or address) of the storage unit being relocated (computed using `r_offset`).

S This means the value of the symbol whose index resides in the relocation entry.

A relocation entry's `r_offset` value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. The SYSTEM V architecture uses only `Elf32_Rel` relocation entries, the field to be relocated holds the addend. In all cases, the addend and the computed result use the same byte order.

**Figure 1-22: Relocation Types**

| Name | Value | Field | Calculation |
|---|---|---|---|
| R_386_NONE | 0 | none | none |
| R_386_32 | 1 | *word32* | S + A |
| R_386_PC32 | 2 | *word32* | S + A - P |
| R_386_GOT32 | 3 | *word32* | G + A - P |
| R_386_PLT32 | 4 | *word32* | L + A - P |
| R_386_COPY | 5 | none | none |
| R_386_GLOB_DAT | 6 | *word32* | S |
| R_386_JMP_SLOT | 7 | *word32* | S |
| R_386_RELATIVE | 8 | *word32* | B + A |
| R_386_GOTOFF | 9 | *word32* | S + A - GOT |
| R_386_GOTPC | 10 | *word32* | GOT + A - P |

Some relocation types have semantics beyond simple calculation.

R_386_GOT32 This relocation type computes the distance from the base of the global offset table to the symbol's global offset table entry. It additionally instructs the link editor to build a global offset table.

R_386_PLT32 This relocation type computes the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a procedure linkage table.

R_386_COPY The link editor creates this relocation type for dynamic linking. Its offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current object file and in a shared object. During execution, the dynamic linker copies data associated with the shared object's symbol to the location specified by the offset.

R_386_GLOB_DAT This relocation type is used to set a global offset table entry to the address of the specified symbol. The special relocation type allows one to determine the correspondence between symbols and global offset table entries.

R_3862_JMP_SLOT The link editor creates this relocation type for dynamic linking. Its offset member gives the location of a procedure linkage table entry. The dynamic linker modifies the procedure linkage table entry to transfer control to the designated symbol's address [see ''Procedure Linkage Table'' in Part 2].

R_386_RELATIVE The link editor creates this relocation type for dynamic linking. Its offset member gives a location within a shared object that contains a value representing a relative address. The dynamic linker computes the corresponding virtual address by adding the virtual address at which the shared object was loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

R_386_GOTOFF This relocation type computes the difference between a symbol's value and the address of the global offset table. It additionally instructs the link editor to build the global offset table.

R_386_GOTPC This relocation type resembles R_386_PC32, except it uses the address of the global offset table in its calculation. The symbol referenced in this relocation normally is _GLOBAL_OFFSET_TABLE_, which additionally instructs the link editor to build the global offset table.

# 2 PROGRAM LOADING AND DYNAMIC LINKING

# Introduction

Part 2 describes the object file information and system actions that create running programs. Some information here applies to all systems; other information is processor-specific.

Executable and shared object files statically represent programs. To execute such programs, the system uses the files to create dynamic program representations, or process images. A process image has segments that hold its text, data, stack, and so on. The major sections in this part discuss the following.

- *Program header.* This section complements Part 1, describing object file structures that relate directly to program execution. The primary data structure, a program header table, locates segment images within the file and contains other information necessary to create the memory image for the program.
- *Program loading.* Given an object file, the system must load it into memory for the program to run.
- *Dynamic linking.* After the system loads the program, it must complete the process image by resolving symbolic references among the object files that compose the process.

**NOTE** There are naming conventions for ELF constants that have specified processor ranges. Names such as DT_, PT_, for processor-specific extensions, incorporate the name of the processor: DT_M32_SPECIAL, for example. Pre–existing processor extensions not using this convention will be supported.

| Pre-existing Extensions |
|---|
| DT_JMP_REL |

# Program Header

An executable or shared object file's program header table is an array of structures, each describing a segment or other information the system needs to prepare the program for execution. An object file *segment* contains one or more *sections*, as ''Segment Contents'' describes below. Program headers are meaningful only for executable and shared object files. A file specifies its own program header size with the ELF header's `e_phentsize` and `e_phnum` members [see ''ELF Header'' in Part 1].

**Figure 2-1: Program Header**

```
typedef struct {
        Elf32_Word      p_type;
        Elf32_Off       p_offset;
        Elf32_Addr      p_vaddr;
        Elf32_Addr      p_paddr;
        Elf32_Word      p_filesz;
        Elf32_Word      p_memsz;
        Elf32_Word      p_flags;
        Elf32_Word      p_align;
} Elf32_Phdr;
```

`p_type` This member tells what kind of segment this array element describes or how to interpret the array element's information. Type values and their meanings appear below.

`p_offset` This member gives the offset from the beginning of the file at which the first byte of the segment resides.

`p_vaddr` This member gives the virtual address at which the first byte of the segment resides in memory.

`p_paddr` On systems for which physical addressing is relevant, this member is reserved for the segment's physical address. Because System V ignores physical addressing for application programs, this member has unspecified contents for executable files and shared objects.

`p_filesz` This member gives the number of bytes in the file image of the segment; it may be zero.

`p_memsz` This member gives the number of bytes in the memory image of the segment; it may be zero.

`p_flags` This member gives flags relevant to the segment. Defined flag values appear below.

`p_align` As ''Program Loading'' later in this part describes, loadable process segments must have congruent values for `p_vaddr` and `p_offset`, modulo the page size. This member gives the value to which the segments are aligned in memory and in the file. Values 0 and 1 mean no alignment is required. Otherwise, `p_align` should be a positive, integral power of 2, and `p_vaddr` should equal `p_offset`, modulo `p_align`.

Some entries describe process segments; others give supplementary information and do not contribute to the process image. Segment entries may appear in any order, except as explicitly noted below. Defined type values follow; other values are reserved for future use.

**Figure 2-2: Segment Types,** `p_type`

| Name | Value |
|---|---:|
| PT_NULL | 0 |
| PT_LOAD | 1 |
| PT_DYNAMIC | 2 |
| PT_INTERP | 3 |
| PT_NOTE | 4 |
| PT_SHLIB | 5 |
| PT_PHDR | 6 |
| PT_LOPROC | 0x70000000 |
| PT_HIPROC | 0x7fffffff |

`PT_NULL` The array element is unused; other members' values are undefined. This type lets the program header table have ignored entries.

`PT_LOAD` The array element specifies a loadable segment, described by `p_filesz` and `p_memsz`. The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size (`p_memsz`) is larger than the file size (`p_filesz`), the ‘‘extra’’ bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size may not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the `p_vaddr` member.

`PT_DYNAMIC` The array element specifies dynamic linking information. See ‘‘Dynamic Section’’ below for more information.

`PT_INTERP` The array element specifies the location and size of a null-terminated path name to invoke as an interpreter. This segment type is meaningful only for executable files (though it may occur for shared objects); it may not occur more than once in a file. If it is present, it must precede any loadable segment entry. See ‘‘Program Interpreter’’ below for further information.

`PT_NOTE` The array element specifies the location and size of auxiliary information. See ‘‘Note Section’’ below for details.

`PT_SHLIB` This segment type is reserved but has unspecified semantics. Programs that contain an array element of this type do not conform to the ABI.

`PT_PHDR` The array element, if present, specifies the location and size of the program header table itself, both in the file and in the memory image of the program. This segment type may not occur more than once in a file. Moreover, it may occur only if the program header table is part of the memory image of the program. If it is present, it must precede any loadable segment entry. See ‘‘Program Interpreter’’ below for further information.

`PT_LOPROC` through `PT_HIPROC`
Values in this inclusive range are reserved for processor-specific semantics.

NOTE

Unless specifically required elsewhere, all program header segment types are optional. That is, a file's program header table may contain only those elements relevant to its contents.

## Base Address

Executable and shared object files have a *base address*, which is the lowest virtual address associated with the memory image of the program's object file. One use of the base address is to relocate the memory image of the program during dynamic linking.

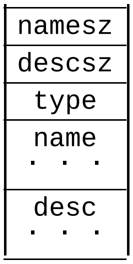
An executable or shared object file's base address is calculated during execution from three values: the memory load address, the maximum page size, and the lowest virtual address of a program's loadable segment. As ''Program Loading'' in this chapter describes, the virtual addresses in the program headers might not represent the actual virtual addresses of the program's memory image. To compute the base address, one determines the memory address associated with the lowest `p_vaddr` value for a `PT_LOAD` segment. One then obtains the base address by truncating the memory address to the nearest multiple of the maximum page size. Depending on the kind of file being loaded into memory, the memory address might or might not match the `p_vaddr` values.

As ''Sections'' in Part 1 describes, the `.bss` section has the type `SHT_NOBITS`. Although it occupies no space in the file, it contributes to the segment's memory image. Normally, these uninitialized data reside at the end of the segment, thereby making `p_memsz` larger than `p_filesz` in the associated program header element.

## Note Section

Sometimes a vendor or system builder needs to mark an object file with special information that other programs will check for conformance, compatibility, etc. Sections of type `SHT_NOTE` and program header elements of type `PT_NOTE` can be used for this purpose. The note information in sections and program header elements holds any number of entries, each of which is an array of 4-byte words in the format of the target processor. Labels appear below to help explain note information organization, but they are not part of the specification.

**Figure 2-3: Note Information**



`namesz` and `name`
: The first `namesz` bytes in `name` contain a null-terminated character representation of the entry's owner or originator. There is no formal mechanism for avoiding name conflicts. By convention, vendors use their own name, such as ‘‘XYZ Computer Company,’’ as the identifier. If no name is present, `namesz` contains 0. Padding is present, if necessary, to ensure 4-byte alignment for the descriptor. Such padding is not included in `namesz`.

`descsz` and `desc`
: The first `descsz` bytes in `desc` hold the note descriptor. The ABI places no constraints on a descriptor's contents. If no descriptor is present, `descsz` contains 0. Padding is present, if necessary, to ensure 4-byte alignment for the next note entry. Such padding is not included in `descsz`.

`type`
: This word gives the interpretation of the descriptor. Each originator controls its own types; multiple interpretations of a single type value may exist. Thus, a program must recognize both the name and the type to ‘‘understand’’ a descriptor. Types currently must be non-negative. The ABI does not define what descriptors mean.

To illustrate, the following note segment holds two entries.

**Figure 2-4: Example Note Segment**

| | +0 | +1 | +2 | +3 | |
|---|---|---|---|---|---|
| `namesz` | 7 | | | | |
| `descsz` | 0 | | | | No descriptor |
| `type` | 1 | | | | |
| `name` | X | Y | Z | | |
| | C | o | \0 | *pad* | |
| `namesz` | 7 | | | | |
| `descsz` | 8 | | | | |
| `type` | 3 | | | | |
| `name` | X | Y | Z | | |
| | C | o | \0 | *pad* | |
| `desc` | *word 0* | | | | |
| | *word 1* | | | | |

**NOTE** The system reserves note information with no name (`namesz==0`) and with a zero-length name (`name[0]=='\0'`) but currently defines no types. All other names must have at least one non-null character.

NOTE

Note information is optional. The presence of note information does not affect a program's ABI conformance, provided the information does not affect the program's execution behavior. Otherwise, the program does not conform to the ABI and has undefined behavior.

# Program Loading

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When—and if—the system physically reads the file depends on the program's execution behavior, system load, etc. A process does not require a physical page unless it references the logical page during execution, and processes commonly leave many pages unreferenced. Therefore delaying physical reads frequently obviates them, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size.

Virtual addresses and file offsets for the SYSTEM V architecture segments are congruent modulo 4 KB (`0x1000`) or larger powers of 2. Because 4 KB is the maximum page size, the files will be suitable for paging regardless of physical page size.

**Figure 2-5: Executable File**

| File Offset | File | Virtual Address |
|---:|:---:|:---|
| `0` | ELF header | |
| `Program header table` | | |
| | Other information | |
| `0x100` | Text segment | `0x8048100` |
| | . . . | |
| | `0x2be00` bytes | `0x8073eff` |
| `0x2bf00` | Data segment | `0x8074f00` |
| | . . . | |
| | `0x4e00` bytes | `0x8079cff` |
| `0x30d00` | Other information | |
| | . . . | |

**Figure 2-6: Program Header Segments**

| Member | Text | Data |
|:---|---:|---:|
| `p_type` | `PT_LOAD` | `PT_LOAD` |
| `p_offset` | `0x100` | `0x2bf00` |
| `p_vaddr` | `0x8048100` | `0x8074f00` |
| `p_paddr` | unspecified | unspecified |
| `p_filesz` | `0x2be00` | `0x4e00` |
| `p_memsz` | `0x2be00` | `0x5e24` |
| `p_flags` | `PF_R + PF_X` | `PF_R + PF_W + PF_X` |
| `p_align` | `0x1000` | `0x1000` |

Although the example's file offsets and virtual addresses are congruent modulo 4 KB for both text and data, up to four file pages hold impure text or data (depending on page size and file system block size).

- The first text page contains the ELF header, the program header table, and other information.

- The last text page holds a copy of the beginning of data.
- The first data page has a copy of the end of text.
- The last data page may contain file information not relevant to the running process.

Logically, the system enforces the memory permissions as if each segment were complete and separate; segments' addresses are adjusted to ensure each logical page in the address space has a single set of permissions. In the example above, the region of the file holding the end of text and the beginning of data will be mapped twice: at one virtual address for text and at a different virtual address for data.

The end of the data segment requires special handling for uninitialized data, which the system defines to begin with zero values. Thus if a file's last data page includes information not in the logical memory page, the extraneous data must be set to zero, not the unknown contents of the executable file. ''Impurities'' in the other three pages are not logically part of the process image; whether the system expunges them is unspecified. The memory image for this program follows, assuming 4 KB (`0x1000`) pages.

**Figure 2-7: Process Image Segments**

| Virtual Address | Contents | Segment |
|---|---|---|
| `0x8048000` | *Header padding*<br>`0x100` bytes | Text |
| `0x8048100` | Text segment<br>. . .<br>`0x2be00` bytes | |
| `0x8073f00` | *Data padding*<br>`0x100` bytes | |
| `0x8074000` | *Text padding*<br>`0xf00` bytes | Data |
| `0x8074f00` | Data segment<br>. . .<br>`0x4e00` bytes | |
| `0x8079d00` | Uninitialized data<br>`0x1024` zero bytes | |
| `0x807ad24` | *Page padding*<br>`0x2dc` zero bytes | |

One aspect of segment loading differs between executable files and shared objects. Executable file segments typically contain absolute code. To let the process execute correctly, the segments must reside at the virtual addresses used to build the executable file. Thus the system uses the `p_vaddr` values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. This lets a segment's virtual address change from one process to another, without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the segments' *relative positions*. Because position-independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file. The following table shows possible shared object virtual address assignments for several processes, illustrating constant relative positioning. The table also illustrates the base address computations.

**Figure 2-8: Example Shared Object Segment Addresses**

| Sourc | Text | Data | Base Address |
|---|---:|---:|---:|
| File | `0x200` | `0x2a400` | `0x0` |
| Process 1 | `0x80000200` | `0x8002a400` | `0x80000000` |
| Process 2 | `0x80081200` | `0x800ab400` | `0x80081000` |
| Process 3 | `0x900c0200` | `0x900ea400` | `0x900c0000` |
| Process 4 | `0x900c6200` | `0x900f0400` | `0x900c6000` |

# Dynamic Linking

## Program Interpreter

An executable file may have one PT_INTERP program header element. During exec(BA_OS), the system retrieves a path name from the PT_INTERP segment and creates the initial process image from the interpreter file's segments. That is, instead of using the original executable file's segment images, the system composes a memory image for the interpreter. It then is the interpreter's responsibility to receive control from the system and provide an environment for the application program.

The interpreter receives control in one of two ways. First, it may receive a file descriptor to read the executable file, positioned at the beginning. It can use this file descriptor to read and/or map the executable file's segments into memory. Second, depending on the executable file format, the system may load the executable file into memory instead of giving the interpreter an open file descriptor. With the possible exception of the file descriptor, the interpreter's initial process state matches what the executable file would have received. The interpreter itself may not require a second interpreter. An interpreter may be either a shared object or an executable file.

- A shared object (the normal case) is loaded as position-independent, with addresses that may vary from one process to another; the system creates its segments in the dynamic segment area used by mmap(KE_OS) and related services. Consequently, a shared object interpreter typically will not conflict with the original executable file's original segment addresses.
- An executable file is loaded at fixed addresses; the system creates its segments using the virtual addresses from the program header table. Consequently, an executable file interpreter's virtual addresses may collide with the first executable file; the interpreter is responsible for resolving conflicts.

## Dynamic Linker

When building an executable file that uses dynamic linking, the link editor adds a program header element of type PT_INTERP to an executable file, telling the system to invoke the dynamic linker as the program interpreter.

**NOTE** The locations of the system provided dynamic linkers are processor–specific.

Exec(BA_OS) and the dynamic linker cooperate to create the process image for the program, which entails the following actions:

- Adding the executable file's memory segments to the process image;
- Adding shared object memory segments to the process image;
- Performing relocations for the executable file and its shared objects;
- Closing the file descriptor that was used to read the executable file, if one was given to the dynamic linker;
- Transferring control to the program, making it look as if the program had received control directly from exec(BA_OS).

The link editor also constructs various data that assist the dynamic linker for executable and shared object files. As shown above in ‘‘Program Header,’’ these data reside in loadable segments, making them available during execution. (Once again, recall the exact segment contents are processor-specific. See the processor supplement for complete information.)

- A .dynamic section with type SHT_DYNAMIC holds various data. The structure residing at the beginning of the section holds the addresses of other dynamic linking information.
- The .hash section with type SHT_HASH holds a symbol hash table.
- The .got and .plt sections with type SHT_PROGBITS hold two separate tables: the global offset table and the procedure linkage table. Sections below explain how the dynamic linker uses and changes the tables to create memory images for object files.

Because every ABI-conforming program imports the basic system services from a shared object library, the dynamic linker participates in every ABI-conforming program execution.

As ‘‘Program Loading’’ explains in the processor supplement, shared objects may occupy virtual memory addresses that are different from the addresses recorded in the file’s program header table. The dynamic linker relocates the memory image, updating absolute addresses before the application gains control. Although the absolute address values would be correct if the library were loaded at the addresses specified in the program header table, this normally is not the case.

If the process environment [see exec(BA_OS)] contains a variable named LD_BIND_NOW with a non-null value, the dynamic linker processes all relocation before transferring control to the program. For example, all the following environment entries would specify this behavior.

- LD_BIND_NOW=1
- LD_BIND_NOW=on
- LD_BIND_NOW=off

Otherwise, LD_BIND_NOW either does not occur in the environment or has a null value. The dynamic linker is permitted to evaluate procedure linkage table entries lazily, thus avoiding symbol resolution and relocation overhead for functions that are not called. See ‘‘Procedure Linkage Table’’ in this part for more information.

## Dynamic Section

If an object file participates in dynamic linking, its program header table will have an element of type PT_DYNAMIC. This ‘‘segment’’ contains the .dynamic section. A special symbol, _DYNAMIC, labels the section, which contains an array of the following structures.

**Figure 2-9: Dynamic Structure**

```
typedef struct {
        Elf32_Sword     d_tag;
        union {
                Elf32_Word      d_val;
                Elf32_Addr      d_ptr;
        } d_un;
} Elf32_Dyn;

extern Elf32_Dyn _DYNAMIC[];
```

For each object with this type, `d_tag` controls the interpretation of `d_un`.

`d_val` These `Elf32_Word` objects represent integer values with various interpretations.

`d_ptr` These `Elf32_Addr` objects represent program virtual addresses. As mentioned previously, a file's virtual addresses might not match the memory virtual addresses during execution. When interpreting addresses contained in the dynamic structure, the dynamic linker computes actual addresses, based on the original file value and the memory base address. For consistency, files do *not* contain relocation entries to ‘‘correct’’ addresses in the dynamic structure.

The following table summarizes the tag requirements for executable and shared object files. If a tag is marked ‘‘mandatory,’’ then the dynamic linking array for an ABI-conforming file must have an entry of that type. Likewise, ‘‘optional’’ means an entry for the tag may appear but is not required.

**Figure 2-10: Dynamic Array Tags,** `d_tag`

| Name | Value | `d_un` | Executable | Shared Object |
|---|---:|---|---|---|
| `DT_NULL` | 0 | ignored | mandatory | mandatory |
| `DT_NEEDED` | 1 | `d_val` | optional | optional |
| `DT_PLTRELSZ` | 2 | `d_val` | optional | optional |
| `DT_PLTGOT` | 3 | `d_ptr` | optional | optional |
| `DT_HASH` | 4 | `d_ptr` | mandatory | mandatory |
| `DT_STRTAB` | 5 | `d_ptr` | mandatory | mandatory |
| `DT_SYMTAB` | 6 | `d_ptr` | mandatory | mandatory |
| `DT_RELA` | 7 | `d_ptr` | mandatory | optional |
| `DT_RELASZ` | 8 | `d_val` | mandatory | optional |
| `DT_RELAENT` | 9 | `d_val` | mandatory | optional |
| `DT_STRSZ` | 10 | `d_val` | mandatory | mandatory |
| `DT_SYMENT` | 11 | `d_val` | mandatory | mandatory |
| `DT_INIT` | 12 | `d_ptr` | optional | optional |
| `DT_FINI` | 13 | `d_ptr` | optional | optional |
| `DT_SONAME` | 14 | `d_val` | ignored | optional |
| `DT_RPATH` | 15 | `d_val` | optional | ignored |
| `DT_SYMBOLIC` | 16 | ignored | ignored | optional |

**Figure 2-10: Dynamic Array Tags,** `d_tag` (continued)

| Name | Value | d_un | Executable | Shared Object |
|---|---:|---|---|---|
| DT_REL | 17 | d_ptr | mandatory | optional |
| DT_RELSZ | 18 | d_val | mandatory | optional |
| DT_RELENT | 19 | d_val | mandatory | optional |
| DT_PLTREL | 20 | d_val | optional | optional |
| DT_DEBUG | 21 | d_ptr | optional | ignored |
| DT_TEXTREL | 22 | ignored | optional | optional |
| DT_JMPREL | 23 | d_ptr | optional | optional |
| DT_LOPROC | 0x70000000 | unspecified | unspecified | unspecified |
| DT_HIPROC | 0x7fffffff | unspecified | unspecified | unspecified |

DT_NULL An entry with a DT_NULL tag marks the end of the _DYNAMIC array.

DT_NEEDED This element holds the string table offset of a null-terminated string, giving the name of a needed library. The offset is an index into the table recorded in the DT_STRTAB entry. See ‘‘Shared Object Dependencies’’ for more information about these names. The dynamic array may contain multiple entries with this type. These entries’ relative order is significant, though their relation to entries of other types is not.

DT_PLTRELSZ This element holds the total size, in bytes, of the relocation entries associated with the procedure linkage table. If an entry of type DT_JMPREL is present, a DT_PLTRELSZ must accompany it.

DT_PLTGOT This element holds an address associated with the procedure linkage table and/or the global offset table. See this section in the processor supplement for details.

DT_HASH This element holds the address of the symbol hash table, described in ‘‘Hash Table.’’ This hash table refers to the symbol table referenced by the DT_SYMTAB element.

DT_STRTAB This element holds the address of the string table, described in Part 1. Symbol names, library names, and other strings reside in this table.

DT_SYMTAB This element holds the address of the symbol table, described in Part 1, with Elf32_Sym entries for the 32-bit class of files.

DT_RELA This element holds the address of a relocation table, described in Part 1. Entries in the table have explicit addends, such as Elf32_Rela for the 32-bit file class. An object file may have multiple relocation sections. When building the relocation table for an executable or shared object file, the link editor catenates those sections to form a single table. Although the sections remain independent in the object file, the dynamic linker sees a single table. When the dynamic linker creates the process image for an executable file or adds a shared object to the process image, it reads the relocation table and performs the associated actions. If this element is present, the dynamic structure must also have DT_RELASZ and DT_RELAENT elements. When relocation is ‘‘mandatory’’ for a file, either DT_RELA or DT_REL may occur (both are permitted but not required).

DT_RELASZ This element holds the total size, in bytes, of the DT_RELA relocation table.

DT_RELAENT This element holds the size, in bytes, of the DT_RELA relocation entry.

DT_STRSZ This element holds the size, in bytes, of the string table.

DT_SYMENT This element holds the size, in bytes, of a symbol table entry.

DT_INIT This element holds the address of the initialization function, discussed in ‘‘Initialization and Termination Functions’’ below.

DT_FINI This element holds the address of the termination function, discussed in ‘‘Initialization and Termination Functions’’ below.

DT_SONAME This element holds the string table offset of a null-terminated string, giving the name of the shared object. The offset is an index into the table recorded in the DT_STRTAB entry. See ‘‘Shared Object Dependencies’’ below for more information about these names.

DT_RPATH This element holds the string table offset of a null-terminated search library search path string, discussed in ‘‘Shared Object Dependencies.’’ The offset is an index into the table recorded in the DT_STRTAB entry.

DT_SYMBOLIC This element’s presence in a shared object library alters the dynamic linker’s symbol resolution algorithm for references within the library. Instead of starting a symbol search with the executable file, the dynamic linker starts from the shared object itself. If the shared object fails to supply the referenced symbol, the dynamic linker then searches the executable file and other shared objects as usual.

DT_REL This element is similar to DT_RELA, except its table has implicit addends, such as Elf32_Rel for the 32-bit file class. If this element is present, the dynamic structure must also have DT_RELSZ and DT_RELENT elements.

DT_RELSZ This element holds the total size, in bytes, of the DT_REL relocation table.

DT_RELENT This element holds the size, in bytes, of the DT_REL relocation entry.

DT_PLTREL This member specifies the type of relocation entry to which the procedure linkage table refers. The d_val member holds DT_REL or DT_RELA, as appropriate. All relocations in a procedure linkage table must use the same relocation.

DT_DEBUG This member is used for debugging. Its contents are not specified for the ABI; programs that access this entry are not ABI-conforming.

DT_TEXTREL This member’s absence signifies that no relocation entry should cause a modification to a non-writable segment, as specified by the segment permissions in the program header table. If this member is present, one or more relocation entries might request modifications to a non-writable segment, and the dynamic linker can prepare accordingly.

DT_JMPREL If present, this entries’s d_ptr member holds the address of relocation entries associated solely with the procedure linkage table. Separating these relocation entries lets the dynamic linker ignore them during process initialization, if lazy binding is enabled. If this entry is present, the related entries of types DT_PLTRELSZ and DT_PLTREL must also be present.

DT_LOPROC through DT_HIPROC

Values in this inclusive range are reserved for processor-specific semantics.

Except for the DT_NULL element at the end of the array, and the relative order of DT_NEEDED elements, entries may appear in any order. Tag values not appearing in the table are reserved.

## Shared Object Dependencies

When the link editor processes an archive library, it extracts library members and copies them into the output object file. These statically linked services are available during execution without involving the dynamic linker. Shared objects also provide services, and the dynamic linker must attach the proper shared object files to the process image for execution. Thus executable and shared object files describe their specific dependencies.

When the dynamic linker creates the memory segments for an object file, the dependencies (recorded in DT_NEEDED entries of the dynamic structure) tell what shared objects are needed to supply the program's services. By repeatedly connecting referenced shared objects and their dependencies, the dynamic linker builds a complete process image. When resolving symbolic references, the dynamic linker examines the symbol tables with a breadth-first search. That is, it first looks at the symbol table of the executable program itself, then at the symbol tables of the DT_NEEDED entries (in order), then at the second level DT_NEEDED entries, and so on. Shared object files must be readable by the process; other permissions are not required.

**NOTE** Even when a shared object is referenced multiple times in the dependency list, the dynamic linker will connect the object only once to the process.

Names in the dependency list are copies either of the DT_SONAME strings or the path names of the shared objects used to build the object file. For example, if the link editor builds an executable file using one shared object with a DT_SONAME entry of `lib1` and another shared object library with the path name `/usr/lib/lib2`, the executable file will contain `lib1` and `/usr/lib/lib2` in its dependency list.

If a shared object name has one or more slash (`/`) characters anywhere in the name, such as `/usr/lib/lib2` above or `directory/file`, the dynamic linker uses that string directly as the path name. If the name has no slashes, such as `lib1` above, three facilities specify shared object path searching, with the following precedence.

- First, the dynamic array tag DT_RPATH may give a string that holds a list of directories, separated by colons (`:`). For example, the string `/home/dir/lib:/home/dir2/lib:` tells the dynamic linker to search first the directory `/home/dir/lib`, then `/home/dir2/lib`, and then the current directory to find dependencies.
- Second, a variable called LD_LIBRARY_PATH in the process environment [see `exec`(BA_OS)] may hold a list of directories as above, optionally followed by a semicolon (`;`) and another directory list. The following values would be equivalent to the previous example:
  - □ `LD_LIBRARY_PATH=/home/dir/lib:/home/dir2/lib:`
  - □ `LD_LIBRARY_PATH=/home/dir/lib;/home/dir2/lib:`
  - □ `LD_LIBRARY_PATH=/home/dir/lib:/home/dir2/lib:;`

  All LD_LIBRARY_PATH directories are searched after those from DT_RPATH. Although some programs (such as the link editor) treat the lists before and after the semicolon differently, the dynamic linker does not. Nevertheless, the dynamic linker accepts the semicolon notation, with the semantics described above.

- Finally, if the other two groups of directories fail to locate the desired library, the dynamic linker searches /usr/lib.

NOTE

For security, the dynamic linker ignores environmental search specifications (such as LD_LIBRARY_PATH) for set-user and set-group ID programs. It does, however, search DT_RPATH directories and /usr/lib.

## Global Offset Table

Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold absolute addresses in private data, thus making the addresses available without compromising the position-independence and sharability of a program's text. A program references its global offset table using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

Initially, the global offset table holds information as required by its relocation entries [see ''Relocation'' in Part 1]. After the system creates memory segments for a loadable object file, the dynamic linker processes the relocation entries, some of which will be type R_386_GLOB_DAT referring to the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the appropriate memory table entries to the proper values. Although the absolute addresses are unknown when the link editor builds an object file, the dynamic linker knows the addresses of all memory segments and can thus calculate the absolute addresses of the symbols contained therein.

If a program requires direct access to the absolute address of a symbol, that symbol will have a global offset table entry. Because the executable file and shared objects have separate global offset tables, a symbol's address may appear in several tables. The dynamic linker processes all the global offset table relocations before giving control to any code in the process image, thus ensuring the absolute addresses are available during execution.

The table's entry zero is reserved to hold the address of the dynamic structure, referenced with the symbol _DYNAMIC. This allows a program, such as the dynamic linker, to find its own dynamic structure without having yet processed its relocation entries. This is especially important for the dynamic linker, because it must initialize itself without relying on other programs to relocate its memory image. On the 32-bit Intel Architecture, entries one and two in the global offset table also are reserved. ''Procedure Linkage Table'' below describes them.

The system may choose different memory segment addresses for the same shared object in different programs; it may even choose different library addresses for different executions of the same program. Nonetheless, memory segments do not change addresses once the process image is established. As long as a process exists, its memory segments reside at fixed virtual addresses.

A global offset table's format and interpretation are processor-specific. For the 32-bit Intel Architecture, the symbol _GLOBAL_OFFSET_TABLE_ may be used to access the table.

Figure 2-11: Global Offset Table

```
extern Elf32_Addr       _GLOBAL_OFFSET_TABLE_[];
```

The symbol `_GLOBAL_OFFSET_TABLE_` may reside in the middle of the `.got` section, allowing both negative and non-negative ‘‘subscripts’’ into the array of addresses.

## Procedure Linkage Table

Much as the global offset table redirects position-independent address calculations to absolute locations, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. Consequently, the link editor arranges to have the program transfer control to entries in the procedure linkage table. On the SYSTEM V architecture, procedure linkage tables reside in shared text, but they use addresses in the private global offset table. The dynamic linker determines the destinations' absolute addresses and modifies the global offset table's memory image accordingly. The dynamic linker thus can redirect the entries without compromising the position-independence and sharability of the program's text. Executable files and shared object files have separate procedure linkage tables.

Figure 2-12: Absolute Procedure Linkage Table

```
.PLT0:pushl got_plus_4
      jmp   *got_plus_8
      nop; nop
      nop; nop
.PLT1:jmp   *name1_in_GOT
      pushl $offset@PC
.PLT2:jmp   *name2_in_GOT
      push  $offset
      jmp   .PLT0@PC
      ...
```

**Figure 2-13: Position-Independent Procedure Linkage Table**

```
.PLT0:pushl 4(%ebx)
      jmp   *8(%ebx)
      nop; nop
      nop; nop
.PLT1:jmp   *name1@GOT(%ebx)
      pushl $offset
      jmp   .PLT0@PC
.PLT2:jmp   *name2@GOT(%ebx)
      pushl $offset
      jmp   .PLT0@PC
      ...
```

**NOTE** As the figures show, the procedure linkage table instructions use different operand addressing modes for absolute code and for position-independent code. Nonetheless, their interfaces to the dynamic linker are the same.

Following the steps below, the dynamic linker and the program ‘‘cooperate’’ to resolve symbolic references through the procedure linkage table and the global offset table.

1. When first creating the memory image of the program, the dynamic linker sets the second and the third entries in the global offset table to special values. Steps below explain more about these values.
2. If the procedure linkage table is position-independent, the address of the global offset table must reside in `%ebx`. Each shared object file in the process image has its own procedure linkage table, and control transfers to a procedure linkage table entry only from within the same object file. Consequently, the calling function is responsible for setting the global offset table base register before calling the procedure linkage table entry.
3. For illustration, assume the program calls `name1`, which transfers control to the label `.PLT1`.
4. The first instruction jumps to the address in the global offset table entry for `name1`. Initially, the global offset table holds the address of the following `pushl` instruction, not the real address of `name1`.
5. Consequently, the program pushes a relocation offset (*offset*) on the stack. The relocation offset is a 32-bit, non-negative byte offset into the relocation table. The designated relocation entry will have type `R_386_JMP_SLOT`, and its offset will specify the global offset table entry used in the previous `jmp` instruction. The relocation entry also contains a symbol table index, thus telling the dynamic linker what symbol is being referenced, `name1` in this case.
6. After pushing the relocation offset, the program then jumps to `.PLT0`, the first entry in the procedure linkage table. The `pushl` instruction places the value of the second global offset table entry (*got_plus_4* or `4(%ebx)`) on the stack, thus giving the dynamic linker one word of identifying information. The program then jumps to the address in the third global offset table entry

(*got_plus_8* or `8(%ebx)`), which transfers control to the dynamic linker.

7. When the dynamic linker receives control, it unwinds the stack, looks at the designated relocation entry, finds the symbol's value, stores the ''real'' address for `name1` in its global offset table entry, and transfers control to the desired destination.
8. Subsequent executions of the procedure linkage table entry will transfer directly to `name1`, without calling the dynamic linker a second time. That is, the `jmp` instruction at `.PLT1` will transfer to `name1`, instead of ''falling through'' to the `pushl` instruction.

The `LD_BIND_NOW` environment variable can change dynamic linking behavior. If its value is non-null, the dynamic linker evaluates procedure linkage table entries before transferring control to the program. That is, the dynamic linker processes relocation entries of type `R_386_JMP_SLOT` during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

**NOTE**

Lazy binding generally improves overall application performance, because unused symbols do not incur the dynamic linking overhead. Nevertheless, two situations make lazy binding undesirable for some applications. First, the initial reference to a shared object function takes longer than subsequent calls, because the dynamic linker intercepts the call to resolve the symbol. Some applications cannot tolerate this unpredictability. Second, if an error occurs and the dynamic linker cannot resolve the symbol, the dynamic linker will terminate the program. Under lazy binding, this might occur at arbitrary times. Once again, some applications cannot tolerate this unpredictability. By turning off lazy binding, the dynamic linker forces the failure to occur during process initialization, before the application receives control.

## Hash Table

A hash table of `Elf32_Word` objects supports symbol table access. Labels appear below to help explain the hash table organization, but they are not part of the specification.

**Figure 2-14: Symbol Hash Table**

| |
|---|
| `nbucket` |
| `nchain` |
| `bucket[0]`<br>. . .<br>`bucket[nbucket - 1]` |
| `chain[0]`<br>. . .<br>`chain[nchain - 1]` |

The `bucket` array contains `nbucket` entries, and the `chain` array contains `nchain` entries; indexes start at 0. Both `bucket` and `chain` hold symbol table indexes. Chain table entries parallel the symbol table. The number of symbol table entries should equal `nchain`; so symbol table indexes also select chain table entries. A hashing function (shown below) accepts a symbol name and returns a value that may be used to compute a `bucket` index. Consequently, if the hashing function returns the value *x* for some name, `bucket[`*x*`%nbucket]` gives an index, *y*, into both the symbol table and the chain table. If the symbol table entry is not the one desired, `chain[`*y*`]` gives the next symbol table entry with the same hash value. One can follow the `chain` links until either the selected symbol table entry holds the desired name or the chain entry contains the value STN_UNDEF.

**Figure 2-15: Hashing Function**

```
unsigned long
elf_hash(const unsigned char *name)
{
        unsigned long    h = 0, g;

        while (*name)
        {
                h = (h << 4) + *name++;
                if (g = h & 0xf0000000)
                        h ^= g >> 24;
                h &= ~g;
        }
        return h;
}
```

## Initialization and Termination Functions

After the dynamic linker has built the process image and performed the relocations, each shared object gets the opportunity to execute some initialization code. These initialization functions are called in no specified order, but all shared object initializations happen before the executable file gains control.

Similarly, shared objects may have termination functions, which are executed with the atexit(BA_OS) mechanism after the base process begins its termination sequence. Once again, the order in which the dynamic linker calls termination functions is unspecified.

Shared objects designate their initialization and termination functions through the DT_INIT and DT_FINI entries in the dynamic structure, described in ‘‘Dynamic Section’’ above. Typically, the code for these functions resides in the .init and .fini sections, mentioned in ‘‘Sections’’ of Part 1.

**NOTE**

Although the atexit(BA_OS) termination processing normally will be done, it is not guaranteed to have executed upon process death. In particular, the process will not execute the termination processing if it calls _exit [see exit(BA_OS)] or if the process dies because it received a signal that it neither caught nor ignored.

# 3 C LIBRARY

# C Library

The C library, libc, contains all of the symbols contained in libsys, and, in addition, contains the routines listed in the following two tables. The first table lists routines from the ANSI C standard.

**Figure 3-1: libc Contents, Names without Synonyms**

abort
abs
asctime
atof
atoi
atol
bsearch
clearerr
clock
ctime
difftime
div
fclose
feof
ferror
fflush
fgetc
fgetpos
fgets
fopen
fprintf
fputc
fputs
fread
freopen
frexp
fscanf
fseek
fsetpos
ftell
fwrite
getc
getchar
getenv
gets
gmtime
isalnum
isalpha
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit
labs
ldexp
ldiv
localtime
longjmp
mblen
mbstowcs
mbtowc
memchr
memcmp
memcpy
memmove
memset
mktime
perror
printf
putc
putchar
puts
qsort
raise
rand
rewind
scanf
setbuf
setjmp
setvbuf
sprintf
srand
sscanf
strcat
strchr
strcmp
strcpy
strcspn
strlen
strncat
strncmp
strncpy
strpbrk
strrchr
strspn
strstr
strtod
strtok
strtol
strtoul
tmpfile
tmpnam
tolower
toupper
ungetc
vfprintf
vprintf
vsprintf
wcstombs
wctomb

Additionally, libc holds the following services.

**Figure 3-2: libc Contents, Names with Synonyms**

_ _assert
cfgetispeed
cfgetospeed
cfsetispeed
cfsetospeed
ctermid
cuserid
dup2
fdopen
_ _filbuf
fileno
_ _flsbuf
fmtmsg †
getdate
getopt
getpass
getsubopt
getw
hcreate
hdestroy
hsearch
isascii
isatty
isnan
isnand †
lfind
lockf †
lsearch
memccpy
mkfifo
mktemp
monitor
nftw
nl_langinfo
pclose
popen
putenv
putw
setlabel
sleep
strdup
swab
tcdrain
tcflow
tcflush
tcgetattr
tcgetpgrp
tcgetsid
tcsendbreak
tcsetattr
tcsetpgrp
tdelete
tell †
tempnam
tfind
toascii
_tolower
tsearch
_toupper
twalk
tzset
_xftw

† Function is at Level 2 in the SVID Issue 3 and therefore at Level 2 in the ABI.

Besides the symbols listed in the With Synonyms table above, synonyms of the form *_name* exist for *name* entries that are not listed with a leading underscore prepended to their name. Thus `libc` contains both `getopt` and `_getopt`, for example.

Of the routines listed above, the following are not defined elsewhere.

`int _ _filbuf(FILE *f);`

This function returns the next input character for `f`, filling its buffer as appropriate. It returns `EOF` if an error occurs.

`int _ _flsbuf(int x, FILE *f);`

This function flushes the output characters for `f` as if `putc(x,f)` had been called and then appends the value of `x` to the resulting output stream. It returns `EOF` if an error occurs and `x` otherwise.

`int _xftw(int, char *, int (*)(char *, struct stat *, int), int);`

Calls to the `ftw`(BA_LIB) function are mapped to this function when applications are compiled. This function is identical to `ftw`(BA_LIB), except that `_xftw()` takes an interposed first argument, which must have the value 2.

See this chapter's other library sections for more SVID, ANSI C, and POSIX facilities. See ‘‘System Data Interfaces’’ later in this chapter for more information.

## Global Data Symbols

The `libc` library requires that some global external data symbols be defined for its routines to work properly. All the data symbols required for the `libsys` library must be provided by `libc`, as well as the data symbols listed in the table below.

For formal declarations of the data objects represented by these symbols, see the *System V Interface Definition, Third Edition* or the ‘‘Data Definitions’’ section of Chapter 6 in the appropriate processor supplement to the *System V ABI*.

For entries in the following table that are in *name* - *_name* form, both symbols in each pair represent the same data. The underscore synonyms are provided to satisfy the ANSI C standard.

**Figure 3-3:** `libc` **Contents, Global External Data Symbols**

```
getdate_err    optarg
_getdate_err   opterr
_ _iob         optind
               optopt
```

I

# Index

# Index

# II

# DWARF Debugging Information Format

This document specifies the second generation of symbolic debugging information based on the DWARF format that has been developed by the UNIX International Programming Language Special Interest Group (SIG). It is being circulated for industry review.

UNIX INTERNATIONAL DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS DOCUMENTATION, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL UNIX INTERNATIONAL BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS DOCUMENTATION.

# TIS Portable Formats Specification, Version 1.1
# DWARF Debugging Information Format

This document describes the DWARF specification, a portable debugging information format. The TIS Committee formed a debug subcommittee to evaluate the widely available formats with the objective of adopting one as the TIS standard. After studying many different formats, the TIS Committee approved DWARF Version 1.0 as the most flexible format for handling the present and future computer language debugging needs. The major advantages of DWARF are:

- it can be easily adopted across numerous 32-bit Intel Architecture environments
- it has been designed from the beginning to be easily extensible
- there is a standardization committee (PLSIG) working on extending it. TIS members can use this committee to extend the format as needed.
- it is a publicly available specification with no licensing requirements. The full report from the debug subcommittee is available from the TIS archives.

There are two versions of DWARF currently available, Version 1.0 and an industry review Version 2.0. Version 1.0 is fully compatible with *sdb*, an existing debugger from USL, and covers C and Fortran completely. There are also some basic extensions for 64-bit architectures. UNIX International has published this version of DWARF. This document was originally designed by USL (UNIX System Laboratories) and turned over to the PLSIG (Programming Languages Special Interest Group) of UNIX International in an effort to extend and standardize the format.

This current version, DWARF Version 2.0 adds significant new functionality, but its main thrust is to achieve a much denser encoding of the DWARF information. Because of the new encoding, DWARF Version 2.0 is not binary compatible with DWARF Version 1.0.

Version 2.0 expands DWARF functionality by addressing the needs of more languages and reducing the size of the DWARF information in the object file. The TIS Committee supports the efforts of the PLSIG by actively reviewing the emerging DWARF Version 2.0 specification. To this end, the following changes are worth noting:

- DASI: Dwarf Augmented Statement Information. This is an enhanced line number information table, and the on-disk version of this is a compressed version. This was proposed by Borland International, which claims a large reduction in the size of DWARF information with DASI. Some experiments have shown a reduction of more than 20% in the DASI.

- Abbreviations: This is a means of pre-defining the TAG-attribute combination in the compilation unit in a table. In some experiments, this has cut the size of DWARF in half.

Industry review Version 2.0 provides a feature for expressing MACRO information. It also provides a new and improved location description method, which is an extremely powerful means for describing variable locations and expressing compiler optimizations. More extensive support for C++ has also been added, and at this point, the PLSIG believes that this document sufficiently supports the debugging needs of C, C++, FORTRAN 77, Fortran90, Modula2 and Pascal.

The industry review draft of the DWARF Version 2.0 specification is presented here. The final Version 2.0 specification may include some minor changes and will be available in 1994.

This document specifies the second generation of symbolic debugging information based on the DWARF format that has been developed by the UNIX International Programming Language Special Interest Group (SIG). It is being circulated for industry review.

NOTICE:

UNIX International is making this documentation available as a reference point for the industry. While UNIX International believes that this specification is well defined in this first release of the document, minor changes may be made prior to products meeting this specification being made available from UNIX System Laboratories or UNIX International members.

Trademarks:

Table of Contents

# List of Figures

# 1. INTRODUCTION

This document defines the format for the information generated by compilers, assemblers and linkage editors that is necessary for symbolic, source-level debugging. The debugging information format does not favor the design of any compiler or debugger. Instead, the goal is to create a method of communicating an accurate picture of the source program to any debugger in a form that is economically extensible to different languages while retaining backward compatibility.

The design of the debugging information format is open-ended, allowing for the addition of new debugging information to accommodate new languages or debugger capabilities while remaining compatible with other languages or different debuggers.

## 1.1 Purpose and Scope

The debugging information format described in this document is designed to meet the symbolic, source-level debugging needs of different languages in a unified fashion by requiring language independent debugging information whenever possible. Individual needs, such as C++ virtual functions or Fortran common blocks are accommodated by creating attributes that are used only for those languages. The UNIX International Programming Languages SIG believes that this document sufficiently covers the debugging information needs of C, C++, FORTRAN77, Fortran90, Modula2 and Pascal.

This document describes DWARF Version 2, the second generation of debugging information based on the DWARF format. While DWARF Version 2 provides new debugging information not available in Version 1, the primary focus of the changes for Version 2 is the representation of the information, rather than the information content itself. The basic structure of the Version 2 format remains as in Version 1: the debugging information is represented as a series of debugging information entries, each containing one or more attributes (name/value pairs). The Version 2 representation, however, is much more compact than the Version 1 representation. In some cases, this greater density has been achieved at the expense of additional complexity or greater difficulty in producing and processing the DWARF information. We believe that the reduction in I/O and in memory paging should more than make up for any increase in processing time.

Because the representation of information has changed from Version 1 to Version 2, Version 2 DWARF information is not binary compatible with Version 1 information. To make it easier for consumers to support both Version 1 and Version 2 DWARF information, the Version 2 information has been moved to a different object file section, `.debug_info`.

The intended audience for this document are the developers of both producers and consumers of debugging information, typically language compilers, debuggers and other tools that need to interpret a binary program in terms of its original source.

## 1.2 Overview

There are two major pieces to the description of the DWARF format in this document. The first piece is the informational content of the debugging entries. The second piece is the way the debugging information is encoded and represented in an object file.

The informational content is described in sections two through six. Section two describes the overall structure of the information and attributes that are common to many or all of the different debugging information entries. Sections three, four and five describe the specific debugging information entries and how they communicate the necessary information about the source program to a debugger. Section six describes debugging information contained outside of the debugging information entries, themselves. The encoding of the DWARF information is presented in section seven.

Section eight describes some future directions for the DWARF specification.

In the following sections, text in normal font describes required aspects of the DWARF format. Text in *italics* is explanatory or supplementary material, and not part of the format definition itself.

## 1.3 Vendor Extensibility

This document does not attempt to cover all interesting languages or even to cover all of the interesting debugging information needs for its primary target languages (C, C++, FORTRAN77, Fortran90, Modula2, Pascal). Therefore the document provides vendors a way to define their own debugging information tags, attributes, base type encodings, location operations, language names, calling conventions and call frame instructions by reserving a portion of the name space and valid values for these constructs for vendor specific additions. Future versions of this document will not use names or values reserved for vendor specific additions. All names and values not reserved for vendor additions, however, are reserved for future versions of this document. See section 7 for details.

## 1.4 Changes from Version 1

The following is a list of the major changes made to the DWARF Debugging Information Format since Version 1 of the format was published (January 20, 1992). The list is not meant to be exhaustive.

- Debugging information entries have been moved from the `.debug` to the `.debug_info` section of an object file.
- The tag, attribute names and attribute forms encodings have been moved out of the debugging information itself to a separate abbreviations table.
- Explicit sibling pointers have been made optional. Each entry now specifies (through the abbreviations table) whether or not it has children.
- New more compact attribute forms have been added, including a variable length constant data form. Attribute values may now have any form within a given class of forms.
- Location descriptions have been replaced by a new, more compact and more expressive format. There is now a way of expressing multiple locations for an object whose location changes during its lifetime.
- There is a new format for line number information that provides information for code contributed to a compilation unit from an included file. Line number information is now in the `.debug_line` section of an object file.
- The representation of the type of a declaration has been reworked.
- A new section provides an encoding for pre-processor macro information.
- Debugging information entries now provide for the representation of non-defining declarations of objects, functions or types.
- More complete support for Modula2 and Pascal has been added.

- There is now a way of describing locations for segmented address spaces.
- A new section provides an encoding for information about call frame activations.
- The representation of enumeration and array types has been reworked so that DWARF presents only a single way of representing lists of items.
- Support has been added for C++ templates and exceptions.

# 2. GENERAL DESCRIPTION

## 2.1 The Debugging Information Entry

DWARF uses a series of debugging information entries to define a low-level representation of a source program. Each debugging information entry is described by an identifying tag and contains a series of attributes. The tag specifies the class to which an entry belongs, and the attributes define the specific characteristics of the entry.

The set of required tag names is listed in Figure 1. The debugging information entries they identify are described in sections three, four and five.

The debugging information entries in DWARF Version 2 are intended to exist in the `.debug_info` section of an object file.

```
DW_TAG_access_declaration     DW_TAG_array_type
DW_TAG_base_type              DW_TAG_catch_block
DW_TAG_class_type             DW_TAG_common_block
DW_TAG_common_inclusion       DW_TAG_compile_unit
DW_TAG_const_type             DW_TAG_constant
DW_TAG_entry_point            DW_TAG_enumeration_type
DW_TAG_enumerator             DW_TAG_file_type
DW_TAG_formal_parameter       DW_TAG_friend
DW_TAG_imported_declaration   DW_TAG_inheritance
DW_TAG_inlined_subroutine     DW_TAG_label
DW_TAG_lexical_block          DW_TAG_member
DW_TAG_module                 DW_TAG_namelist
DW_TAG_namelist_item          DW_TAG_packed_type
DW_TAG_pointer_type           DW_TAG_ptr_to_member_type
DW_TAG_reference_type         DW_TAG_set_type
DW_TAG_string_type            DW_TAG_structure_type
DW_TAG_subprogram             DW_TAG_subrange_type
DW_TAG_subroutine_type        DW_TAG_template_type_param
DW_TAG_template_value_param   DW_TAG_thrown_type
DW_TAG_try_block              DW_TAG_typedef
DW_TAG_union_type             DW_TAG_unspecified_parameters
DW_TAG_variable               DW_TAG_variant
DW_TAG_variant_part           DW_TAG_volatile_type
DW_TAG_with_stmt
```

**Figure 1.** Tag names

## 2.2 Attribute Types

Each attribute value is characterized by an attribute name. The set of attribute names is listed in Figure 2.

The permissible values for an attribute belong to one or more classes of attribute value forms. Each form class may be represented in one or more ways. For instance, some attribute values consist of a single piece of constant data. ‘‘Constant data’’ is the class of attribute value that those attributes may have. There are several representations of constant data, however (one, two, four, eight bytes and variable length data). The particular representation for any given instance of an attribute is encoded along with the attribute name as part of the information that guides the

```
DW_AT_abstract_origin       DW_AT_accessibility
DW_AT_address_class         DW_AT_artificial
DW_AT_base_types            DW_AT_bit_offset
DW_AT_bit_size              DW_AT_byte_size
DW_AT_calling_convention    DW_AT_common_reference
DW_AT_comp_dir              DW_AT_const_value
DW_AT_containing_type       DW_AT_count
DW_AT_data_member_location  DW_AT_decl_column
DW_AT_decl_file             DW_AT_decl_line
DW_AT_declaration           DW_AT_default_value
DW_AT_discr                 DW_AT_discr_list
DW_AT_discr_value           DW_AT_encoding
DW_AT_external              DW_AT_frame_base
DW_AT_friend                DW_AT_high_pc
DW_AT_identifier_case       DW_AT_import
DW_AT_inline                DW_AT_is_optional
DW_AT_language              DW_AT_location
DW_AT_low_pc                DW_AT_lower_bound
DW_AT_macro_info            DW_AT_name
DW_AT_namelist_item         DW_AT_ordering
DW_AT_priority              DW_AT_producer
DW_AT_prototyped            DW_AT_return_addr
DW_AT_segment               DW_AT_sibling
DW_AT_specification         DW_AT_start_scope
DW_AT_static_link           DW_AT_stmt_list
DW_AT_stride_size           DW_AT_string_length
DW_AT_type                  DW_AT_upper_bound
DW_AT_use_location          DW_AT_variable_parameter
DW_AT_virtuality            DW_AT_visibility
DW_AT_vtable_elem_location
```

**Figure 2.** Attribute names

interpretation of a debugging information entry. Attribute value forms may belong to one of the following classes.

address — Refers to some location in the address space of the described program.

block — An arbitrary number of uninterpreted bytes of data.

constant — One, two, four or eight bytes of uninterpreted data, or data encoded in the variable length format known as LEB128 (see section 7.6).

flag — A small constant that indicates the presence or absence of an attribute.

reference — Refers to some member of the set of debugging information entries that describe the program. There are two types of reference. The first is an offset relative to the beginning of the compilation unit in which the reference occurs and must refer to an entry within that same compilation unit. The second type of reference is the address of any debugging information entry within the same executable or shared object; it may refer to an entry in a different compilation unit from the unit containing the reference.

string — A null-terminated sequence of zero or more (non-null) bytes. Data in this form are generally printable strings. Strings may be represented directly in the debugging information entry or as an offset in a separate string table.

There are no limitations on the ordering of attributes within a debugging information entry, but to prevent ambiguity, no more than one attribute with a given name may appear in any debugging information entry.

## 2.3 Relationship of Debugging Information Entries

*A variety of needs can be met by permitting a single debugging information entry to ‘‘own’’ an arbitrary number of other debugging entries and by permitting the same debugging information entry to be one of many owned by another debugging information entry. This makes it possible to describe, for example, the static block structure within a source file, show the members of a structure, union, or class, and associate declarations with source files or source files with shared objects.*

The ownership relation of debugging information entries is achieved naturally because the debugging information is represented as a tree. The nodes of the tree are the debugging information entries themselves. The child entries of any node are exactly those debugging information entries owned by that node.[1]

The tree itself is represented by flattening it in prefix order. Each debugging information entry is defined either to have child entries or not to have child entries (see section 7.5.3). If an entry is defined not to have children, the next physically succeeding entry is the sibling of the prior entry. If an entry is defined to have children, the next physically succeeding entry is the first child of the prior entry. Additional children of the parent entry are represented as siblings of the first child. A chain of sibling entries is terminated by a null entry.

In cases where a producer of debugging information feels that it will be important for consumers of that information to quickly scan chains of sibling entries, ignoring the children of individual siblings, that producer may attach an `AT_sibling` attribute to any debugging information entry. The value of this attribute is a reference to the sibling entry of the entry to which the attribute is attached.

## 2.4 Location Descriptions

*The debugging information must provide consumers a way to find the location of program variables, determine the bounds of dynamic arrays and strings and possibly to find the base address of a subroutine’s stack frame or the return address of a subroutine. Furthermore, to meet the needs of recent computer architectures and optimization techniques, the debugging information must be able to describe the location of an object whose location changes over the object’s lifetime.*

[1] While the ownership relation of the debugging information entries is represented as a tree, other relations among the entries exist, for example, a pointer from an entry representing a variable to another entry representing the type of that variable. If all such relations are taken into account, the debugging entries form a graph, not a tree.

Information about the location of program objects is provided by location descriptions. Location descriptions can be either of two forms:

1. *Location expressions* which are a language independent representation of addressing rules of arbitrary complexity built from a few basic building blocks, or *operations*. They are sufficient for describing the location of any object as long as its lifetime is either static or the same as the lexical block that owns it, and it does not move throughout its lifetime.
2. *Location lists* which are used to describe objects that have a limited lifetime or change their location throughout their lifetime. Location lists are more completely described below.

The two forms are distinguished in a context sensitive manner. As the value of an attribute, a location expression is encoded as a block and a location list is encoded as a constant offset into a location list table.

*Note: The Version 1 concept of "location descriptions" was replaced in Version 2 with this new abstraction because it is denser and more descriptive.*

### 2.4.1 Location Expressions

A location expression consists of zero or more location operations. An expression with zero operations is used to denote an object that is present in the source code but not present in the object code (perhaps because of optimization). The location operations fall into two categories, register names and addressing operations. Register names always appear alone and indicate that the referred object is contained inside a particular register. Addressing operations are memory address computation rules. All location operations are encoded as a stream of opcodes that are each followed by zero or more literal operands. The number of operands is determined by the opcode.

### 2.4.2 Register Name Operators

The following operations can be used to name a register.

*Note that the register number represents a DWARF specific mapping of numbers onto the actual registers of a given architecture. The mapping should be chosen to gain optimal density and should be shared by all users of a given architecture. The Programming Languages SIG recommends that this mapping be defined by the ABI*[2] *authoring committee for each architecture.*

1. `DW_OP_reg0`, `DW_OP_reg1`, ..., `DW_OP_reg31`
   The `DW_OP_reg`*n* operations encode the names of up to 32 registers, numbered from 0 through 31, inclusive. The object addressed is in register *n*.
2. `DW_OP_regx`
   The `DW_OP_regx` operation has a single unsigned LEB128 literal operand that encodes the name of a register.

[2] *System V Application Binary Interface*, consisting of the generic interface and processor supplements for each target architecture.

### 2.4.3 Addressing Operations

Each addressing operation represents a postfix operation on a simple stack machine. Each element of the stack is the size of an address on the target machine. The value on the top of the stack after ‘‘executing’’ the location expression is taken to be the result (the address of the object, or the value of the array bound, or the length of a dynamic string). In the case of locations used for structure members, the computation assumes that the base address of the containing structure has been pushed on the stack before evaluation of the addressing operation.

#### 2.4.3.1 Literal Encodings

The following operations all push a value onto the addressing stack.

1. `DW_OP_lit0`, `DW_OP_lit1`, ..., `DW_OP_lit31`
   The `DW_OP_lit`*n* operations encode the unsigned literal values from 0 through 31, inclusive.

2. `DW_OP_addr`
   The `DW_OP_addr` operation has a single operand that encodes a machine address and whose size is the size of an address on the target machine.

3. `DW_OP_const1u`
   The single operand of the `DW_OP_const1u` operation provides a 1-byte unsigned integer constant.

4. `DW_OP_const1s`
   The single operand of the `DW_OP_const1s` operation provides a 1-byte signed integer constant.

5. `DW_OP_const2u`
   The single operand of the `DW_OP_const2u` operation provides a 2-byte unsigned integer constant.

6. `DW_OP_const2s`
   The single operand of the `DW_OP_const2s` operation provides a 2-byte signed integer constant.

7. `DW_OP_const4u`
   The single operand of the `DW_OP_const4u` operation provides a 4-byte unsigned integer constant.

8. `DW_OP_const4s`
   The single operand of the `DW_OP_const4s` operation provides a 4-byte signed integer constant.

9. `DW_OP_const8u`
   The single operand of the `DW_OP_const8u` operation provides an 8-byte unsigned integer constant.

10. `DW_OP_const8s`
    The single operand of the `DW_OP_const8s` operation provides an 8-byte signed integer constant.

11. `DW_OP_constu`
    The single operand of the `DW_OP_constu` operation provides an unsigned LEB128 integer constant.

12. `DW_OP_consts`
    The single operand of the `DW_OP_consts` operation provides a signed LEB128 integer constant.

#### 2.4.3.2 Register Based Addressing

The following operations push a value onto the stack that is the result of adding the contents of a register with a given signed offset.

1. `DW_OP_fbreg`
   The `DW_OP_fbreg` operation provides a signed LEB128 offset from the address specified by the location descriptor in the `DW_AT_frame_base` attribute of the current function. *(This is typically a "stack pointer" register plus or minus some offset. On more sophisticated systems it might be a location list that adjusts the offset according to changes in the stack pointer as the PC changes.)*

2. `DW_OP_breg0`, `DW_OP_breg1`, ..., `DW_OP_breg31`
   The single operand of the `DW_OP_breg`*n* operations provides a signed LEB128 offset from the specified register.

3. `DW_OP_bregx`
   The `DW_OP_bregx` operation has two operands: a signed LEB128 offset from the specified register which is defined with an unsigned LEB128 number.

#### 2.4.3.3 Stack Operations

The following operations manipulate the ‘‘location stack.’’ Location operations that index the location stack assume that the top of the stack (most recently added entry) has index 0.

1. `DW_OP_dup`
   The `DW_OP_dup` operation duplicates the value at the top of the location stack.

2. `DW_OP_drop`
   The `DW_OP_drop` operation pops the value at the top of the stack.

3. `DW_OP_pick`
   The single operand of the `DW_OP_pick` operation provides a 1-byte index. The stack entry with the specified index (0 through 255, inclusive) is pushed on the stack.

4. `DW_OP_over`
   The `DW_OP_over` operation duplicates the entry currently second in the stack at the top of the stack. This is equivalent to an `DW_OP_pick` operation, with index 1.

5. `DW_OP_swap`
   The `DW_OP_swap` operation swaps the top two stack entries. The entry at the top of the stack becomes the second stack entry, and the second entry becomes the top of the stack.

6. `DW_OP_rot`
   The `DW_OP_rot` operation rotates the first three stack entries. The entry at the top of the stack becomes the third stack entry, the second entry becomes the top of the stack, and the third entry becomes the second entry.

7. `DW_OP_deref`
   The `DW_OP_deref` operation pops the top stack entry and treats it as an address. The value retrieved from that address is pushed. The size of the data retrieved from the dereferenced address is the size of an address on the target machine.

8. DW_OP_deref_size
   The DW_OP_deref_size operation behaves like the DW_OP_deref operation: it pops the top stack entry and treats it as an address. The value retrieved from that address is pushed. In the DW_OP_deref_size operation, however, the size in bytes of the data retrieved from the dereferenced address is specified by the single operand. This operand is a 1-byte unsigned integral constant whose value may not be larger than the size of an address on the target machine. The data retrieved is zero extended to the size of an address on the target machine before being pushed on the expression stack.

9. DW_OP_xderef
   The DW_OP_xderef operation provides an extended dereference mechanism. The entry at the top of the stack is treated as an address. The second stack entry is treated as an ‘‘address space identifier’’ for those architectures that support multiple address spaces. The top two stack elements are popped, a data item is retrieved through an implementation-defined address calculation and pushed as the new stack top. The size of the data retrieved from the dereferenced address is the size of an address on the target machine.

10. DW_OP_xderef_size
    The DW_OP_xderef_size operation behaves like the DW_OP_xderef operation: the entry at the top of the stack is treated as an address. The second stack entry is treated as an ‘‘address space identifier’’ for those architectures that support multiple address spaces. The top two stack elements are popped, a data item is retrieved through an implementation-defined address calculation and pushed as the new stack top. In the DW_OP_xderef_size operation, however, the size in bytes of the data retrieved from the dereferenced address is specified by the single operand. This operand is a 1-byte unsigned integral constant whose value may not be larger than the size of an address on the target machine. The data retrieved is zero extended to the size of an address on the target machine before being pushed on the expression stack.

#### 2.4.3.4 Arithmetic and Logical Operations

The following provide arithmetic and logical operations. The arithmetic operations perform ‘‘addressing arithmetic,’’ that is, unsigned arithmetic that wraps on an address-sized boundary. The operations do not cause an exception on overflow.

1. DW_OP_abs
   The DW_OP_abs operation pops the top stack entry and pushes its absolute value.

2. DW_OP_and
   The DW_OP_and operation pops the top two stack values, performs a bitwise *and* operation on the two, and pushes the result.

3. DW_OP_div
   The DW_OP_div operation pops the top two stack values, divides the former second entry by the former top of the stack using signed division, and pushes the result.

4. DW_OP_minus
   The DW_OP_minus operation pops the top two stack values, subtracts the former top of the stack from the former second entry, and pushes the result.

5. DW_OP_mod
   The DW_OP_mod operation pops the top two stack values and pushes the result of the calculation: former second stack entry modulo the former top of the stack.

6. DW_OP_mul
   The DW_OP_mul operation pops the top two stack entries, multiplies them together, and pushes the result.

7. DW_OP_neg
   The DW_OP_neg operation pops the top stack entry, and pushes its negation.

8. DW_OP_not
   The DW_OP_not operation pops the top stack entry, and pushes its bitwise complement.

9. DW_OP_or
   The DW_OP_or operation pops the top two stack entries, performs a bitwise *or* operation on the two, and pushes the result.

10. DW_OP_plus
    The DW_OP_plus operation pops the top two stack entries, adds them together, and pushes the result.

11. DW_OP_plus_uconst
    The DW_OP_plus_uconst operation pops the top stack entry, adds it to the unsigned LEB128 constant operand and pushes the result. *This operation is supplied specifically to be able to encode more field offsets in two bytes than can be done with "*DW_OP_lit*n* DW_OP_add*".*

12. DW_OP_shl
    The DW_OP_shl operation pops the top two stack entries, shifts the former second entry left by the number of bits specified by the former top of the stack, and pushes the result.

13. DW_OP_shr
    The DW_OP_shr operation pops the top two stack entries, shifts the former second entry right (logically) by the number of bits specified by the former top of the stack, and pushes the result.

14. DW_OP_shra
    The DW_OP_shra operation pops the top two stack entries, shifts the former second entry right (arithmetically) by the number of bits specified by the former top of the stack, and pushes the result.

15. DW_OP_xor
    The DW_OP_xor operation pops the top two stack entries, performs the logical *exclusive-or* operation on the two, and pushes the result.

#### 2.4.3.5 Control Flow Operations

The following operations provide simple control of the flow of a location expression.

1. Relational operators
   The six relational operators each pops the top two stack values, compares the former top of the stack with the former second entry, and pushes the constant value 1 onto the stack if the result of the operation is true or the constant value 0 if the result of the operation is false. The comparisons are done as signed operations. The six operators are DW_OP_le (less than or equal to), DW_OP_ge (greater than or equal to), DW_OP_eq (equal to), DW_OP_lt (less than), DW_OP_gt (greater than) and DW_OP_ne (not equal to).

2. `DW_OP_skip`
   `DW_OP_skip` is an unconditional branch. Its single operand is a 2-byte signed integer constant. The 2-byte constant is the number of bytes of the location expression to skip from the current operation, beginning after the 2-byte constant.

3. `DW_OP_bra`
   `DW_OP_bra` is a conditional branch. Its single operand is a 2-byte signed integer constant. This operation pops the top of stack. If the value popped is not the constant 0, the 2-byte constant operand is the number of bytes of the location expression to skip from the current operation, beginning after the 2-byte constant.

#### 2.4.3.6 Special Operations

There are two special operations currently defined:

1. `DW_OP_piece`
   *Many compilers store a single variable in sets of registers, or store a variable partially in memory and partially in registers.* `DW_OP_piece` *provides a way of describing how large a part of a variable a particular addressing expression refers to.*

   `DW_OP_piece` takes a single argument which is an unsigned LEB128 number. The number describes the size in bytes of the piece of the object referenced by the addressing expression whose result is at the top of the stack.

2. `DW_OP_nop`
   The `DW_OP_nop` operation is a place holder. It has no effect on the location stack or any of its values.

### 2.4.4 Sample Stack Operations

*The stack operations defined in section 2.4.3.3 are fairly conventional, but the following examples illustrate their behavior graphically.*

| Before | | Operation | After | |
|---|---|---|---|---|
| 0 | 17 | DW_OP_dup | 0 | 17 |
| 1 | 29 | | 1 | 17 |
| 2 | 1000 | | 2 | 29 |
| | | | 3 | 1000 |
| 0 | 17 | DW_OP_drop | 0 | 29 |
| 1 | 29 | | 1 | 1000 |
| 2 | 1000 | | | |
| 0 | 17 | DW_OP_pick 2 | 0 | 1000 |
| 1 | 29 | | 1 | 17 |
| 2 | 1000 | | 2 | 29 |
| | | | 3 | 1000 |
| 0 | 17 | DW_OP_over | 0 | 29 |
| 1 | 29 | | 1 | 17 |
| 2 | 1000 | | 2 | 29 |
| | | | 3 | 1000 |
| 0 | 17 | DW_OP_swap | 0 | 29 |
| 1 | 29 | | 1 | 17 |
| 2 | 1000 | | 2 | 1000 |
| 0 | 17 | DW_OP_rot | 0 | 29 |
| 1 | 29 | | 1 | 1000 |
| 2 | 1000 | | 2 | 17 |

### 2.4.5 Example Location Expressions

*The addressing expression represented by a location expression, if evaluated, generates the runtime address of the value of a symbol except where the* `DW_OP_reg`*n, or* `DW_OP_regx` *operations are used.*

*Here are some examples of how location operations are used to form location expressions:*

`DW_OP_reg3`

*The value is in register 3.*

`DW_OP_regx 54`

*The value is in register 54.*

`DW_OP_addr 0x80d0045c`

*The value of a static variable is*
*at machine address 0x80d0045c.*

`DW_OP_breg11 44`

*Add 44 to the value in*
*register 11 to get the address of an*
*automatic variable instance.*

`DW_OP_fbreg -50`

*Given an* `DW_AT_frame_base` *value of*
*"*`OPBREG31 64`*," this example*
*computes the address of a local variable*
*that is -50 bytes from a logical frame*
*pointer that is computed by adding*
*64 to the current stack pointer (register 31).*

`DW_OP_bregx 54 32 DW_OP_deref`

*A call-by-reference parameter*
*whose address is in the*
*word 32 bytes from where register*
*54 points.*

`DW_OP_plus_uconst 4`

*A structure member is four bytes*
*from the start of the structure*
*instance. The base address is*
*assumed to be already on the stack.*

`DW_OP_reg3 DW_OP_piece 4 DW_OP_reg10 DW_OP_piece 2`

*A variable whose first four bytes reside*
*in register 3 and whose next two bytes*
*reside in register 10.*

### 2.4.6 Location Lists

Location lists are used in place of location expressions whenever the object whose location is being described can change location during its lifetime. Location lists are contained in a separate object file section called `.debug_loc`. A location list is indicated by a location attribute whose value is represented as a constant offset from the beginning of the `.debug_loc` section to the first byte of the list for the object in question.

Each entry in a location list consists of:

1. A beginning address. This address is relative to the base address of the compilation unit referencing this location list. It marks the beginning of the address range over which the location is valid.
2. An ending address, again relative to the base address of the compilation unit referencing this location list. It marks the first address past the end of the address range over which the location is valid.
3. A location expression describing the location of the object over the range specified by the beginning and end addresses.

Address ranges may overlap. When they do, they describe a situation in which an object exists simultaneously in more than one place. If all of the address ranges in a given location list do not collectively cover the entire range over which the object in question is defined, it is assumed that the object is not available for the portion of the range that is not covered.

The end of any given location list is marked by a 0 for the beginning address and a 0 for the end address; no location description is present. A location list containing only such a 0 entry describes an object that exists in the source code but not in the executable program.

## 2.5 Types of Declarations

Any debugging information entry describing a declaration that has a type has a `DW_AT_type` attribute, whose value is a reference to another debugging information entry. The entry referenced may describe a base type, that is, a type that is not defined in terms of other data types, or it may describe a user-defined type, such as an array, structure or enumeration. Alternatively, the entry referenced may describe a type modifier: constant, packed, pointer, reference or volatile, which in turn will reference another entry describing a type or type modifier (using a `DW_AT_type` attribute of its own). See section 5 for descriptions of the entries describing base types, user-defined types and type modifiers.

## 2.6 Accessibility of Declarations

*Some languages, notably C++ and Ada, have the concept of the accessibility of an object or of some other program entity. The accessibility specifies which classes of other program objects are permitted access to the object in question.*

The accessibility of a declaration is represented by a `DW_AT_accessibility` attribute, whose value is a constant drawn from the set of codes listed in Figure 3.

```
DW_ACCESS_public
DW_ACCESS_private
DW_ACCESS_protected
```

**Figure 3.** Accessibility codes

## 2.7 Visibility of Declarations

*Modula2 has the concept of the visibility of a declaration. The visibility specifies which declarations are to be visible outside of the module in which they are declared.*

The visibility of a declaration is represented by a `DW_AT_visibility` attribute, whose value is a constant drawn from the set of codes listed in Figure 4.

```
DW_VIS_local
DW_VIS_exported
DW_VIS_qualified
```

**Figure 4.** Visibility codes

## 2.8 Virtuality of Declarations

*C++ provides for virtual and pure virtual structure or class member functions and for virtual base classes.*

The virtuality of a declaration is represented by a `DW_AT_virtuality` attribute, whose value is a constant drawn from the set of codes listed in Figure 5.

```
DW_VIRTUALITY_none
DW_VIRTUALITY_virtual
DW_VIRTUALITY_pure_virtual
```

**Figure 5.** Virtuality codes

## 2.9 Artificial Entries

*A compiler may wish to generate debugging information entries for objects or types that were not actually declared in the source of the application. An example is a formal parameter entry to represent the hidden* `this` *parameter that most C++ implementations pass as the first argument to non-static member functions.*

Any debugging information entry representing the declaration of an object or type artificially generated by a compiler and not explicitly declared by the source program may have a `DW_AT_artificial` attribute. The value of this attribute is a flag.

## 2.10 Target-Specific Addressing Information

*In some systems, addresses are specified as offsets within a given segment rather than as locations within a single flat address space.*

Any debugging information entry that contains a description of the location of an object or subroutine may have a `DW_AT_segment` attribute, whose value is a location description. The description evaluates to the segment value of the item being described. If the entry containing the `DW_AT_segment` attribute has a `DW_AT_low_pc` or `DW_AT_high_pc` attribute, or a location description that evaluates to an address, then those values represent the offset portion of the address within the segment specified by `DW_AT_segment`.

If an entry has no `DW_AT_segment` attribute, it inherits the segment value from its parent entry. If none of the entries in the chain of parents for this entry back to its containing compilation unit entry have `DW_AT_segment` attributes, then the entry is assumed to exist within a flat address space. Similarly, if the entry has a `DW_AT_segment` attribute containing an empty location description, that entry is assumed to exist within a flat address space.

*Some systems support different classes of addresses. The address class may affect the way a pointer is dereferenced or the way a subroutine is called.*

Any debugging information entry representing a pointer or reference type or a subroutine or subroutine type may have a `DW_AT_address_class` attribute, whose value is a constant. The set of permissible values is specific to each target architecture. The value `DW_ADDR_none`, however, is common to all encodings, and means that no address class has been specified.

*For example, the Intel386™ processor might use the following values:*

| Name | Value | Meaning |
|---|---|---|
| `DW_ADDR_none` | `0` | no class specified |
| `DW_ADDR_near16` | `1` | 16-bit offset, no segment |
| `DW_ADDR_far16` | `2` | 16-bit offset, 16-bit segment |
| `DW_ADDR_huge16` | `3` | 16-bit offset, 16-bit segment |
| `DW_ADDR_near32` | `4` | 32-bit offset, no segment |
| `DW_ADDR_far32` | `5` | 32-bit offset, 16-bit segment |

**Figure 6.** Example address class codes

## 2.11 Non-Defining Declarations

A debugging information entry representing a program object or type typically represents the defining declaration of that object or type. In certain contexts, however, a debugger might need information about a declaration of a subroutine, object or type that is not also a definition to evaluate an expression correctly.

*As an example, consider the following fragment of C code:*

```
void myfunc()
{
        int     x;
        {
                extern float x;
                g(x);
        }
}
```

*ANSI-C scoping rules require that the value of the variable* `x` *passed to the function* `g` *is the value of the global variable* `x` *rather than of the local version.*

Debugging information entries that represent non-defining declarations of a program object or type have a `DW_AT_declaration` attribute, whose value is a flag.

## 2.12 Declaration Coordinates

*It is sometimes useful in a debugger to be able to associate a declaration with its occurrence in the program source.*

Any debugging information entry representing the declaration of an object, module, subprogram or type may have `DW_AT_decl_file`, `DW_AT_decl_line` and `DW_AT_decl_column` attributes, each of whose value is a constant.

The value of the `DW_AT_decl_file` attribute corresponds to a file number from the statement information table for the compilation unit containing this debugging information entry and represents the source file in which the declaration appeared (see section 6.2). The value 0 indicates that no source file has been specified.

The value of the `DW_AT_decl_line` attribute represents the source line number at which the first character of the identifier of the declared object appears. The value 0 indicates that no source line has been specified.

The value of the `DW_AT_decl_column` attribute represents the source column number at which the first character of the identifier of the declared object appears. The value 0 indicates that no column has been specified.

## 2.13 Identifier Names

Any debugging information entry representing a program entity that has been given a name may have a `DW_AT_name` attribute, whose value is a string representing the name as it appears in the source program. A debugging information entry containing no name attribute, or containing a name attribute whose value consists of a name containing a single null byte, represents a program entity for which no name was given in the source.

*Note that since the names of program objects described by DWARF are the names as they appear in the source program, implementations of language translators that use some form of mangled name (as do many implementations of C++) should use the unmangled form of the name in the DWARF* `DW_AT_name` *attribute, including the keyword* `operator`*, if present. Sequences of multiple whitespace characters may be compressed.*

# 3. PROGRAM SCOPE ENTRIES

This section describes debugging information entries that relate to different levels of program scope: compilation unit, module, subprogram, and so on. These entries may be thought of as bounded by ranges of text addresses within the program.

## 3.1 Compilation Unit Entries

An object file may be derived from one or more compilation units. Each such compilation unit will be described by a debugging information entry with the tag `DW_TAG_compile_unit`.

*A compilation unit typically represents the text and data contributed to an executable by a single relocatable object file. It may be derived from several source files, including pre-processed ''include files.''*

The compilation unit entry may have the following attributes:

1. A `DW_AT_low_pc` attribute whose value is the relocated address of the first machine instruction generated for that compilation unit.

2. A `DW_AT_high_pc` attribute whose value is the relocated address of the first location past the last machine instruction generated for that compilation unit.

   *The address may be beyond the last valid instruction in the executable, of course, for this and other similar attributes.*

   The presence of low and high pc attributes in a compilation unit entry imply that the code generated for that compilation unit is contiguous and exists totally within the boundaries specified by those two attributes. If that is not the case, no low and high pc attributes should be produced.

3. A `DW_AT_name` attribute whose value is a null-terminated string containing the full or relative path name of the primary source file from which the compilation unit was derived.

4. A `DW_AT_language` attribute whose constant value is a code indicating the source language of the compilation unit. The set of language names and their meanings are given in Figure 7.

| | |
|---|---|
| `DW_LANG_C` | Non-ANSI C, such as K&R |
| `DW_LANG_C89` | ISO/ANSI C |
| `DW_LANG_C_plus_plus` | C++ |
| `DW_LANG_Fortran77` | FORTRAN77 |
| `DW_LANG_Fortran90` | Fortran90 |
| `DW_LANG_Modula2` | Modula2 |
| `DW_LANG_Pascal83` | ISO/ANSI Pascal |

**Figure 7.** Language names

5. A `DW_AT_stmt_list` attribute whose value is a reference to line number information for this compilation unit.

   This information is placed in a separate object file section from the debugging information entries themselves. The value of the statement list attribute is the offset in the `.debug_line` section of the first byte of the line number information for this compilation unit. See section 6.2.

6. A `DW_AT_macro_info` attribute whose value is a reference to the macro information for this compilation unit.

   This information is placed in a separate object file section from the debugging information entries themselves. The value of the macro information attribute is the offset in the `.debug_macinfo` section of the first byte of the macro information for this compilation unit. See section 6.3.

7. A `DW_AT_comp_dir` attribute whose value is a null-terminated string containing the current working directory of the compilation command that produced this compilation unit in whatever form makes sense for the host system.

   *The suggested form for the value of the* `DW_AT_comp_dir` *attribute on UNIX systems is ‘‘hostname*`:`*pathname’’. If no hostname is available, the suggested form is ‘‘*`:`*pathname’’.*

8. A `DW_AT_producer` attribute whose value is a null-terminated string containing information about the compiler that produced the compilation unit. The actual contents of the string will be specific to each producer, but should begin with the name of the compiler vendor or some other identifying character sequence that should avoid confusion with other producer values.

9. A `DW_AT_identifier_case` attribute whose constant value is a code describing the treatment of identifiers within this compilation unit. The set of identifier case codes is given in Figure 8.

```
DW_ID_case_sensitive
DW_ID_up_case
DW_ID_down_case
DW_ID_case_insensitive
```

**Figure 8.** Identifier case codes

   `DW_ID_case_sensitive` is the default for all compilation units that do not have this attribute. It indicates that names given as the values of `DW_AT_name` attributes in debugging information entries for the compilation unit reflect the names as they appear in the source program. The debugger should be sensitive to the case of identifier names when doing identifier lookups.

   `DW_ID_up_case` means that the producer of the debugging information for this compilation unit converted all source names to upper case. The values of the name attributes may not reflect the names as they appear in the source program. The debugger should convert all names to upper case when doing lookups.

   `DW_ID_down_case` means that the producer of the debugging information for this compilation unit converted all source names to lower case. The values of the name attributes may not reflect the names as they appear in the source program. The debugger should convert all names to lower case when doing lookups.

   `DW_ID_case_insensitive` means that the values of the name attributes reflect the names as they appear in the source program but that a case insensitive lookup should be used to access those names.

10. A `DW_AT_base_types` attribute whose value is a reference. This attribute points to a debugging information entry representing another compilation unit. It may be used to specify the compilation unit containing the base type entries used by entries in the current

compilation unit (see section 5.1).

*This attribute provides a consumer a way to find the definition of base types for a compilation unit that does not itself contain such definitions. This allows a consumer, for example, to interpret a type conversion to a base type correctly.*

A compilation unit entry owns debugging information entries that represent the declarations made in the corresponding compilation unit.

## 3.2 Module Entries

*Several languages have the concept of a ‘‘module.’’*

A module is represented by a debugging information entry with the tag `DW_TAG_module`. Module entries may own other debugging information entries describing program entities whose declaration scopes end at the end of the module itself.

If the module has a name, the module entry has a `DW_AT_name` attribute whose value is a null-terminated string containing the module name as it appears in the source program.

If the module contains initialization code, the module entry has a `DW_AT_low_pc` attribute whose value is the relocated address of the first machine instruction generated for that initialization code. It also has a `DW_AT_high_pc` attribute whose value is the relocated address of the first location past the last machine instruction generated for the initialization code.

If the module has been assigned a priority, it may have a `DW_AT_priority` attribute. The value of this attribute is a reference to another debugging information entry describing a variable with a constant value. The value of this variable is the actual constant value of the module’s priority, represented as it would be on the target architecture.

*A Modula2 definition module may be represented by a module entry containing a* `DW_AT_declaration` *attribute.*

## 3.3 Subroutine and Entry Point Entries

The following tags exist to describe debugging information entries for subroutines and entry points:

| | |
|---|---|
| `DW_TAG_subprogram` | A global or file static subroutine or function. |
| `DW_TAG_inlined_subroutine` | A particular inlined instance of a subroutine or function. |
| `DW_TAG_entry_point` | A Fortran entry point. |

### 3.3.1 General Subroutine and Entry Point Information

The subroutine or entry point entry has a `DW_AT_name` attribute whose value is a null-terminated string containing the subroutine or entry point name as it appears in the source program.

If the name of the subroutine described by an entry with the tag `DW_TAG_subprogram` is visible outside of its containing compilation unit, that entry has a `DW_AT_external` attribute, whose value is a flag.

*Additional attributes for functions that are members of a class or structure are described in section 5.5.5.*

*A common debugger feature is to allow the debugger user to call a subroutine within the subject program. In certain cases, however, the generated code for a subroutine will not obey the standard calling conventions for the target architecture and will therefore not be safe to call from within a debugger.*

A subroutine entry may contain a `DW_AT_calling_convention` attribute, whose value is a constant. If this attribute is not present, or its value is the constant `DW_CC_normal`, then the subroutine may be safely called by obeying the ‘‘standard’’ calling conventions of the target architecture. If the value of the calling convention attribute is the constant `DW_CC_nocall`, the subroutine does not obey standard calling conventions, and it may not be safe for the debugger to call this subroutine.

If the semantics of the language of the compilation unit containing the subroutine entry distinguishes between ordinary subroutines and subroutines that can serve as the ‘‘main program,’’ that is, subroutines that cannot be called directly following the ordinary calling conventions, then the debugging information entry for such a subroutine may have a calling convention attribute whose value is the constant `DW_CC_program`.

*The* `DW_CC_program` *value is intended to support Fortran main programs. It is not intended as a way of finding the entry address for the program.*

### 3.3.2 Subroutine and Entry Point Return Types

If the subroutine or entry point is a function that returns a value, then its debugging information entry has a `DW_AT_type` attribute to denote the type returned by that function.

*Debugging information entries for C* `void` *functions should not have an attribute for the return type.*

*In ANSI-C there is a difference between the types of functions declared using function prototype style declarations and those declared using non-prototype declarations.*

A subroutine entry declared with a function prototype style declaration may have a `DW_AT_prototyped` attribute, whose value is a flag.

### 3.3.3 Subroutine and Entry Point Locations

A subroutine entry has a `DW_AT_low_pc` attribute whose value is the relocated address of the first machine instruction generated for the subroutine. It also has a `DW_AT_high_pc` attribute whose value is the relocated address of the first location past the last machine instruction generated for the subroutine.

*Note that for the low and high pc attributes to have meaning, DWARF makes the assumption that the code for a single subroutine is allocated in a single contiguous block of memory.*

An entry point has a `DW_AT_low_pc` attribute whose value is the relocated address of the first machine instruction generated for the entry point.

Subroutines and entry points may also have `DW_AT_segment` and `DW_AT_address_class` attributes, as appropriate, to specify which segments the code for the subroutine resides in and the addressing mode to be used in calling that subroutine.

A subroutine entry representing a subroutine declaration that is not also a definition does not have low and high pc attributes.

### 3.3.4 Declarations Owned by Subroutines and Entry Points

The declarations enclosed by a subroutine or entry point are represented by debugging information entries that are owned by the subroutine or entry point entry. Entries representing the formal parameters of the subroutine or entry point appear in the same order as the corresponding declarations in the source program.

*There is no ordering requirement on entries for declarations that are children of subroutine or entry point entries but that do not represent formal parameters. The formal parameter entries may be interspersed with other entries used by formal parameter entries, such as type entries.*

The unspecified parameters of a variable parameter list are represented by a debugging information entry with the tag `DW_TAG_unspecified_parameters`.

The entry for a subroutine or entry point that includes a Fortran common block has a child entry with the tag `DW_TAG_common_inclusion`. The common inclusion entry has a `DW_AT_common_reference` attribute whose value is a reference to the debugging entry for the common block being included (see section 4.2).

### 3.3.5 Low-Level Information

A subroutine or entry point entry may have a `DW_AT_return_addr` attribute, whose value is a location description. The location calculated is the place where the return address for the subroutine or entry point is stored.

A subroutine or entry point entry may also have a `DW_AT_frame_base` attribute, whose value is a location description that computes the ‘‘frame base’’ for the subroutine or entry point.

*The frame base for a procedure is typically an address fixed relative to the first unit of storage allocated for the procedure’s stack frame. The* `DW_AT_frame_base` *attribute can be used in several ways:*

1. *In procedures that need location lists to locate local variables, the* `DW_AT_frame_base` *can hold the needed location list, while all variables’ location descriptions can be simpler location expressions involving the frame base.*
2. *It can be used as a key in resolving "up-level" addressing with nested routines. (See* `DW_AT_static_link`*, below)*

*Some languages support nested subroutines. In such languages, it is possible to reference the local variables of an outer subroutine from within an inner subroutine. The* `DW_AT_static_link` *and* `DW_AT_frame_base` *attributes allow debuggers to support this same kind of referencing.*

If a subroutine or entry point is nested, it may have a `DW_AT_static_link` attribute, whose value is a location description that computes the frame base of the relevant instance of the subroutine that immediately encloses the subroutine or entry point.

In the context of supporting nested subroutines, the `DW_AT_frame_base` attribute value should obey the following constraints:

1. It should compute a value that does not change during the life of the procedure, and
2. The computed value should be unique among instances of the same subroutine. (For typical `DW_AT_frame_base` use, this means that a recursive subroutine’s stack frame must have non-zero size.)

*If a debugger is attempting to resolve an up-level reference to a variable, it uses the nesting structure of DWARF to determine which subroutine is the lexical parent and the* `DW_AT_static_link` *value to identify the appropriate active frame of the parent. It can then attempt to find the reference within the context of the parent.*

### 3.3.6 Types Thrown by Exceptions

*In C++ a subroutine may declare a set of types for which that subroutine may generate or ‘‘throw’’ an exception.*

If a subroutine explicitly declares that it may throw an exception for one or more types, each such type is represented by a debugging information entry with the tag `DW_TAG_thrown_type`. Each such entry is a child of the entry representing the subroutine that may throw this type. All thrown type entries should follow all entries representing the formal parameters of the subroutine and precede all entries representing the local variables or lexical blocks contained in the subroutine. Each thrown type entry contains a `DW_AT_type` attribute, whose value is a reference to an entry describing the type of the exception that may be thrown.

### 3.3.7 Function Template Instantiations

*In C++ a function template is a generic definition of a function that is instantiated differently when called with values of different types. DWARF does not represent the generic template definition, but does represent each instantiation.*

A template instantiation is represented by a debugging information entry with the tag `DW_TAG_subprogram`. With three exceptions, such an entry will contain the same attributes and have the same types of child entries as would an entry for a subroutine defined explicitly using the instantiation types. The exceptions are:

1. Each formal parameterized type declaration appearing in the template definition is represented by a debugging information entry with the tag `DW_TAG_template_type_parameter`. Each such entry has a `DW_AT_name` attribute, whose value is a null-terminated string containing the name of the formal type parameter as it appears in the source program. The template type parameter entry also has a `DW_AT_type` attribute describing the actual type by which the formal is replaced for this instantiation. All template type parameter entries should appear before the entries describing the instantiated formal parameters to the function.

2. If the compiler has generated a special compilation unit to hold the template instantiation and that compilation unit has a different name from the compilation unit containing the template definition, the name attribute for the debugging entry representing that compilation unit should be empty or omitted.

3. If the subprogram entry representing the template instantiation or any of its child entries contain declaration coordinate attributes, those attributes should refer to the source for the template definition, not to any source generated artificially by the compiler for this instantiation.

### 3.3.8 Inline Subroutines

A declaration or a definition of an inlinable subroutine is represented by a debugging information entry with the tag `DW_TAG_subprogram`. The entry for a subroutine that is explicitly declared to be available for inline expansion or that was expanded inline implicitly by the compiler has a `DW_AT_inline` attribute whose value is a constant. The set of values for the `DW_AT_inline`

| Name | Meaning |
|---|---|
| DW_INL_not_inlined | Not declared inline nor inlined by the compiler |
| DW_INL_inlined | Not declared inline but inlined by the compiler |
| DW_INL_declared_not_inlined | Declared inline but not inlined by the compiler |
| DW_INL_declared_inlined | Declared inline and inlined by the compiler |

**Figure 9.** Inline codes

attribute is given in Figure 9.

#### 3.3.8.1 Abstract Instances

For the remainder of this discussion, any debugging information entry that is owned (either directly or indirectly) by a debugging information entry that contains the DW_AT_inline attribute will be referred to as an ‘‘abstract instance entry.’’ Any subroutine entry that contains a DW_AT_inline attribute will be known as an ‘‘abstract instance root.’’ Any set of abstract instance entries that are all children (either directly or indirectly) of some abstract instance root, together with the root itself, will be known as an ‘‘abstract instance tree.’’

A debugging information entry that is a member of an abstract instance tree should not contain a DW_AT_high_pc, DW_AT_low_pc, DW_AT_location, DW_AT_return_addr, DW_AT_start_scope, or DW_AT_segment attribute.

*It would not make sense to put these attributes into abstract instance entries since such entries do not represent actual (concrete) instances and thus do not actually exist at run-time.*

The rules for the relative location of entries belonging to abstract instance trees are exactly the same as for other similar types of entries that are not abstract. Specifically, the rule that requires that an entry representing a declaration be a direct child of the entry representing the scope of the declaration applies equally to both abstract and non-abstract entries. Also, the ordering rules for formal parameter entries, member entries, and so on, all apply regardless of whether or not a given entry is abstract.

#### 3.3.8.2 Concrete Inlined Instances

Each inline expansion of an inlinable subroutine is represented by a debugging information entry with the tag DW_TAG_inlined_subroutine. Each such entry should be a direct child of the entry that represents the scope within which the inlining occurs.

Each inlined subroutine entry contains a DW_AT_low_pc attribute, representing the address of the first instruction associated with the given inline expansion. Each inlined subroutine entry also contains a DW_AT_high_pc attribute, representing the address of the first location past the last instruction associated with the inline expansion.

For the remainder of this discussion, any debugging information entry that is owned (either directly or indirectly) by a debugging information entry with the tag DW_TAG_inlined_subroutine will be referred to as a ‘‘concrete inlined instance entry.’’ Any entry that has the tag DW_TAG_inlined_subroutine will be known as a ‘‘concrete inlined instance root.’’ Any set of concrete inlined instance entries that are all children (either directly or indirectly) of some concrete inlined instance root, together with the root itself, will be known as a ‘‘concrete inlined instance tree.’’

Each concrete inlined instance tree is uniquely associated with one (and only one) abstract instance tree.

*Note, however, that the reverse is not true. Any given abstract instance tree may be associated with several different concrete inlined instance trees, or may even be associated with zero concrete inlined instance trees.*

Also, each separate entry within a given concrete inlined instance tree is uniquely associated with one particular entry in the associated abstract instance tree. In other words, there is a one-to-one mapping from entries in a given concrete inlined instance tree to the entries in the associated abstract instance tree.

*Note, however, that the reverse is not true. A given abstract instance tree that is associated with a given concrete inlined instance tree may (and quite probably will) contain more entries than the associated concrete inlined instance tree (see below).*

Concrete inlined instance entries do not have most of the attributes (except for `DW_AT_low_pc`, `DW_AT_high_pc`, `DW_AT_location`, `DW_AT_return_addr`, `DW_AT_start_scope` and `DW_AT_segment`) that such entries would otherwise normally have. In place of these omitted attributes, each concrete inlined instance entry has a `DW_AT_abstract_origin` attribute that may be used to obtain the missing information (indirectly) from the associated abstract instance entry. The value of the abstract origin attribute is a reference to the associated abstract instance entry.

For each pair of entries that are associated via a `DW_AT_abstract_origin` attribute, both members of the pair will have the same tag. So, for example, an entry with the tag `DW_TAG_local_variable` can only be associated with another entry that also has the tag `DW_TAG_local_variable.` The only exception to this rule is that the root of a concrete instance tree (which must always have the tag `DW_TAG_inlined_subroutine`) can only be associated with the root of its associated abstract instance tree (which must have the tag `DW_TAG_subprogram`).

In general, the structure and content of any given concrete instance tree will be directly analogous to the structure and content of its associated abstract instance tree. There are two exceptions to this general rule however.

1. No entries representing anonymous types are ever made a part of any concrete instance inlined tree.
2. No entries representing members of structure, union or class types are ever made a part of any concrete inlined instance tree.

*Entries that represent members and anonymous types are omitted from concrete inlined instance trees because they would simply be redundant duplicates of the corresponding entries in the associated abstract instance trees. If any entry within a concrete inlined instance tree needs to refer to an anonymous type that was declared within the scope of the relevant inline function, the reference should simply refer to the abstract instance entry for the given anonymous type.*

If an entry within a concrete inlined instance tree contains attributes describing the declaration coordinates of that entry, then those attributes should refer to the file, line and column of the original declaration of the subroutine, not to the point at which it was inlined.

#### 3.3.8.3 Out-of-Line Instances of Inline Subroutines

Under some conditions, compilers may need to generate concrete executable instances of inline subroutines other than at points where those subroutines are actually called. For the remainder of this discussion, such concrete instances of inline subroutines will be referred to as ‘‘concrete out-of-line instances.’’

*In C++, for example, taking the address of a function declared to be inline can necessitate the generation of a concrete out-of-line instance of the given function.*

The DWARF representation of a concrete out-of-line instance of an inline subroutine is essentially the same as for a concrete inlined instance of that subroutine (as described in the preceding section). The representation of such a concrete out-of-line instance makes use of `DW_AT_abstract_origin` attributes in exactly the same way as they are used for a concrete inlined instance (that is, as references to corresponding entries within the associated abstract instance tree) and, as for concrete instance trees, the entries for anonymous types and for all members are omitted.

The differences between the DWARF representation of a concrete out-of-line instance of a given subroutine and the representation of a concrete inlined instance of that same subroutine are as follows:

1. The root entry for a concrete out-of-line instance of a given inline subroutine has the same tag as does its associated (abstract) inline subroutine entry (that is, it does not have the tag `DW_TAG_inlined_subroutine`).
2. The root entry for a concrete out-of-line instance tree is always directly owned by the same parent entry that also owns the root entry of the associated abstract instance.

## 3.4 Lexical Block Entries

*A lexical block is a bracketed sequence of source statements that may contain any number of declarations. In some languages (C and C++) blocks can be nested within other blocks to any depth.*

A lexical block is represented by a debugging information entry with the tag `DW_TAG_lexical_block`.

The lexical block entry has a `DW_AT_low_pc` attribute whose value is the relocated address of the first machine instruction generated for the lexical block. The lexical block entry also has a `DW_AT_high_pc` attribute whose value is the relocated address of the first location past the last machine instruction generated for the lexical block.

If a name has been given to the lexical block in the source program, then the corresponding lexical block entry has a `DW_AT_name` attribute whose value is a null-terminated string containing the name of the lexical block as it appears in the source program.

*This is not the same as a C or C++ label (see below).*

The lexical block entry owns debugging information entries that describe the declarations within that lexical block. There is one such debugging information entry for each local declaration of an identifier or inner lexical block.

## 3.5 Label Entries

*A label is a way of identifying a source statement. A labeled statement is usually the target of one or more ‘‘go to’’ statements.*

A label is represented by a debugging information entry with the tag `DW_TAG_label`. The entry for a label should be owned by the debugging information entry representing the scope within which the name of the label could be legally referenced within the source program.

The label entry has a `DW_AT_low_pc` attribute whose value is the relocated address of the first machine instruction generated for the statement identified by the label in the source program. The label entry also has a `DW_AT_name` attribute whose value is a null-terminated string containing the name of the label as it appears in the source program.

## 3.6 With Statement Entries

*Both Pascal and Modula support the concept of a ‘‘with’’ statement. The with statement specifies a sequence of executable statements within which the fields of a record variable may be referenced, unqualified by the name of the record variable.*

A with statement is represented by a debugging information entry with the tag `DW_TAG_with_stmt`. A with statement entry has a `DW_AT_low_pc` attribute whose value is the relocated address of the first machine instruction generated for the body of the with statement. A with statement entry also has a `DW_AT_high_pc` attribute whose value is the relocated address of the first location after the last machine instruction generated for the body of the statement.

The with statement entry has a `DW_AT_type` attribute, denoting the type of record whose fields may be referenced without full qualification within the body of the statement. It also has a `DW_AT_location` attribute, describing how to find the base address of the record object referenced within the body of the with statement.

## 3.7 Try and Catch Block Entries

*In C++ a lexical block may be designated as a ‘‘catch block.’’ A catch block is an exception handler that handles exceptions thrown by an immediately preceding ‘‘try block.’’ A catch block designates the type of the exception that it can handle.*

A try block is represented by a debugging information entry with the tag `DW_TAG_try_block`. A catch block is represented by a debugging information entry with the tag `DW_TAG_catch_block`. Both try and catch block entries contain a `DW_AT_low_pc` attribute whose value is the relocated address of the first machine instruction generated for that block. These entries also contain a `DW_AT_high_pc` attribute whose value is the relocated address of the first location past the last machine instruction generated for that block.

Catch block entries have at least one child entry, an entry representing the type of exception accepted by that catch block. This child entry will have one of the tags `DW_TAG_formal_parameter` or `DW_TAG_unspecified_parameters`, and will have the same form as other parameter entries.

The first sibling of each try block entry will be a catch block entry.

# 4. DATA OBJECT AND OBJECT LIST ENTRIES

This section presents the debugging information entries that describe individual data objects: variables, parameters and constants, and lists of those objects that may be grouped in a single declaration, such as a common block.

## 4.1 Data Object Entries

Program variables, formal parameters and constants are represented by debugging information entries with the tags `DW_TAG_variable`, `DW_TAG_formal_parameter` and `DW_TAG_constant`, respectively.

*The tag* `DW_TAG_constant` *is used for languages that distinguish between variables that may have constant value and true named constants.*

The debugging information entry for a program variable, formal parameter or constant may have the following attributes:

1. A `DW_AT_name` attribute whose value is a null-terminated string containing the data object name as it appears in the source program.

   If a variable entry describes a C++ anonymous union, the name attribute is omitted or consists of a single zero byte.

2. If the name of a variable is visible outside of its enclosing compilation unit, the variable entry has a `DW_AT_external` attribute, whose value is a flag.

   *The definitions of C++ static data members of structures or classes are represented by variable entries flagged as external. Both file static and local variables in C and C++ are represented by non-external variable entries.*

3. A `DW_AT_location` attribute, whose value describes the location of a variable or parameter at run-time.

   A data object entry representing a non-defining declaration of the object will not have a location attribute, and will have the `DW_AT_declaration` attribute.

   In a variable entry representing the definition of the variable (that is, with no `DW_AT_declaration` attribute) if no location attribute is present, or if the location attribute is present but describes a null entry (as described in section 2.4), the variable is assumed to exist in the source code but not in the executable program (but see number 9, below).

   The location of a variable may be further specified with a `DW_AT_segment` attribute, if appropriate.

4. A `DW_AT_type` attribute describing the type of the variable, constant or formal parameter.

5. If the variable entry represents the defining declaration for a C++ static data member of a structure, class or union, the entry has a `DW_AT_specification` attribute, whose value is a reference to the debugging information entry representing the declaration of this data member. The referenced entry will be a child of some class, structure or union type entry.

   Variable entries containing the `DW_AT_specification` attribute do not need to duplicate information provided by the declaration entry referenced by the specification attribute. In particular, such variable entries do not need to contain attributes for the name or type of the data member whose definition they represent.

6. *Some languages distinguish between parameters whose value in the calling function can be modified by the callee (variable parameters), and parameters whose value in the calling function cannot be modified by the callee (constant parameters).*

   If a formal parameter entry represents a parameter whose value in the calling function may be modified by the callee, that entry may have a `DW_AT_variable_parameter` attribute, whose value is a flag. The absence of this attribute implies that the parameter's value in the calling function cannot be modified by the callee.

7. *Fortran90 has the concept of an optional parameter.*

   If a parameter entry represents an optional parameter, it has a `DW_AT_is_optional` attribute, whose value is a flag.

8. A formal parameter entry describing a formal parameter that has a default value may have a `DW_AT_default_value` attribute. The value of this attribute is a reference to the debugging information entry for a variable or subroutine. The default value of the parameter is the value of the variable (which may be constant) or the value returned by the subroutine. If the value of the `DW_AT_default_value` attribute is 0, it means that no default value has been specified.

9. An entry describing a variable whose value is constant and not represented by an object in the address space of the program, or an entry describing a named constant, does not have a location attribute. Such entries have a `DW_AT_const_value` attribute, whose value may be a string or any of the constant data or data block forms, as appropriate for the representation of the variable's value. The value of this attribute is the actual constant value of the variable, represented as it would be on the target architecture.

10. If the scope of an object begins sometime after the low pc value for the scope most closely enclosing the object, the object entry may have a `DW_AT_start_scope` attribute. The value of this attribute is the offset in bytes of the beginning of the scope for the object from the low pc value of the debugging information entry that defines its scope.

    *The scope of a variable may begin somewhere in the middle of a lexical block in a language that allows executable code in a block before a variable declaration, or where one declaration containing initialization code may change the scope of a subsequent declaration. For example, in the following C code:*

    ```
    float x = 99.99;

    int myfunc()
    {
                                float f = x;
                                float x = 88.99;

                                return 0;
    }
    ```

    *ANSI-C scoping rules require that the value of the variable* `x` *assigned to the variable* `f` *in the initialization sequence is the value of the global variable* `x`*, rather than the local* `x`*, because the scope of the local variable* `x` *only starts after the full declarator for the local* `x`.

## 4.2 Common Block Entries

A Fortran common block may be described by a debugging information entry with the tag DW_TAG_common_block. The common block entry has a DW_AT_name attribute whose value is a null-terminated string containing the common block name as it appears in the source program. It also has a DW_AT_location attribute whose value describes the location of the beginning of the common block. The common block entry owns debugging information entries describing the variables contained within the common block.

## 4.3 Imported Declaration Entries

*Some languages support the concept of importing into a given module declarations made in a different module.*

An imported declaration is represented by a debugging information entry with the tag DW_TAG_imported_declaration. The entry for the imported declaration has a DW_AT_name attribute whose value is a null-terminated string containing the name of the entity whose declaration is being imported as it appears in the source program. The imported declaration entry also has a DW_AT_import attribute, whose value is a reference to the debugging information entry representing the declaration that is being imported.

## 4.4 Namelist Entries

*At least one language, Fortran90, has the concept of a namelist. A namelist is an ordered list of the names of some set of declared objects. The namelist object itself may be used as a replacement for the list of names in various contexts.*

A namelist is represented by a debugging information entry with the tag DW_TAG_namelist. If the namelist itself has a name, the namelist entry has a DW_AT_name attribute, whose value is a null-terminated string containing the namelist’s name as it appears in the source program.

Each name that is part of the namelist is represented by a debugging information entry with the tag DW_TAG_namelist_item. Each such entry is a child of the namelist entry, and all of the namelist item entries for a given namelist are ordered as were the list of names they correspond to in the source program.

Each namelist item entry contains a DW_AT_namelist_item attribute whose value is a reference to the debugging information entry representing the declaration of the item whose name appears in the namelist.

# 5. TYPE ENTRIES

This section presents the debugging information entries that describe program types: base types, modified types and user-defined types.

If the scope of the declaration of a named type begins sometime after the low pc value for the scope most closely enclosing the declaration, the declaration may have a `DW_AT_start_scope` attribute. The value of this attribute is the offset in bytes of the beginning of the scope for the declaration from the low pc value of the debugging information entry that defines its scope.

## 5.1 Base Type Entries

*A base type is a data type that is not defined in terms of other data types. Each programming language has a set of base types that are considered to be built into that language.*

A base type is represented by a debugging information entry with the tag `DW_TAG_base_type`. A base type entry has a `DW_AT_name` attribute whose value is a null-terminated string describing the name of the base type as recognized by the programming language of the compilation unit containing the base type entry.

A base type entry also has a `DW_AT_encoding` attribute describing how the base type is encoded and is to be interpreted. The value of this attribute is a constant. The set of values and their meanings for the `DW_AT_encoding` attribute is given in Figure 10.

| Name | Meaning |
|---|---|
| `DW_ATE_address` | linear machine address |
| `DW_ATE_boolean` | true or false |
| `DW_ATE_complex_float` | complex floating-point number |
| `DW_ATE_float` | floating-point number |
| `DW_ATE_signed` | signed binary integer |
| `DW_ATE_signed_char` | signed character |
| `DW_ATE_unsigned` | unsigned binary integer |
| `DW_ATE_unsigned_char` | unsigned character |

**Figure 10.** Encoding attribute values

All encodings assume the representation that is ‘‘normal’’ for the target architecture.

A base type entry has a `DW_AT_byte_size` attribute, whose value is a constant, describing the size in bytes of the storage unit used to represent an object of the given type.

If the value of an object of the given type does not fully occupy the storage unit described by the byte size attribute, the base type entry may have a `DW_AT_bit_size` attribute and a `DW_AT_bit_offset` attribute, both of whose values are constants. The bit size attribute describes the actual size in bits used to represent a value of the given type. The bit offset attribute describes the offset in bits of the high order bit of a value of the given type from the high order bit of the storage unit used to contain that value.

*For example, the C type* `int` *on a machine that uses 32-bit integers would be represented by a base type entry with a name attribute whose value was ‘‘*`int`*,’’ an encoding attribute whose value was* `DW_ATE_signed` *and a byte size attribute whose value was* `4`.

## 5.2 Type Modifier Entries

A base or user-defined type may be modified in different ways in different languages. A type modifier is represented in DWARF by a debugging information entry with one of the tags given in Figure 11.

| Tag | Meaning |
|---|---|
| `DW_TAG_const_type` | C or C++ const qualified type |
| `DW_TAG_packed_type` | Pascal packed type |
| `DW_TAG_pointer_type` | The address of the object whose type is being modified |
| `DW_TAG_reference_type` | A C++ reference to the object whose type is being modified |
| `DW_TAG_volatile_type` | C or C++ volatile qualified type |

**Figure 11.** Type modifier tags

Each of the type modifier entries has a `DW_AT_type` attribute, whose value is a reference to a debugging information entry describing a base type, a user-defined type or another type modifier.

A modified type entry describing a pointer or reference type may have a `DW_AT_address_class` attribute to describe how objects having the given pointer or reference type ought to be dereferenced.

When multiple type modifiers are chained together to modify a base or user-defined type, they are ordered as if part of a right-associative expression involving the base or user-defined type.

*As examples of how type modifiers are ordered, take the following C declarations:*

```
const char * volatile p;
```

*which represents a volatile pointer to a constant character.*
*This is encoded in DWARF as:*

```
DW_TAG_volatile_type →
        DW_TAG_pointer_type →
             DW_TAG_const_type →
                   DW_TAG_base_type
```

```
volatile char * const p;
```

*on the other hand, represents a constant pointer*
*to a volatile character.*
*This is encoded as:*

```
DW_TAG_const_type →
        DW_TAG_pointer_type →
             DW_TAG_volatile_type →
                   DW_TAG_base_type
```

## 5.3 Typedef Entries

Any arbitrary type named via a typedef is represented by a debugging information entry with the tag `DW_TAG_typedef`. The typedef entry has a `DW_AT_name` attribute whose value is a null-terminated string containing the name of the typedef as it appears in the source program. The typedef entry also contains a `DW_AT_type` attribute.

If the debugging information entry for a typedef represents a declaration of the type that is not also a definition, it does not contain a type attribute.

## 5.4 Array Type Entries

*Many languages share the concept of an ‘‘array,’’ which is a table of components of identical type.*

An array type is represented by a debugging information entry with the tag `DW_TAG_array_type`.

If a name has been given to the array type in the source program, then the corresponding array type entry has a `DW_AT_name` attribute whose value is a null-terminated string containing the array type name as it appears in the source program.

The array type entry describing a multidimensional array may have a `DW_AT_ordering` attribute whose constant value is interpreted to mean either row-major or column-major ordering of array elements. The set of values and their meanings for the ordering attribute are listed in Figure 12. If no ordering attribute is present, the default ordering for the source language (which is indicated by the `DW_AT_language` attribute of the enclosing compilation unit entry) is assumed.

| |
|---|
| `DW_ORD_col_major` |
| `DW_ORD_row_major` |

**Figure 12.** Array ordering

The ordering attribute may optionally appear on one-dimensional arrays; it will be ignored.

An array type entry has a `DW_AT_type` attribute describing the type of each element of the array.

If the amount of storage allocated to hold each element of an object of the given array type is different from the amount of storage that is normally allocated to hold an individual object of the indicated element type, then the array type entry has a `DW_AT_stride_size` attribute, whose constant value represents the size in bits of each element of the array.

If the size of the entire array can be determined statically at compile time, the array type entry may have a `DW_AT_byte_size` attribute, whose constant value represents the total size in bytes of an instance of the array type.

*Note that if the size of the array can be determined statically at compile time, this value can usually be computed by multiplying the number of array elements by the size of each element.*

Each array dimension is described by a debugging information entry with either the tag `DW_TAG_subrange_type` or the tag `DW_TAG_enumeration_type`. These entries are children of the array type entry and are ordered to reflect the appearance of the dimensions in the source program (i.e. leftmost dimension first, next to leftmost second, and so on).

*In languages, such as ANSI-C, in which there is no concept of a ‘‘multidimensional array,’’ an array of arrays may be represented by a debugging information entry for a multidimensional array.*

## 5.5 Structure, Union, and Class Type Entries

*The languages C, C++, and Pascal, among others, allow the programmer to define types that are collections of related components. In C and C++, these collections are called ‘‘structures.’’ In Pascal, they are called ‘‘records.’’ The components may be of different types. The components are called ‘‘members’’ in C and C++, and ‘‘fields’’ in Pascal.*

*The components of these collections each exist in their own space in computer memory. The components of a C or C++ ‘‘union’’ all coexist in the same memory.*

*Pascal and other languages have a ‘‘discriminated union,’’ also called a ‘‘variant record.’’ Here, selection of a number of alternative substructures (‘‘variants’’) is based on the value of a component that is not part of any of those substructures (the ‘‘discriminant’’).*

*Among the languages discussed in this document, the ‘‘class’’ concept is unique to C++. A class is similar to a structure. A C++ class or structure may have ‘‘member functions’’ which are subroutines that are within the scope of a class or structure.*

### 5.5.1 General Structure Description

Structure, union, and class types are represented by debugging information entries with the tags `DW_TAG_structure_type`, `DW_TAG_union_type` and `DW_TAG_class_type`, respectively. If a name has been given to the structure, union, or class in the source program, then the corresponding structure type, union type, or class type entry has a `DW_AT_name` attribute whose value is a null-terminated string containing the type name as it appears in the source program.

If the size of an instance of the structure type, union type, or class type entry can be determined statically at compile time, the entry has a `DW_AT_byte_size` attribute whose constant value is the number of bytes required to hold an instance of the structure, union, or class, and any padding bytes.

*For C and C++, an incomplete structure, union or class type is represented by a structure, union or class entry that does not have a byte size attribute and that has a* `DW_AT_declaration` *attribute.*

The members of a structure, union, or class are represented by debugging information entries that are owned by the corresponding structure type, union type, or class type entry and appear in the same order as the corresponding declarations in the source program.

*Data member declarations occurring within the declaration of a structure, union or class type are considered to be ‘‘definitions’’ of those members, with the exception of C++ ‘‘static’’ data members, whose definitions appear outside of the declaration of the enclosing structure, union or class type. Function member declarations appearing within a structure, union or class type declaration are definitions only if the body of the function also appears within the type declaration.*

If the definition for a given member of the structure, union or class does not appear within the body of the declaration, that member also has a debugging information entry describing its definition. That entry will have a `DW_AT_specification` attribute referencing the debugging entry owned by the body of the structure, union or class debugging entry and representing a non-defining declaration of the data or function member. The referenced entry will not have information about the location of that member (low and high pc attributes for function members, location descriptions for data members) and will have a `DW_AT_declaration` attribute.

### 5.5.2 Derived Classes and Structures

The class type or structure type entry that describes a derived class or structure owns debugging information entries describing each of the classes or structures it is derived from, ordered as they were in the source program. Each such entry has the tag `DW_TAG_inheritance`.

An inheritance entry has a DW_AT_type attribute whose value is a reference to the debugging information entry describing the structure or class from which the parent structure or class of the inheritance entry is derived. It also has a DW_AT_data_member_location attribute, whose value is a location description describing the location of the beginning of the data members contributed to the entire class by this subobject relative to the beginning address of the data members of the entire class.

An inheritance entry may have a DW_AT_accessibility attribute. If no accessibility attribute is present, private access is assumed. If the structure or class referenced by the inheritance entry serves as a virtual base class, the inheritance entry has a DW_AT_virtuality attribute.

*In C++, a derived class may contain access declarations that change the accessibility of individual class members from the overall accessibility specified by the inheritance declaration. A single access declaration may refer to a set of overloaded names.*

If a derived class or structure contains access declarations, each such declaration may be represented by a debugging information entry with the tag DW_TAG_access_declaration. Each such entry is a child of the structure or class type entry.

An access declaration entry has a DW_AT_name attribute, whose value is a null-terminated string representing the name used in the declaration in the source program, including any class or structure qualifiers.

An access declaration entry also has a DW_AT_accessibility attribute describing the declared accessibility of the named entities.

### 5.5.3 Friends

Each ‘‘friend’’ declared by a structure, union or class type may be represented by a debugging information entry that is a child of the structure, union or class type entry; the friend entry has the tag DW_TAG_friend.

A friend entry has a DW_AT_friend attribute, whose value is a reference to the debugging information entry describing the declaration of the friend.

### 5.5.4 Structure Data Member Entries

A data member (as opposed to a member function) is represented by a debugging information entry with the tag DW_TAG_member. The member entry for a named member has a DW_AT_name attribute whose value is a null-terminated string containing the member name as it appears in the source program. If the member entry describes a C++ anonymous union, the name attribute is omitted or consists of a single zero byte.

The structure data member entry has a DW_AT_type attribute to denote the type of that member.

If the member entry is defined in the structure or class body, it has a DW_AT_data_member_location attribute whose value is a location description that describes the location of that member relative to the base address of the structure, union, or class that most closely encloses the corresponding member declaration.

*The addressing expression represented by the location description for a structure data member expects the base address of the structure data member to be on the expression stack before being evaluated.*

*The location description for a data member of a union may be omitted, since all data members of a union begin at the same address.*

If the member entry describes a bit field, then that entry has the following attributes:

1. A `DW_AT_byte_size` attribute whose constant value is the number of bytes that contain an instance of the bit field and any padding bits.

   *The byte size attribute may be omitted if the size of the object containing the bit field can be inferred from the type attribute of the data member containing the bit field.*

2. A `DW_AT_bit_offset` attribute whose constant value is the number of bits to the left of the leftmost (most significant) bit of the bit field value.

3. A `DW_AT_bit_size` attribute whose constant value is the number of bits occupied by the bit field value.

The location description for a bit field calculates the address of an anonymous object containing the bit field. The address is relative to the structure, union, or class that most closely encloses the bit field declaration. The number of bytes in this anonymous object is the value of the byte size attribute of the bit field. The offset (in bits) from the most significant bit of the anonymous object to the most significant bit of the bit field is the value of the bit offset attribute.

*For example, take one possible representation of the following structure definition in both big and little endian byte orders:*

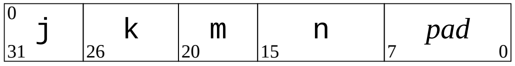
```
struct S {
                 int   j:5;
                 int   k:6;
                 int   m:5;
                 int   n:8;
};
```

*In both cases, the location descriptions for the debugging information entries for* `j`, `k`, `m` *and* `n` *describe the address of the same 32-bit word that contains all three members. (In the big-endian case, the location description addresses the most significant byte, in the little-endian case, the least significant). The following diagram shows the structure layout and lists the bit offsets for each case. The offsets are from the most significant bit of the object addressed by the location description.*

Bit Offsets:

```
    j:0
    k:5
    m:11
    n:16
```

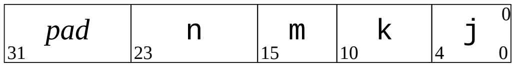
Big-Endian



Bit Offsets:

```
    j:27
    k:21
    m:16
    n:8
```

Little-Endian



### 5.5.5 Structure Member Function Entries

A member function is represented in the debugging information by a debugging information entry with the tag `DW_TAG_subprogram`. The member function entry may contain the same attributes and follows the same rules as non-member global subroutine entries (see section 3.3).

If the member function entry describes a virtual function, then that entry has a `DW_AT_virtuality` attribute.

An entry for a virtual function also has a `DW_AT_vtable_elem_location` attribute whose value contains a location description yielding the address of the slot for the function within the virtual function table for the enclosing class or structure.

If a subroutine entry represents the defining declaration of a member function and that definition appears outside of the body of the enclosing class or structure declaration, the subroutine entry has a `DW_AT_specification` attribute, whose value is a reference to the debugging information entry representing the declaration of this function member. The referenced entry will be a child of some class or structure type entry.

Subroutine entries containing the `DW_AT_specification` attribute do not need to duplicate information provided by the declaration entry referenced by the specification attribute. In particular, such entries do not need to contain attributes for the name or return type of the function member whose definition they represent.

### 5.5.6 Class Template Instantiations

*In C++ a class template is a generic definition of a class type that is instantiated differently when an instance of the class is declared or defined. The generic description of the class may include both parameterized types and parameterized constant values. DWARF does not represent the generic template definition, but does represent each instantiation.*

A class template instantiation is represented by a debugging information with the tag `DW_TAG_class_type`. With four exceptions, such an entry will contain the same attributes and have the same types of child entries as would an entry for a class type defined explicitly using the instantiation types and values. The exceptions are:

1. Each formal parameterized type declaration appearing in the template definition is represented by a debugging information entry with the tag `DW_TAG_template_type_parameter`. Each such entry has a `DW_AT_name` attribute, whose value is a null-terminated string containing the name of the formal type parameter as it appears in the source program. The template type parameter entry also has a `DW_AT_type` attribute describing the actual type by which the formal is replaced for this instantiation.

2. Each formal parameterized value declaration appearing in the templated definition is represented by a debugging information entry with the tag `DW_TAG_template_value_parameter`. Each such entry has a `DW_AT_name` attribute, whose value is a null-terminated string containing the name of the formal value parameter as it appears in the source program. The template value parameter entry also has a `DW_AT_type` attribute describing the type of the parameterized value. Finally, the template value parameter entry has a `DW_AT_const_value` attribute, whose value is the actual constant value of the value parameter for this instantiation as represented on the target architecture.

3. If the compiler has generated a special compilation unit to hold the template instantiation and that compilation unit has a different name from the compilation unit containing the template definition, the name attribute for the debugging entry representing that compilation unit should be empty or omitted.
4. If the class type entry representing the template instantiation or any of its child entries contain declaration coordinate attributes, those attributes should refer to the source for the template definition, not to any source generated artificially by the compiler.

### 5.5.7 Variant Entries

A variant part of a structure is represented by a debugging information entry with the tag `DW_TAG_variant_part` and is owned by the corresponding structure type entry.

If the variant part has a discriminant, the discriminant is represented by a separate debugging information entry which is a child of the variant part entry. This entry has the form of a structure data member entry. The variant part entry will have a `DW_AT_discr` attribute whose value is a reference to the member entry for the discriminant.

If the variant part does not have a discriminant (tag field), the variant part entry has a `DW_AT_type` attribute to represent the tag type.

Each variant of a particular variant part is represented by a debugging information entry with the tag `DW_TAG_variant` and is a child of the variant part entry. The value that selects a given variant may be represented in one of three ways. The variant entry may have a `DW_AT_discr_value` attribute whose value represents a single case label. The value of this attribute is encoded as an LEB128 number. The number is signed if the tag type for the variant part containing this variant is a signed type. The number is unsigned if the tag type is an unsigned type.

Alternatively, the variant entry may contain a `DW_AT_discr_list` attribute, whose value represents a list of discriminant values. This list is represented by any of the block forms and may contain a mixture of case labels and label ranges. Each item on the list is prefixed with a discriminant value descriptor that determines whether the list item represents a single label or a label range. A single case label is represented as an LEB128 number as defined above for the `DW_AT_discr_value` attribute. A label range is represented by two LEB128 numbers, the low value of the range followed by the high value. Both values follow the rules for signedness just described. The discriminant value descriptor is a constant that may have one of the values given in Figure 13.

```
DW_DSC_label
DW_DSC_range
```

**Figure 13.** Discriminant descriptor values

If a variant entry has neither a `DW_AT_discr_value` attribute nor a `DW_AT_discr_list` attribute, or if it has a `DW_AT_discr_list` attribute with 0 size, the variant is a default variant.

The components selected by a particular variant are represented by debugging information entries owned by the corresponding variant entry and appear in the same order as the corresponding declarations in the source program.

## 5.6 Enumeration Type Entries

*An ‘‘enumeration type’’ is a scalar that can assume one of a fixed number of symbolic values.*

An enumeration type is represented by a debugging information entry with the tag DW_TAG_enumeration_type.

If a name has been given to the enumeration type in the source program, then the corresponding enumeration type entry has a DW_AT_name attribute whose value is a null-terminated string containing the enumeration type name as it appears in the source program. These entries also have a DW_AT_byte_size attribute whose constant value is the number of bytes required to hold an instance of the enumeration.

Each enumeration literal is represented by a debugging information entry with the tag DW_TAG_enumerator. Each such entry is a child of the enumeration type entry, and the enumerator entries appear in the same order as the declarations of the enumeration literals in the source program.

Each enumerator entry has a DW_AT_name attribute, whose value is a null-terminated string containing the name of the enumeration literal as it appears in the source program. Each enumerator entry also has a DW_AT_const_value attribute, whose value is the actual numeric value of the enumerator as represented on the target system.

## 5.7 Subroutine Type Entries

*It is possible in C to declare pointers to subroutines that return a value of a specific type. In both ANSI C and C++, it is possible to declare pointers to subroutines that not only return a value of a specific type, but accept only arguments of specific types. The type of such pointers would be described with a ‘‘pointer to’’ modifier applied to a user-defined type.*

A subroutine type is represented by a debugging information entry with the tag DW_TAG_subroutine_type. If a name has been given to the subroutine type in the source program, then the corresponding subroutine type entry has a DW_AT_name attribute whose value is a null-terminated string containing the subroutine type name as it appears in the source program.

If the subroutine type describes a function that returns a value, then the subroutine type entry has a DW_AT_type attribute to denote the type returned by the subroutine. If the types of the arguments are necessary to describe the subroutine type, then the corresponding subroutine type entry owns debugging information entries that describe the arguments. These debugging information entries appear in the order that the corresponding argument types appear in the source program.

*In ANSI-C there is a difference between the types of functions declared using function prototype style declarations and those declared using non-prototype declarations.*

A subroutine entry declared with a function prototype style declaration may have a DW_AT_prototyped attribute, whose value is a flag.

Each debugging information entry owned by a subroutine type entry has a tag whose value has one of two possible interpretations.

1. Each debugging information entry that is owned by a subroutine type entry and that defines a single argument of a specific type has the tag DW_TAG_formal_parameter.

The formal parameter entry has a type attribute to denote the type of the corresponding formal parameter.

2. The unspecified parameters of a variable parameter list are represented by a debugging information entry owned by the subroutine type entry with the tag `DW_TAG_unspecified_parameters`.

## 5.8 String Type Entries

*A ''string'' is a sequence of characters that have specific semantics and operations that separate them from arrays of characters. Fortran is one of the languages that has a string type.*

A string type is represented by a debugging information entry with the tag `DW_TAG_string_type`. If a name has been given to the string type in the source program, then the corresponding string type entry has a `DW_AT_name` attribute whose value is a null-terminated string containing the string type name as it appears in the source program.

The string type entry may have a `DW_AT_string_length` attribute whose value is a location description yielding the location where the length of the string is stored in the program. The string type entry may also have a `DW_AT_byte_size` attribute, whose constant value is the size in bytes of the data to be retrieved from the location referenced by the string length attribute. If no byte size attribute is present, the size of the data to be retrieved is the same as the size of an address on the target machine.

If no string length attribute is present, the string type entry may have a `DW_AT_byte_size` attribute, whose constant value is the length in bytes of the string.

## 5.9 Set Entries

*Pascal provides the concept of a ''set,'' which represents a group of values of ordinal type.*

A set is represented by a debugging information entry with the tag `DW_TAG_set_type`. If a name has been given to the set type, then the set type entry has a `DW_AT_name` attribute whose value is a null-terminated string containing the set type name as it appears in the source program.

The set type entry has a `DW_AT_type` attribute to denote the type of an element of the set.

If the amount of storage allocated to hold each element of an object of the given set type is different from the amount of storage that is normally allocated to hold an individual object of the indicated element type, then the set type entry has a `DW_AT_byte_size` attribute, whose constant value represents the size in bytes of an instance of the set type.

## 5.10 Subrange Type Entries

*Several languages support the concept of a ''subrange'' type object. These objects can represent a subset of the values that an object of the basis type for the subrange can represent. Subrange type entries may also be used to represent the bounds of array dimensions.*

A subrange type is represented by a debugging information entry with the tag `DW_TAG_subrange_type`. If a name has been given to the subrange type, then the subrange type entry has a `DW_AT_name` attribute whose value is a null-terminated string containing the subrange type name as it appears in the source program.

The subrange entry may have a `DW_AT_type` attribute to describe the type of object of whose values this subrange is a subset.

If the amount of storage allocated to hold each element of an object of the given subrange type is different from the amount of storage that is normally allocated to hold an individual object of the indicated element type, then the subrange type entry has a `DW_AT_byte_size` attribute, whose constant value represents the size in bytes of each element of the subrange type.

The subrange entry may have the attributes `DW_AT_lower_bound` and `DW_AT_upper_bound` to describe, respectively, the lower and upper bound values of the subrange. The `DW_AT_upper_bound` attribute may be replaced by a `DW_AT_count` attribute, whose value describes the number of elements in the subrange rather than the value of the last element. If a bound or count value is described by a constant not represented in the program's address space and can be represented by one of the constant attribute forms, then the value of the lower or upper bound or count attribute may be one of the constant types. Otherwise, the value of the lower or upper bound or count attribute is a reference to a debugging information entry describing an object containing the bound value or itself describing a constant value.

If either the lower or upper bound or count values are missing, the bound value is assumed to be a language-dependent default constant.

*The default lower bound value for C or C++ is 0. For Fortran, it is 1. No other default values are currently defined by DWARF.*

If the subrange entry has no type attribute describing the basis type, the basis type is assumed to be the same as the object described by the lower bound attribute (if it references an object). If there is no lower bound attribute, or it does not reference an object, the basis type is the type of the upper bound or count attribute (if it references an object). If there is no upper bound or count attribute or it does not reference an object, the type is assumed to be the same type, in the source language of the compilation unit containing the subrange entry, as a signed integer with the same size as an address on the target machine.

## 5.11 Pointer to Member Type Entries

*In C++, a pointer to a data or function member of a class or structure is a unique type.*

A debugging information entry representing the type of an object that is a pointer to a structure or class member has the tag `DW_TAG_ptr_to_member_type`.

If the pointer to member type has a name, the pointer to member entry has a `DW_AT_name` attribute, whose value is a null-terminated string containing the type name as it appears in the source program.

The pointer to member entry has a `DW_AT_type` attribute to describe the type of the class or structure member to which objects of this type may point.

The pointer to member entry also has a `DW_AT_containing_type` attribute, whose value is a reference to a debugging information entry for the class or structure to whose members objects of this type may point.

Finally, the pointer to member entry has a `DW_AT_use_location` attribute whose value is a location description that computes the address of the member of the class or structure to which the pointer to member type entry can point.

*The method used to find the address of a given member of a class or structure is common to any instance of that class or structure and to any instance of the pointer or member type. The method is thus associated with the type entry, rather than with each instance of the type.*

*The* `DW_AT_use_location` *expression, however, cannot be used on its own, but must be used in conjunction with the location expressions for a particular object of the given pointer to member type and for a particular structure or class instance. The* `DW_AT_use_location` *attribute expects two values to be pushed onto the location expression stack before the* `DW_AT_use_location` *expression is evaluated. The first value pushed should be the value of the pointer to member object itself. The second value pushed should be the base address of the entire structure or union instance containing the member whose address is being calculated.*

*So, for an expression like*

```
object.*mbr_ptr
```

*where* `mbr_ptr` *has some pointer to member type, a debugger should:*

1. *Push the value of* `mbr_ptr` *onto the location expression stack.*
2. *Push the base address of* `object` *onto the location expression stack.*
3. *Evaluate the* `DW_AT_use_location` *expression for the type of* `mbr_ptr`.

## 5.12 File Type Entries

*Some languages, such as Pascal, provide a first class data type to represent files.*

A file type is represented by a debugging information entry with the tag `DW_TAG_file_type.` If the file type has a name, the file type entry has a `DW_AT_name` attribute, whose value is a null-terminated string containing the type name as it appears in the source program.

The file type entry has a `DW_AT_type` attribute describing the type of the objects contained in the file.

The file type entry also has a `DW_AT_byte_size` attribute, whose value is a constant representing the size in bytes of an instance of this file type.

# 6. OTHER DEBUGGING INFORMATION

This section describes debugging information that is not represented in the form of debugging information entries and is not contained within the `.debug_info` section.

## 6.1 Accelerated Access

*A debugger frequently needs to find the debugging information for a program object defined outside of the compilation unit where the debugged program is currently stopped. Sometimes it will know only the name of the object; sometimes only the address. To find the debugging information associated with a global object by name, using the DWARF debugging information entries alone, a debugger would need to run through all entries at the highest scope within each compilation unit. For lookup by address, for a subroutine, a debugger can use the low and high pc attributes of the compilation unit entries to quickly narrow down the search, but these attributes only cover the range of addresses for the text associated with a compilation unit entry. To find the debugging information associated with a data object, an exhaustive search would be needed. Furthermore, any search through debugging information entries for different compilation units within a large program would potentially require the access of many memory pages, probably hurting debugger performance.*

To make lookups of program objects by name or by address faster, a producer of DWARF information may provide two different types of tables containing information about the debugging information entries owned by a particular compilation unit entry in a more condensed format.

### 6.1.1 Lookup by Name

For lookup by name, a table is maintained in a separate object file section called `.debug_pubnames`. The table consists of sets of variable length entries, each set describing the names of global objects whose definitions or declarations are represented by debugging information entries owned by a single compilation unit. Each set begins with a header containing four values: the total length of the entries for that set, not including the length field itself, a version number, the offset from the beginning of the `.debug_info` section of the compilation unit entry referenced by the set and the size in bytes of the contents of the `.debug_info` section generated to represent that compilation unit. This header is followed by a variable number of offset/name pairs. Each pair consists of the offset from the beginning of the compilation unit entry corresponding to the current set to the debugging information entry for the given object, followed by a null-terminated character string representing the name of the object as given by the `DW_AT_name` attribute of the referenced debugging entry. Each set of names is terminated by zero.

In the case of the name of a static data member or function member of a C++ structure, class or union, the name presented in the `.debug_pubnames` section is not the simple name given by the `DW_AT_name` attribute of the referenced debugging entry, but rather the fully class qualified name of the data or function member.

### 6.1.2 Lookup by Address

For lookup by address, a table is maintained in a separate object file section called `.debug_aranges`. The table consists of sets of variable length entries, each set describing the portion of the program’s address space that is covered by a single compilation unit. Each set begins with a header containing five values:

1. The total length of the entries for that set, not including the length field itself.
2. A version number.
3. The offset from the beginning of the `.debug_info` section of the compilation unit entry referenced by the set.
4. The size in bytes of an address on the target architecture. For segmented addressing, this is the size of the offset portion of the address.
5. The size in bytes of a segment descriptor on the target architecture. If the target system uses a flat address space, this value is 0.

This header is followed by a variable number of address range descriptors. Each descriptor is a pair consisting of the beginning address of a range of text or data covered by some entry owned by the corresponding compilation unit entry, followed by the length of that range. A particular set is terminated by an entry consisting of two zeroes. By scanning the table, a debugger can quickly decide which compilation unit to look in to find the debugging information for an object that has a given address.

## 6.2 Line Number Information

*A source-level debugger will need to know how to associate statements in the source files with the corresponding machine instruction addresses in the executable object or the shared objects used by that executable object. Such an association would make it possible for the debugger user to specify machine instruction addresses in terms of source statements. This would be done by specifying the line number and the source file containing the statement. The debugger can also use this information to display locations in terms of the source files and to single step from statement to statement.*

As mentioned in section 3.1, above, the line number information generated for a compilation unit is represented in the `.debug_line` section of an object file and is referenced by a corresponding compilation unit debugging information entry in the `.debug_info` section.

*If space were not a consideration, the information provided in the* `.debug_line` *section could be represented as a large matrix, with one row for each instruction in the emitted object code. The matrix would have columns for:*

— *the source file name*

— *the source line number*

— *the source column number*

— *whether this instruction is the beginning of a source statement*

— *whether this instruction is the beginning of a basic block.*

*Such a matrix, however, would be impractically large. We shrink it with two techniques. First, we delete from the matrix each row whose file, line and source column information is identical with that of its predecessors. Second, we design a byte-coded language for a state machine and store a stream of bytes in the object file instead of the matrix. This language can be much more compact than the matrix. When a consumer of the statement information executes, it must ‘‘run’’ the state machine to generate the matrix for each compilation unit it is interested in. The concept of an encoded matrix also leaves room for expansion. In the future, columns can be added to the matrix to encode other things that are related to individual instruction addresses.*

### 6.2.1 Definitions

The following terms are used in the description of the line number information format:

state machine
The hypothetical machine used by a consumer of the line number information to expand the byte-coded instruction stream into a matrix of line number information.

statement program
A series of byte-coded line number information instructions representing one compilation unit.

basic block
A sequence of instructions that is entered only at the first instruction and exited only at the last instruction. We define a procedure invocation to be an exit from a basic block.

sequence
A series of contiguous target machine instructions. One compilation unit may emit multiple sequences (that is, not all instructions within a compilation unit are assumed to be contiguous).

sbyte
Small signed integer.

ubyte
Small unsigned integer.

uhalf
Medium unsigned integer.

sword
Large signed integer.

uword
Large unsigned integer.

LEB128
Variable length signed and unsigned data. See section 7.6.

### 6.2.2 State Machine Registers

The statement information state machine has the following registers:

`address`
The program-counter value corresponding to a machine instruction generated by the compiler.

`file`
An unsigned integer indicating the identity of the source file corresponding to a machine instruction.

`line`
An unsigned integer indicating a source line number. Lines are numbered beginning at 1. The compiler may emit the value 0 in cases where an instruction cannot be attributed to any source line.

`column`
An unsigned integer indicating a column number within a source line. Columns are numbered beginning at 1. The value 0 is reserved to indicate that a statement begins at the ‘‘left edge’’ of the line.

`is_stmt`
A boolean indicating that the current instruction is the beginning of a statement.

`basic_block`
A boolean indicating that the current instruction is the beginning of a basic block.

`end_sequence`
A boolean indicating that the current address is that of the first byte after the end of a sequence of target machine instructions.

At the beginning of each sequence within a statement program, the state of the registers is:

| | |
|---|---|
| address | 0 |
| file | 1 |
| line | 1 |
| column | 0 |
| is_stmt | determined by default_is_stmt in the statement program prologue |
| basic_block | ‘‘false’’ |
| end_sequence | ‘‘false’’ |

### 6.2.3 Statement Program Instructions

The state machine instructions in a statement program belong to one of three categories:

special opcodes These have a ubyte opcode field and no arguments. Most of the instructions in a statement program are special opcodes.

standard opcodes These have a ubyte opcode field which may be followed by zero or more LEB128 arguments (except for DW_LNS_fixed_advance_pc, see below). The opcode implies the number of arguments and their meanings, but the statement program prologue also specifies the number of arguments for each standard opcode.

extended opcodes These have a multiple byte format. The first byte is zero; the next bytes are an unsigned LEB128 integer giving the number of bytes in the instruction itself (does not include the first zero byte or the size). The remaining bytes are the instruction itself.

### 6.2.4 The Statement Program Prologue

The optimal encoding of line number information depends to a certain degree upon the architecture of the target machine. The statement program prologue provides information used by consumers in decoding the statement program instructions for a particular compilation unit and also provides information used throughout the rest of the statement program. The statement program for each compilation unit begins with a prologue containing the following fields in order:

1. total_length (uword)
   The size in bytes of the statement information for this compilation unit (not including the total_length field itself).

2. version (uhalf)
   Version identifier for the statement information format.

3. prologue_length (uword)
   The number of bytes following the prologue_length field to the beginning of the first byte of the statement program itself.

4. minimum_instruction_length (ubyte)
   The size in bytes of the smallest target machine instruction. Statement program opcodes that alter the address register first multiply their operands by this value.

5. default_is_stmt (ubyte)
   The initial value of the is_stmt register.

   *A simple code generator that emits machine instructions in the order implied by the source program would set this to ‘‘true,’’ and every entry in the matrix would represent a* *statement boundary. A pipeline scheduling code generator would set this to ‘‘false’’ and emit a specific statement program opcode for each instruction that represented a statement boundary.*

6. `line_base` (sbyte)
   This parameter affects the meaning of the special opcodes. See below.

7. `line_range` (ubyte)
   This parameter affects the meaning of the special opcodes. See below.

8. `opcode_base` (ubyte)
   The number assigned to the first special opcode.

9. `standard_opcode_lengths` (array of ubyte)
   This array specifies the number of LEB128 operands for each of the standard opcodes. The first element of the array corresponds to the opcode whose value is 1, and the last element corresponds to the opcode whose value is `opcode_base - 1`. By increasing `opcode_base`, and adding elements to this array, new standard opcodes can be added, while allowing consumers who do not know about these new opcodes to be able to skip them.

10. `include_directories` (sequence of path names)
    The sequence contains an entry for each path that was searched for included source files in this compilation. (The paths include those directories specified explicitly by the user for the compiler to search and those the compiler searches without explicit direction). Each path entry is either a full path name or is relative to the current directory of the compilation. The current directory of the compilation is understood to be the first entry and is not explicitly represented. Each entry is a null-terminated string containing a full path name. The last entry is followed by a single null byte.

11. `file_names` (sequence of file entries)
    The sequence contains an entry for each source file that contributed to the statement information for this compilation unit or is used in other contexts, such as in a declaration coordinate or a macro file inclusion. Each entry has a null-terminated string containing the file name, an unsigned LEB128 number representing the directory index of the directory in which the file was found, an unsigned LEB128 number representing the time of last modification for the file and an unsigned LEB128 number representing the length in bytes of the file. A compiler may choose to emit LEB128(0) for the time and length fields to indicate that this information is not available. The last entry is followed by a single null byte.

    The directory index represents an entry in the `include_directories` section. The index is LEB128(0) if the file was found in the current directory of the compilation, LEB128(1) if it was found in the first directory in the `include_directories` section, and so on. The directory index is ignored for file names that represent full path names.

    The statement program assigns numbers to each of the file entries in order, beginning with 1, and uses those numbers instead of file names in the `file` register.

    A compiler may generate a single null byte for the file names field and define file names using the extended opcode `DEFINE_FILE`.

### 6.2.5 The Statement Program

As stated before, the goal of a statement program is to build a matrix representing one compilation unit, which may have produced multiple sequences of target-machine instructions. Within a sequence, addresses may only increase. (Line numbers may decrease in cases of pipeline scheduling.)

#### 6.2.5.1 Special Opcodes

Each 1-byte special opcode has the following effect on the state machine:

1. Add a signed integer to the `line` register.
2. Multiply an unsigned integer by the `minimum_instruction_length` field of the statement program prologue and add the result to the `address` register.
3. Append a row to the matrix using the current values of the state machine registers.
4. Set the `basic_block` register to ‘‘false.’’

All of the special opcodes do those same four things; they differ from one another only in what values they add to the `line` and `address` registers.

*Instead of assigning a fixed meaning to each special opcode, the statement program uses several parameters in the prologue to configure the instruction set. There are two reasons for this. First, although the opcode space available for special opcodes now ranges from 10 through 255, the lower bound may increase if one adds new standard opcodes. Thus, the* `opcode_base` *field of the statement program prologue gives the value of the first special opcode. Second, the best choice of special-opcode meanings depends on the target architecture. For example, for a RISC machine where the compiler-generated code interleaves instructions from different lines to schedule the pipeline, it is important to be able to add a negative value to the* `line` *register to express the fact that a later instruction may have been emitted for an earlier source line. For a machine where pipeline scheduling never occurs, it is advantageous to trade away the ability to decrease the* `line` *register (a standard opcode provides an alternate way to decrease the line number) in return for the ability to add larger positive values to the* `address` *register. To permit this variety of strategies, the statement program prologue defines a* `line_base` *field that specifies the minimum value which a special opcode can add to the* `line` *register and a* `line_range` *field that defines the range of values it can add to the* `line` *register.*

A special opcode value is chosen based on the amount that needs to be added to the `line` and `address` registers. The maximum line increment for a special opcode is the value of the `line_base` field in the prologue, plus the value of the `line_range` field, minus 1 (`line base + line range - 1`). If the desired line increment is greater than the maximum line increment, a standard opcode must be used instead of a special opcode. The ‘‘address advance’’ is calculated by dividing the desired address increment by the `minimum_instruction_length` field from the prologue. The special opcode is then calculated using the following formula:

```
        opcode = (desired line increment - line_base) +
                (line_range * address advance) + opcode_base
```

If the resulting opcode is greater than 255, a standard opcode must be used instead.

To decode a special opcode, subtract the `opcode_base` from the opcode itself. The amount to increment the `address` register is the adjusted opcode divided by the `line_range`. The amount to increment the `line` register is the `line_base` plus the result of the adjusted opcode modulo the `line_range`. That is,

```
    line increment = line_base + (adjusted opcode % line_range)
```

*As an example, suppose that the* `opcode_base` *is 16,* `line_base` *is -1 and* `line_range` *is 4. This means that we can use a special opcode whenever two successive rows in the matrix have source line numbers differing by any value within the range [-1, 2] (and, because of the limited number of opcodes available, when the difference between addresses is within the range [0, 59]).*

*The opcode mapping would be:*

| Opcode | Line advance | Address advance |
|---|---|---|
| 16 | -1 | 0 |
| 17 | 0 | 0 |
| 18 | 1 | 0 |
| 19 | 2 | 0 |
| 20 | -1 | 1 |
| 21 | 0 | 1 |
| 22 | 1 | 1 |
| 23 | 2 | 1 |
| 253 | 0 | 59 |
| 254 | 1 | 59 |
| 255 | 2 | 59 |

There is no requirement that the expression `255 - line_base + 1` be an integral multiple of `line_range`.

#### 6.2.5.2 Standard Opcodes

There are currently 9 standard ubyte opcodes. In the future additional ubyte opcodes may be defined by setting the `opcode_base` field in the statement program prologue to a value greater than 10.

1. `DW_LNS_copy`
   Takes no arguments. Append a row to the matrix using the current values of the state-machine registers. Then set the `basic_block` register to ‘‘false.’’

2. `DW_LNS_advance_pc`
   Takes a single unsigned LEB128 operand, multiplies it by the `minimum_instruction_length` field of the prologue, and adds the result to the `address` register of the state machine.

3. `DW_LNS_advance_line`
   Takes a single signed LEB128 operand and adds that value to the `line` register of the state machine.

4. `DW_LNS_set_file`
   Takes a single unsigned LEB128 operand and stores it in the `file` register of the state machine.

5. `DW_LNS_set_column`
   Takes a single unsigned LEB128 operand and stores it in the `column` register of the state machine.

6. `DW_LNS_negate_stmt`
   Takes no arguments. Set the `is_stmt` register of the state machine to the logical negation of its current value.

7. `DW_LNS_set_basic_block`
   Takes no arguments. Set the `basic_block` register of the state machine to ‘‘true.’’

8. `DW_LNS_const_add_pc`
   Takes no arguments. Add to the `address` register of the state machine the address increment value corresponding to special opcode 255.

   *The motivation for* `DW_LNS_const_add_pc` *is this: when the statement program needs to advance the address by a small amount, it can use a single special opcode, which occupies a single byte. When it needs to advance the address by up to twice the range of the last special opcode, it can use* `DW_LNS_const_add_pc` *followed by a special opcode, for a total of two bytes. Only if it needs to advance the address by more than twice that range will it need to use both* `DW_LNS_advance_pc` *and a special opcode, requiring three or more bytes.*

9. `DW_LNS_fixed_advance_pc`
   Takes a single uhalf operand. Add to the `address` register of the state machine the value of the (unencoded) operand. This is the only extended opcode that takes an argument that is not a variable length number.

   *The motivation for* `DW_LNS_fixed_advance_pc` *is this: existing assemblers cannot emit* `DW_LNS_advance_pc` *or special opcodes because they cannot encode LEB128 numbers or judge when the computation of a special opcode overflows and requires the use of* `DW_LNS_advance_pc`*. Such assemblers, however, can use* `DW_LNS_fixed_advance_pc` *instead, sacrificing compression.*

### 6.2.5.3 Extended Opcodes

There are three extended opcodes currently defined. The first byte following the length field of the encoding for each contains a sub-opcode.

1. `DW_LNE_end_sequence`
   Set the `end_sequence` register of the state machine to ‘‘true’’ and append a row to the matrix using the current values of the state-machine registers. Then reset the registers to the initial values specified above.

   Every statement program sequence must end with a `DW_LNE_end_sequence` instruction which creates a row whose address is that of the byte after the last target machine instruction of the sequence.

2. `DW_LNE_set_address`
   Takes a single relocatable address as an operand. The size of the operand is the size appropriate to hold an address on the target machine. Set the `address` register to the value given by the relocatable address.

   *All of the other statement program opcodes that affect the* `address` *register add a delta to it. This instruction stores a relocatable value into it instead.*

3. `DW_LNE_define_file`
   Takes 4 arguments. The first is a null terminated string containing a source file name. The second is an unsigned LEB128 number representing the directory index of the directory in

which the file was found. The third is an unsigned LEB128 number representing the time of last modification of the file. The fourth is an unsigned LEB128 number representing the length in bytes of the file. The time and length fields may contain LEB128(0) if the information is not available.

The directory index represents an entry in the `include_directories` section of the statement program prologue. The index is LEB128(0) if the file was found in the current directory of the compilation, LEB128(1) if it was found in the first directory in the `include_directories` section, and so on. The directory index is ignored for file names that represent full path names.

The files are numbered, starting at 1, in the order in which they appear; the names in the prologue come before names defined by the `DW_LNE_define_file` instruction. These numbers are used in the the `file` register of the state machine.

*Appendix 3 gives some sample statement programs.*

## 6.3 Macro Information

*Some languages, such as C and C++, provide a way to replace text in the source program with macros defined either in the source file itself, or in another file included by the source file. Because these macros are not themselves defined in the target language, it is difficult to represent their definitions using the standard language constructs of DWARF. The debugging information therefore reflects the state of the source after the macro definition has been expanded, rather than as the programmer wrote it. The macro information table provides a way of preserving the original source in the debugging information.*

As described in section 3.1, the macro information for a given compilation unit is represented in the `.debug_macinfo` section of an object file. The macro information for each compilation unit is represented as a series of ‘‘macinfo’’ entries. Each macinfo entry consists of a ‘‘type code’’ and up to two additional operands. The series of entries for a given compilation unit ends with an entry containing a type code of 0.

### 6.3.1 Macinfo Types

The valid macinfo types are as follows:

| | |
|---|---|
| `DW_MACINFO_define` | A macro definition. |
| `DW_MACINFO_undef` | A macro un-definition. |
| `DW_MACINFO_start_file` | The start of a new source file inclusion. |
| `DW_MACINFO_end_file` | The end of the current source file inclusion. |
| `DW_MACINFO_vendor_ext` | Vendor specific macro information directives that do not fit into one of the standard categories. |

#### 6.3.1.1 Define and Undefine Entries

All `DW_MACINFO_define` and `DW_MACINFO_undef` entries have two operands. The first operand encodes the line number of the source line on which the relevant defining or undefining pre-processor directives appeared.

The second operand consists of a null-terminated character string. In the case of a `DW_MACINFO_undef` entry, the value of this string will be simply the name of the pre-processor symbol which was undefined at the indicated source line.

In the case of a DW_MACINFO_define entry, the value of this string will be the name of the pre-processor symbol that was defined at the indicated source line, followed immediately by the macro formal parameter list including the surrounding parentheses (in the case of a function-like macro) followed by the definition string for the macro. If there is no formal parameter list, then the name of the defined macro is followed directly by its definition string.

In the case of a function-like macro definition, no whitespace characters should appear between the name of the defined macro and the following left parenthesis. Also, no whitespace characters should appear between successive formal parameters in the formal parameter list. (Successive formal parameters should, however, be separated by commas.) Also, exactly one space character should separate the right parenthesis which terminates the formal parameter list and the following definition string.

In the case of a ‘‘normal’’ (i.e. non-function-like) macro definition, exactly one space character should separate the name of the defined macro from the following definition text.

#### 6.3.1.2 Start File Entries

Each DW_MACINFO_start_file entry also has two operands. The first operand encodes the line number of the source line on which the inclusion pre-processor directive occurred.

The second operand encodes a source file name index. This index corresponds to a file number in the statement information table for the relevant compilation unit. This index indicates (indirectly) the name of the file which is being included by the inclusion directive on the indicated source line.

#### 6.3.1.3 End File Entries

A DW_MACINFO_end_file entry has no operands. The presence of the entry marks the end of the current source file inclusion.

#### 6.3.1.4 Vendor Extension Entries

A DW_MACINFO_vendor_ext entry has two operands. The first is a constant. The second is a null-terminated character string. The meaning and/or significance of these operands is intentionally left undefined by this specification.

A consumer must be able to totally ignore all DW_MACINFO_vendor_ext entries that it does not understand.

### 6.3.2 Base Source Entries

In addition to producing a matched pair of DW_MACINFO_start_file and DW_MACINFO_end_file entries for each inclusion directive actually processed during compilation, a producer should generate such a matched pair also for the ‘‘base’’ source file submitted to the compiler for compilation. If the base source file for a compilation is submitted to the compiler via some means other than via a named disk file (e.g. via the standard input *stream* on a UNIX system) then the compiler should still produce this matched pair of DW_MACINFO_start_file and DW_MACINFO_end_file entries for the base source file, however, the file name indicated (indirectly) by the DW_MACINFO_start_file entry of the pair should reference a statement information file name entry consisting of a null string.

### 6.3.3 Macinfo Entries for Command Line Options

In addition to producing DW_MACINFO_define and DW_MACINFO_undef entries for each of the define and undefine directives processed during compilation, the DWARF producer should generate a DW_MACINFO_define or DW_MACINFO_undef entry for each pre-processor symbol which is defined or undefined by some means other than via a define or undefine directive within the compiled source text. In particular, pre-processor symbol definitions and un-definitions which occur as a result of command line options (when invoking the compiler) should be represented by their own DW_MACINFO_define and DW_MACINFO_undef entries.

All such DW_MACINFO_define and DW_MACINFO_undef entries representing compilation options should appear before the first DW_MACINFO_start_file entry for that compilation unit and should encode the value 0 in their line number operands.

### 6.3.4 General Rules and Restrictions

All macinfo entries within a .debug_macinfo section for a given compilation unit should appear in the same order in which the directives were processed by the compiler.

All macinfo entries representing command line options should appear in the same order as the relevant command line options were given to the compiler. In the case where the compiler itself implicitly supplies one or more macro definitions or un-definitions in addition to those which may be specified on the command line, macinfo entries should also be produced for these implicit definitions and un-definitions, and these entries should also appear in the proper order relative to each other and to any definitions or undefinitions given explicitly by the user on the command line.

## 6.4 Call Frame Information

*Debuggers often need to be able to view and modify the state of any subroutine activation that is on the call stack. An activation consists of:*

- *A code location that is within the subroutine. This location is either the place where the program stopped when the debugger got control (e.g. a breakpoint), or is a place where a subroutine made a call or was interrupted by an asynchronous event (e.g. a signal).*
- *An area of memory that is allocated on a stack called a ‘‘call frame.’’ The call frame is identified by an address on the stack. We refer to this address as the Canonical Frame Address or CFA.*
- *A set of registers that are in use by the subroutine at the code location.*

*Typically, a set of registers are designated to be preserved across a call. If a callee wishes to use such a register, it saves the value that the register had at entry time in its call frame and restores it on exit. The code that allocates space on the call frame stack and performs the save operation is called the subroutine’s prologue, and the code that performs the restore operation and deallocates the frame is called its epilogue. Typically, the prologue code is physically at the beginning of a subroutine and the epilogue code is at the end.*

*To be able to view or modify an activation that is not on the top of the call frame stack, the debugger must ‘‘virtually unwind’’ the stack of activations until it finds the activation of interest. A debugger unwinds a stack in steps. Starting with the current activation it restores any registers that were preserved by the current activation and computes the predecessor’s CFA and code location. This has the logical effect of returning from the current subroutine to its predecessor. We say that the debugger virtually unwinds the stack because it preserves enough information to be able to ‘‘rewind’’ the stack back to the state it was in before it attempted to unwind it.*

*The unwinding operation needs to know where registers are saved and how to compute the predecessor’s CFA and code location. When considering an architecture-independent way of* *encoding this information one has to consider a number of special things.*

- *Prologue and epilogue code is not always in distinct blocks at the beginning and end of a subroutine. It is common to duplicate the epilogue code at the site of each return from the code. Sometimes a compiler breaks up the register save/unsave operations and moves them into the body of the subroutine to just where they are needed.*
- *Compilers use different ways to manage the call frame. Sometimes they use a frame pointer register, sometimes not.*
- *The algorithm to compute the CFA changes as you progress through the prologue and epilogue code. (By definition, the CFA value does not change.)*
- *Some subroutines have no call frame.*
- *Sometimes a register is saved in another register that by convention does not need to be saved.*
- *Some architectures have special instructions that perform some or all of the register management in one instruction, leaving special information on the stack that indicates how registers are saved.*
- *Some architectures treat return address values specially. For example, in one architecture, the call instruction guarantees that the low order two bits will be zero and the return instruction ignores those bits. This leaves two bits of storage that are available to other uses that must be treated specially.*

### 6.4.1 Structure of Call Frame Information

DWARF supports virtual unwinding by defining an architecture independent basis for recording how procedures save and restore registers throughout their lifetimes. This basis must be augmented on some machines with specific information that is defined by either an architecture specific ABI authoring committee, a hardware vendor, or a compiler producer. The body defining a specific augmentation is referred to below as the ‘‘augmenter.’’

Abstractly, this mechanism describes a very large table that has the following structure:

| LOC | CFA | R0 | R1 | ... | RN |
|---|---|---|---|---|---|
| L0 | | | | | |
| L1 | | | | | |
| ... | | | | | |
| LN | | | | | |

The first column indicates an address for every location that contains code in a program. (In shared objects, this is an object-relative offset.) The remaining columns contain virtual unwinding rules that are associated with the indicated location. The first column of the rules defines the CFA rule which is a register and a signed offset that are added together to compute the CFA value.

The remaining columns are labeled by register number. This includes some registers that have special designation on some architectures such as the PC and the stack pointer register. (The actual mapping of registers for a particular architecture is performed by the augmenter.) The register columns contain rules that describe whether a given register has been saved and the rule to find the value for the register in the previous frame.

The register rules are:

undefined — A register that has this rule has no value in the previous frame. (By convention, it is not preserved by a callee.)

same value — This register has not been modified from the previous frame. (By convention, it is preserved by the callee, but the callee has not modified it.)

offset(N) — The previous value of this register is saved at the address CFA+N where CFA is the current CFA value and N is a signed offset.

register(R) — The previous value of this register is stored in another register numbered R.

architectural — The rule is defined externally to this specification by the augmenter.

*This table would be extremely large if actually constructed as described. Most of the entries at any point in the table are identical to the ones above them. The whole table can be represented quite compactly by recording just the differences starting at the beginning address of each subroutine in the program.*

The virtual unwind information is encoded in a self-contained section called `.debug_frame`. Entries in a `.debug_frame` section are aligned on an addressing unit boundary and come in two forms: A Common Information Entry (CIE) and a Frame Description Entry (FDE). Sizes of data objects used in the encoding of the `.debug_frame` section are described in terms of the same data definitions used for the line number information (see section 6.2.1).

A Common Information Entry holds information that is shared among many Frame Descriptors. There is at least one CIE in every non-empty `.debug_frame` section. A CIE contains the following fields, in order:

1. `length`
   A uword constant that gives the number of bytes of the CIE structure, not including the length field, itself (length mod <addressing unit size> == 0).
2. `CIE_id`
   A uword constant that is used to distinguish CIEs from FDEs.
3. `version`
   A ubyte version number. This number is specific to the call frame information and is independent of the DWARF version number.
4. `augmentation`
   A null terminated string that identifies the augmentation to this CIE or to the FDEs that use it. If a reader encounters an augmentation string that is unexpected, then only the following fields can be read: CIE: `length`, `CIE_id`, `version`, `augmentation`; FDE: `length`, `CIE_pointer`, `initial_location`, `address_range`. If there is no augmentation, this value is a zero byte.
5. `code_alignment_factor`
   An unsigned LEB128 constant that is factored out of all advance location instructions (see below).
6. `data_alignment_factor`
   A signed LEB128 constant that is factored out of all offset instructions (see below.)

7. return_address_register
   A ubyte constant that indicates which column in the rule table represents the return address of the function. Note that this column might not correspond to an actual machine register.

8. initial_instructions
   A sequence of rules that are interpreted to create the initial setting of each column in the table.

9. padding
   Enough DW_CFA_nop instructions to make the size of this entry match the length value above.

An FDE contains the following fields, in order:

1. length
   A uword constant that gives the number of bytes of the header and instruction stream for this function (not including the length field itself) (length mod <addressing unit size> == 0).

2. CIE_pointer
   A uword constant offset into the .debug_frame section that denotes the CIE that is associated with this FDE.

3. initial_location An addressing-unit sized constant indicating the address of the first location associated with this table entry.

4. address_range
   An addressing unit sized constant indicating the number of bytes of program instructions described by this entry.

5. instructions
   A sequence of table defining instructions that are described below.

### 6.4.2 Call Frame Instructions

Each call frame instruction is defined to take 0 or more operands. Some of the operands may be encoded as part of the opcode (see section 7.23). The instructions are as follows:

1. DW_CFA_advance_loc takes a single argument that represents a constant delta. The required action is to create a new table row with a location value that is computed by taking the current entry's location value and adding (delta * code_alignment_factor). All other values in the new row are initially identical to the current row.

2. DW_CFA_offset takes two arguments: an unsigned LEB128 constant representing a factored offset and a register number. The required action is to change the rule for the register indicated by the register number to be an offset(N) rule with a value of (N = factored offset * data_alignment_factor).

3. DW_CFA_restore takes a single argument that represents a register number. The required action is to change the rule for the indicated register to the rule assigned it by the initial_instructions in the CIE.

4. DW_CFA_set_loc takes a single argument that represents an address. The required action is to create a new table row using the specified address as the location. All other values in the new row are initially identical to the current row. The new location value should always be greater than the current one.

5. DW_CFA_advance_loc1 takes a single ubyte argument that represents a constant delta. This instruction is identical to DW_CFA_advance_loc except for the encoding and size of the delta argument.

6. DW_CFA_advance_loc2 takes a single uhalf argument that represents a constant delta. This instruction is identical to DW_CFA_advance_loc except for the encoding and size of the delta argument.

7. DW_CFA_advance_loc4 takes a single uword argument that represents a constant delta. This instruction is identical to DW_CFA_advance_loc except for the encoding and size of the delta argument.

8. DW_CFA_offset_extended takes two unsigned LEB128 arguments representing a register number and a factored offset. This instruction is identical to DW_CFA_offset except for the encoding and size of the register argument.

9. DW_CFA_restore_extended takes a single unsigned LEB128 argument that represents a register number. This instruction is identical to DW_CFA_restore except for the encoding and size of the register argument.

10. DW_CFA_undefined takes a single unsigned LEB128 argument that represents a register number. The required action is to set the rule for the specified register to ‘‘undefined.’’

11. DW_CFA_same_value takes a single unsigned LEB128 argument that represents a register number. The required action is to set the rule for the specified register to ‘‘same value.’’

12. DW_CFA_register takes two unsigned LEB128 arguments representing register numbers. The required action is to set the rule for the first register to be the second register.

13. DW_CFA_remember_state

14. DW_CFA_restore_state
    These instructions define a stack of information. Encountering the DW_CFA_remember_state instruction means to save the rules for every register on the current row on the stack. Encountering the DW_CFA_restore_state instruction means to pop the set of rules off the stack and place them in the current row. *(This operation is useful for compilers that move epilogue code into the body of a function.)*

15. DW_CFA_def_cfa takes two unsigned LEB128 arguments representing a register number and an offset. The required action is to define the current CFA rule to use the provided register and offset.

16. DW_CFA_def_cfa_register takes a single unsigned LEB128 argument representing a register number. The required action is to define the current CFA rule to use the provided register (but to keep the old offset).

17. DW_CFA_def_cfa_offset takes a single unsigned LEB128 argument representing an offset. The required action is to define the current CFA rule to use the provided offset (but to keep the old register).

18. DW_CFA_nop has no arguments and no required actions. It is used as padding to make the FDE an appropriate size.

### 6.4.3 Call Frame Instruction Usage

*To determine the virtual unwind rule set for a given location (L1), one searches through the FDE headers looking at the* `initial_location` *and* `address_range` *values to see if L1 is contained in the FDE. If so, then:*

1. *Initialize a register set by reading the* `initial_instructions` *field of the associated CIE.*
2. *Read and process the FDE's instruction sequence until a* `DW_CFA_advance_loc`, `DW_CFA_set_loc`*, or the end of the instruction stream is encountered.*
3. *If a* `DW_CFA_advance_loc` *or* `DW_CFA_set_loc` *instruction was encountered, then compute a new location value (L2). If L1 >= L2 then process the instruction and go back to step 2.*
4. *The end of the instruction stream can be thought of as a*
   `DW_CFA_set_loc( initial_location + address_range )`
   *instruction. Unless the FDE is ill-formed, L1 should be less than L2 at this point.*

*The rules in the register set now apply to location L1.*

*For an example, see Appendix 5.*

# 7. DATA REPRESENTATION

This section describes the binary representation of the debugging information entry itself, of the attribute types and of other fundamental elements described above.

## 7.1 Vendor Extensibility

To reserve a portion of the DWARF name space and ranges of enumeration values for use for vendor specific extensions, special labels are reserved for tag names, attribute names, base type encodings, location operations, language names, calling conventions and call frame instructions. The labels denoting the beginning and end of the reserved value range for vendor specific extensions consist of the appropriate prefix ( `DW_TAG`, `DW_AT`, `DW_ATE`, `DW_OP`, `DW_LANG`, or `DW_CFA` respectively) followed by `_lo_user` or `_hi_user`. For example, for entry tags, the special labels are `DW_TAG_lo_user` and `DW_TAG_hi_user`. Values in the range between *prefix*`_lo_user` and *prefix*`_hi_user` inclusive, are reserved for vendor specific extensions. Vendors may use values in this range without conflicting with current or future system-defined values. All other values are reserved for use by the system.

Vendor defined tags, attributes, base type encodings, location atoms, language names, calling conventions and call frame instructions, conventionally use the form *prefix_vendor_id_name*, where *vendor_id* is some identifying character sequence chosen so as to avoid conflicts with other vendors.

To ensure that extensions added by one vendor may be safely ignored by consumers that do not understand those extensions, the following rules should be followed:

1. New attributes should be added in such a way that a debugger may recognize the format of a new attribute value without knowing the content of that attribute value.
2. The semantics of any new attributes should not alter the semantics of previously existing attributes.
3. The semantics of any new tags should not conflict with the semantics of previously existing tags.

## 7.2 Reserved Error Values

As a convenience for consumers of DWARF information, the value 0 is reserved in the encodings for attribute names, attribute forms, base type encodings, location operations, languages, statement program opcodes, macro information entries and tag names to represent an error condition or unknown value. DWARF does not specify names for these reserved values, since they do not represent valid encodings for the given type and should not appear in DWARF debugging information.

## 7.3 Executable Objects and Shared Objects

The relocated addresses in the debugging information for an executable object are virtual addresses and the relocated addresses in the debugging information for a shared object are offsets relative to the start of the lowest segment used by that shared object.

*This requirement makes the debugging information for shared objects position independent. Virtual addresses in a shared object may be calculated by adding the offset to the base address at which the object was attached. This offset is available in the run-time linker's data structures.*

## 7.4 File Constraints

All debugging information entries in a relocatable object file, executable object or shared object are required to be physically contiguous.

## 7.5 Format of Debugging Information

For each compilation unit compiled with a DWARF Version 2 producer, a contribution is made to the `.debug_info` section of the object file. Each such contribution consists of a compilation unit header followed by a series of debugging information entries. Unlike the information encoding for DWARF Version 1, Version 2 debugging information entries do not themselves contain the debugging information entry tag or the attribute name and form encodings for each attribute. Instead, each debugging information entry begins with a code that represents an entry in a separate abbreviations table. This code is followed directly by a series of attribute values. The appropriate entry in the abbreviations table guides the interpretation of the information contained directly in the `.debug_info` section. Each compilation unit is associated with a particular abbreviation table, but multiple compilation units may share the same table.

*This encoding was based on the observation that typical DWARF producers produce a very limited number of different types of debugging information entries. By extracting the common information from those entries into a separate table, we are able to compress the generated information.*

### 7.5.1 Compilation Unit Header

The header for the series of debugging information entries contributed by a single compilation unit consists of the following information:

1. A 4-byte unsigned integer representing the length of the `.debug_info` contribution for that compilation unit, not including the length field itself.

2. A 2-byte unsigned integer representing the version of the DWARF information for that compilation unit. For DWARF Version 2, the value in this field is 2.

3. A 4-byte unsigned offset into the `.debug_abbrev` section. This offset associates the compilation unit with a particular set of debugging information entry abbreviations.

4. A 1-byte unsigned integer representing the size in bytes of an address on the target architecture. If the system uses segmented addressing, this value represents the size of the offset portion of an address.

*The compilation unit header does not replace the* `DW_TAG_compile_unit` *debugging information entry. It is additional information that is represented outside the standard DWARF tag/attributes format.*

### 7.5.2 Debugging Information Entry

Each debugging information entry begins with an unsigned LEB128 number containing the abbreviation code for the entry. This code represents an entry within the abbreviation table associated with the compilation unit containing this entry. The abbreviation code is followed by a series of attribute values.

On some architectures, there are alignment constraints on section boundaries. To make it easier to pad debugging information sections to satisfy such constraints, the abbreviation code 0 is reserved. Debugging information entries consisting of only the 0 abbreviation code are considered null entries.

### 7.5.3 Abbreviation Tables

The abbreviation tables for all compilation units are contained in a separate object file section called `.debug_abbrev`. As mentioned before, multiple compilation units may share the same abbreviation table.

The abbreviation table for a single compilation unit consists of a series of abbreviation declarations. Each declaration specifies the tag and attributes for a particular form of debugging information entry. Each declaration begins with an unsigned LEB128 number representing the abbreviation code itself. It is this code that appears at the beginning of a debugging information entry in the `.debug_info` section. As described above, the abbreviation code 0 is reserved for null debugging information entries. The abbreviation code is followed by another unsigned LEB128 number that encodes the entry's tag. The encodings for the tag names are given in Figures 14 and 15.

Following the tag encoding is a 1-byte value that determines whether a debugging information entry using this abbreviation has child entries or not. If the value is `DW_CHILDREN_yes`, the next physically succeeding entry of any debugging information entry using this abbreviation is the first child of the prior entry. If the 1-byte value following the abbreviation's tag encoding is `DW_CHILDREN_no`, the next physically succeeding entry of any debugging information entry using this abbreviation is a sibling of the prior entry. (Either the first child or sibling entries may be null entries). The encodings for the child determination byte are given in Figure 16. (As mentioned in section 2.3, each chain of sibling entries is terminated by a null entry).

Finally, the child encoding is followed by a series of attribute specifications. Each attribute specification consists of two parts. The first part is an unsigned LEB128 number representing the attribute's name. The second part is an unsigned LEB128 number representing the attribute's form. The series of attribute specifications ends with an entry containing 0 for the name and 0 for the form.

The attribute form `DW_FORM_indirect` is a special case. For attributes with this form, the attribute value itself in the `.debug_info` section begins with an unsigned LEB128 number that represents its form. This allows producers to choose forms for particular attributes dynamically, without having to add a new entry to the abbreviation table.

The abbreviations for a given compilation unit end with an entry consisting of a 0 byte for the abbreviation code.

*See Appendix 2 for a depiction of the organization of the debugging information.*

### 7.5.4 Attribute Encodings

The encodings for the attribute names are given in Figures 17 and 18.

The attribute form governs how the value of the attribute is encoded. The possible forms may belong to one of the following form classes:

address
: Represented as an object of appropriate size to hold an address on the target machine (`DW_FORM_addr`). This address is relocatable in a relocatable object file and is relocated in an executable file or shared object.

block
: Blocks come in four forms. The first consists of a 1-byte length followed by 0 to 255 contiguous information bytes (`DW_FORM_block1`). The second consists of a 2-byte length followed by 0 to 65,535 contiguous information bytes (`DW_FORM_block2`). The third consists of a 4-byte

| Tag name | Value |
|---|---|
| DW_TAG_array_type | 0x01 |
| DW_TAG_class_type | 0x02 |
| DW_TAG_entry_point | 0x03 |
| DW_TAG_enumeration_type | 0x04 |
| DW_TAG_formal_parameter | 0x05 |
| DW_TAG_imported_declaration | 0x08 |
| DW_TAG_label | 0x0a |
| DW_TAG_lexical_block | 0x0b |
| DW_TAG_member | 0x0d |
| DW_TAG_pointer_type | 0x0f |
| DW_TAG_reference_type | 0x10 |
| DW_TAG_compile_unit | 0x11 |
| DW_TAG_string_type | 0x12 |
| DW_TAG_structure_type | 0x13 |
| DW_TAG_subroutine_type | 0x15 |
| DW_TAG_typedef | 0x16 |
| DW_TAG_union_type | 0x17 |
| DW_TAG_unspecified_parameters | 0x18 |
| DW_TAG_variant | 0x19 |
| DW_TAG_common_block | 0x1a |
| DW_TAG_common_inclusion | 0x1b |
| DW_TAG_inheritance | 0x1c |
| DW_TAG_inlined_subroutine | 0x1d |
| DW_TAG_module | 0x1e |
| DW_TAG_ptr_to_member_type | 0x1f |
| DW_TAG_set_type | 0x20 |
| DW_TAG_subrange_type | 0x21 |
| DW_TAG_with_stmt | 0x22 |
| DW_TAG_access_declaration | 0x23 |
| DW_TAG_base_type | 0x24 |
| DW_TAG_catch_block | 0x25 |
| DW_TAG_const_type | 0x26 |
| DW_TAG_constant | 0x27 |
| DW_TAG_enumerator | 0x28 |
| DW_TAG_file_type | 0x29 |

**Figure 14.** Tag encodings (part 1)

length followed by 0 to 4,294,967,295 contiguous information bytes (DW_FORM_block4). The fourth consists of an unsigned LEB128 length followed by the number of bytes specified by the length (DW_FORM_block). In all forms, the length is the number of information bytes that follow. The information bytes may contain any mixture of relocated (or relocatable) addresses, references to other debugging information entries or data bytes.

constant There are six forms of constants: one, two, four and eight byte values (respectively, DW_FORM_data1, DW_FORM_data2, DW_FORM_data4, and DW_FORM_data8). There are also variable

| Tag name | Value |
|---|---|
| DW_TAG_friend | 0x2a |
| DW_TAG_namelist | 0x2b |
| DW_TAG_namelist_item | 0x2c |
| DW_TAG_packed_type | 0x2d |
| DW_TAG_subprogram | 0x2e |
| DW_TAG_template_type_param | 0x2f |
| DW_TAG_template_value_param | 0x30 |
| DW_TAG_thrown_type | 0x31 |
| DW_TAG_try_block | 0x32 |
| DW_TAG_variant_part | 0x33 |
| DW_TAG_variable | 0x34 |
| DW_TAG_volatile_type | 0x35 |
| DW_TAG_lo_user | 0x4080 |
| DW_TAG_hi_user | 0xffff |

**Figure 15.** Tag encodings (part 2)

| Child determination name | Value |
|---|---|
| DW_CHILDREN_no | 0 |
| DW_CHILDREN_yes | 1 |

**Figure 16.** Child determination encodings

length constant data forms encoded using LEB128 numbers (see below). Both signed (`DW_FORM_sdata`) and unsigned (`DW_FORM_udata`) variable length constants are available.

flag

A flag is represented as a single byte of data (`DW_FORM_flag`). If the flag has value zero, it indicates the absence of the attribute. If the flag has a non-zero value, it indicates the presence of the attribute.

reference

There are two types of reference. The first is an offset relative to the first byte of the compilation unit header for the compilation unit containing the reference. The offset must refer to an entry within that same compilation unit. There are five forms for this type of reference: one, two, four and eight byte offsets (respectively, `DW_FORM_ref1`, `DW_FORM_ref2`, `DW_FORM_ref4`, and `DW_FORM_ref8`). There are is also an unsigned variable length offset encoded using LEB128 numbers (`DW_FORM_ref_udata`).

The second type of reference is the address of any debugging information entry within the same executable or shared object; it may refer to an entry in a different compilation unit from the unit containing the reference. This type of reference (`DW_FORM_ref_addr`) is the size of an address on the target architecture; it is relocatable in a relocatable object file and relocated in an executable file or shared object.

*The use of compilation unit relative references will reduce the number of link-time relocations and so speed up linking.*

*The use of address-type references allows for the commonization of information, such as types, across compilation units.*

| Attribute name | Value | Classes |
|---|---|---|
| `DW_AT_sibling` | `0x01` | reference |
| `DW_AT_location` | `0x02` | block, constant |
| `DW_AT_name` | `0x03` | string |
| `DW_AT_ordering` | `0x09` | constant |
| `DW_AT_byte_size` | `0x0b` | constant |
| `DW_AT_bit_offset` | `0x0c` | constant |
| `DW_AT_bit_size` | `0x0d` | constant |
| `DW_AT_stmt_list` | `0x10` | constant |
| `DW_AT_low_pc` | `0x11` | address |
| `DW_AT_high_pc` | `0x12` | address |
| `DW_AT_language` | `0x13` | constant |
| `DW_AT_discr` | `0x15` | reference |
| `DW_AT_discr_value` | `0x16` | constant |
| `DW_AT_visibility` | `0x17` | constant |
| `DW_AT_import` | `0x18` | reference |
| `DW_AT_string_length` | `0x19` | block, constant |
| `DW_AT_common_reference` | `0x1a` | reference |
| `DW_AT_comp_dir` | `0x1b` | string |
| `DW_AT_const_value` | `0x1c` | string, constant, block |
| `DW_AT_containing_type` | `0x1d` | reference |
| `DW_AT_default_value` | `0x1e` | reference |
| `DW_AT_inline` | `0x20` | constant |
| `DW_AT_is_optional` | `0x21` | flag |
| `DW_AT_lower_bound` | `0x22` | constant, reference |
| `DW_AT_producer` | `0x25` | string |
| `DW_AT_prototyped` | `0x27` | flag |
| `DW_AT_return_addr` | `0x2a` | block, constant |
| `DW_AT_start_scope` | `0x2c` | constant |
| `DW_AT_stride_size` | `0x2e` | constant |
| `DW_AT_upper_bound` | `0x2f` | constant, reference |

**Figure 17.** Attribute encodings, part 1

string — A string is a sequence of contiguous non-null bytes followed by one null byte. A string may be represented immediately in the debugging information entry itself (`DW_FORM_string`), or may be represented as a 4-byte offset into a string table contained in the `.debug_str` section of the object file (`DW_FORM_strp`).

The form encodings are listed in Figure 19.

## 7.6 Variable Length Data

The special constant data forms `DW_FORM_sdata` and `DW_FORM_udata` are encoded using ‘‘Little Endian Base 128’’ (LEB128) numbers. LEB128 is a scheme for encoding integers densely that exploits the assumption that most integers are small in magnitude. (This encoding is equally suitable whether the target machine architecture represents data in big-endian or little-endian order. It is ‘‘little endian’’ only in the sense that it avoids using space to represent the ‘‘big’’ end of an unsigned integer, when the big end is all zeroes or sign extension bits).

| Attribute name | Value | Classes |
|---|---|---|
| DW_AT_abstract_origin | 0x31 | reference |
| DW_AT_accessibility | 0x32 | constant |
| DW_AT_address_class | 0x33 | constant |
| DW_AT_artificial | 0x34 | flag |
| DW_AT_base_types | 0x35 | reference |
| DW_AT_calling_convention | 0x36 | constant |
| DW_AT_count | 0x37 | constant, reference |
| DW_AT_data_member_location | 0x38 | block, reference |
| DW_AT_decl_column | 0x39 | constant |
| DW_AT_decl_file | 0x3a | constant |
| DW_AT_decl_line | 0x3b | constant |
| DW_AT_declaration | 0x3c | flag |
| DW_AT_discr_list | 0x3d | block |
| DW_AT_encoding | 0x3e | constant |
| DW_AT_external | 0x3f | flag |
| DW_AT_frame_base | 0x40 | block, constant |
| DW_AT_friend | 0x41 | reference |
| DW_AT_identifier_case | 0x42 | constant |
| DW_AT_macro_info | 0x43 | constant |
| DW_AT_namelist_item | 0x44 | block |
| DW_AT_priority | 0x45 | reference |
| DW_AT_segment | 0x46 | block, constant |
| DW_AT_specification | 0x47 | reference |
| DW_AT_static_link | 0x48 | block, constant |
| DW_AT_type | 0x49 | reference |
| DW_AT_use_location | 0x4a | block, constant |
| DW_AT_variable_parameter | 0x4b | flag |
| DW_AT_virtuality | 0x4c | constant |
| DW_AT_vtable_elem_location | 0x4d | block, reference |
| DW_AT_lo_user | 0x2000 | — |
| DW_AT_hi_user | 0x3fff | — |

**Figure 18.** Attribute encodings, part 2

DW_FORM_udata (unsigned LEB128) numbers are encoded as follows: start at the low order end of an unsigned integer and chop it into 7-bit chunks. Place each chunk into the low order 7 bits of a byte. Typically, several of the high order bytes will be zero; discard them. Emit the remaining bytes in a stream, starting with the low order byte; set the high order bit on each byte except the last emitted byte. The high bit of zero on the last byte indicates to the decoder that it has encountered the last byte.

The integer zero is a special case, consisting of a single zero byte.

*Figure 20 gives some examples of* DW_FORM_udata *numbers. The* 0x80 *in each case is the high order bit of the byte, indicating that an additional byte follows:*

The encoding for DW_FORM_sdata (signed, 2s complement LEB128) numbers is similar, except that the criterion for discarding high order bytes is not whether they are zero, but whether they consist entirely of sign extension bits. Consider the 32-bit integer -2. The three high level bytes of the number are sign extension, thus LEB128 would represent it as a single byte

| Form name | Value | Class |
|---|---|---|
| DW_FORM_addr | 0x01 | address |
| DW_FORM_block2 | 0x03 | block |
| DW_FORM_block4 | 0x04 | block |
| DW_FORM_data2 | 0x05 | constant |
| DW_FORM_data4 | 0x06 | constant |
| DW_FORM_data8 | 0x07 | constant |
| DW_FORM_string | 0x08 | string |
| DW_FORM_block | 0x09 | block |
| DW_FORM_block1 | 0x0a | block |
| DW_FORM_data1 | 0x0b | constant |
| DW_FORM_flag | 0x0c | flag |
| DW_FORM_sdata | 0x0d | constant |
| DW_FORM_strp | 0x0e | string |
| DW_FORM_udata | 0x0f | constant |
| DW_FORM_ref_addr | 0x10 | reference |
| DW_FORM_ref1 | 0x11 | reference |
| DW_FORM_ref2 | 0x12 | reference |
| DW_FORM_ref4 | 0x13 | reference |
| DW_FORM_ref8 | 0x14 | reference |
| DW_FORM_ref_udata | 0x15 | reference |
| DW_FORM_indirect | 0x16 | (see section 7.5.3) |

**Figure 19.** Attribute form encodings

| Number | First byte | Second byte |
|---|---|---|
| 2 | 2 | — |
| 127 | 127 | — |
| 128 | 0+0x80 | 1 |
| 129 | 1+0x80 | 1 |
| 130 | 2+0x80 | 1 |
| 12857 | 57+0x80 | 100 |

**Figure 20.** Examples of unsigned LEB128 encodings

containing the low order 7 bits, with the high order bit cleared to indicate the end of the byte stream. Note that there is nothing within the LEB128 representation that indicates whether an encoded number is signed or unsigned. The decoder must know what type of number to expect.

*Figure 21 gives some examples of* `DW_FORM_sdata` *numbers.*

*Appendix 4 gives algorithms for encoding and decoding these forms.*

## 7.7 Location Descriptions

### 7.7.1 Location Expressions

A location expression is stored in a block of contiguous bytes. The bytes form a set of operations. Each location operation has a 1-byte code that identifies that operation. Operations can be followed by one or more bytes of additional data. All operations in a location expression are concatenated from left to right. The encodings for the operations in a location expression are described in Figures 22 and 23.

| Number | First byte | Second byte |
|---|---|---|
| 2 | 2 | — |
| -2 | 0x7e | — |
| 127 | 127+0x80 | 0 |
| -127 | 1+0x80 | 0x7f |
| 128 | 0+0x80 | 1 |
| -128 | 0+0x80 | 0x7f |
| 129 | 1+0x80 | 1 |
| -129 | 0x7f+0x80 | 0x7e |

**Figure 21.** Examples of signed LEB128 encodings

| Operation | Code | No. of Operands | Notes |
|---|---|---|---|
| `DW_OP_addr` | `0x03` | 1 | constant address (size target specific) |
| `DW_OP_deref` | `0x06` | 0 | |
| `DW_OP_const1u` | `0x08` | 1 | 1-byte constant |
| `DW_OP_const1s` | `0x09` | 1 | 1-byte constant |
| `DW_OP_const2u` | `0x0a` | 1 | 2-byte constant |
| `DW_OP_const2s` | `0x0b` | 1 | 2-byte constant |
| `DW_OP_const4u` | `0x0c` | 1 | 4-byte constant |
| `DW_OP_const4s` | `0x0d` | 1 | 4-byte constant |
| `DW_OP_const8u` | `0x0e` | 1 | 8-byte constant |
| `DW_OP_const8s` | `0x0f` | 1 | 8-byte constant |
| `DW_OP_constu` | `0x10` | 1 | ULEB128 constant |
| `DW_OP_consts` | `0x11` | 1 | SLEB128 constant |
| `DW_OP_dup` | `0x12` | 0 | |
| `DW_OP_drop` | `0x13` | 0 | |
| `DW_OP_over` | `0x14` | 0 | |
| `DW_OP_pick` | `0x15` | 1 | 1-byte stack index |
| `DW_OP_swap` | `0x16` | 0 | |
| `DW_OP_rot` | `0x17` | 0 | |
| `DW_OP_xderef` | `0x18` | 0 | |
| `DW_OP_abs` | `0x19` | 0 | |
| `DW_OP_and` | `0x1a` | 0 | |
| `DW_OP_div` | `0x1b` | 0 | |
| `DW_OP_minus` | `0x1c` | 0 | |
| `DW_OP_mod` | `0x1d` | 0 | |
| `DW_OP_mul` | `0x1e` | 0 | |
| `DW_OP_neg` | `0x1f` | 0 | |
| `DW_OP_not` | `0x20` | 0 | |
| `DW_OP_or` | `0x21` | 0 | |
| `DW_OP_plus` | `0x22` | 0 | |
| `DW_OP_plus_uconst` | `0x23` | 1 | ULEB128 addend |
| `DW_OP_shl` | `0x24` | 0 | |
| `DW_OP_shr` | `0x25` | 0 | |
| `DW_OP_shra` | `0x26` | 0 | |

**Figure 22.** Location operation encodings, part 1

| Operation | Code | No. of Operands | Notes |
|---|---|---|---|
| DW_OP_xor | 0x27 | 0 | |
| DW_OP_skip | 0x2f | 1 | signed 2-byte constant |
| DW_OP_bra | 0x28 | 1 | signed 2-byte constant |
| DW_OP_eq | 0x29 | 0 | |
| DW_OP_ge | 0x2a | 0 | |
| DW_OP_gt | 0x2b | 0 | |
| DW_OP_le | 0x2c | 0 | |
| DW_OP_lt | 0x2d | 0 | |
| DW_OP_ne | 0x2e | 0 | |
| DW_OP_lit0 | 0x30 | 0 | literals 0..31 = (DW_OP_LIT0\|literal) |
| DW_OP_lit1 | 0x31 | 0 | |
| ... | | | |
| DW_OP_lit31 | 0x4f | 0 | |
| DW_OP_reg0 | 0x50 | 0 | reg 0..31 = (DW_OP_REG0\|regnum) |
| DW_OP_reg1 | 0x51 | 0 | |
| ... | | | |
| DW_OP_reg31 | 0x6f | 0 | |
| DW_OP_breg0 | 0x70 | 1 | SLEB128 offset |
| DW_OP_breg1 | 0x71 | 1 | base reg 0..31 = (DW_OP_BREG0\|regnum) |
| ... | | | |
| DW_OP_breg31 | 0x8f | 1 | |
| DW_OP_regx | 0x90 | 1 | ULEB128 register |
| DW_OP_fbreg | 0x91 | 1 | SLEB128 offset |
| DW_OP_bregx | 0x92 | 2 | ULEB128 register followed by SLEB128 offset |
| DW_OP_piece | 0x93 | 1 | ULEB128 size of piece addressed |
| DW_OP_deref_size | 0x94 | 1 | 1-byte size of data retrieved |
| DW_OP_xderef_size | 0x95 | 1 | 1-byte size of data retrieved |
| DW_OP_nop | 0x96 | 0 | |
| DW_OP_lo_user | 0xe0 | | |
| DW_OP_hi_user | 0xff | | |

**Figure 23.** Location operation encodings, part 2

### 7.7.2 Location Lists

Each entry in a location list consists of two relative addresses followed by a 2-byte length, followed by a block of contiguous bytes. The length specifies the number of bytes in the block that follows. The two addresses are the same size as used by DW_FORM_addr on the target machine.

## 7.8 Base Type Encodings

The values of the constants used in the DW_AT_encoding attribute are given in Figure 24.

## 7.9 Accessibility Codes

The encodings of the constants used in the DW_AT_accessibility attribute are given in Figure 25.

| Base type encoding name | Value |
|---|---|
| DW_ATE_address | 0x1 |
| DW_ATE_boolean | 0x2 |
| DW_ATE_complex_float | 0x3 |
| DW_ATE_float | 0x4 |
| DW_ATE_signed | 0x5 |
| DW_ATE_signed_char | 0x6 |
| DW_ATE_unsigned | 0x7 |
| DW_ATE_unsigned_char | 0x8 |
| DW_ATE_lo_user | 0x80 |
| DW_ATE_hi_user | 0xff |

**Figure 24.** Base type encoding values

| Accessibility code name | Value |
|---|---|
| DW_ACCESS_public | 1 |
| DW_ACCESS_protected | 2 |
| DW_ACCESS_private | 3 |

**Figure 25.** Accessibility encodings

## 7.10 Visibility Codes

The encodings of the constants used in the DW_AT_visibility attribute are given in Figure 26.

| Visibility code name | Value |
|---|---|
| DW_VIS_local | 1 |
| DW_VIS_exported | 2 |
| DW_VIS_qualified | 3 |

**Figure 26.** Visibility encodings

## 7.11 Virtuality Codes

The encodings of the constants used in the DW_AT_virtuality attribute are given in Figure 27.

| Virtuality code name | Value |
|---|---|
| DW_VIRTUALITY_none | 0 |
| DW_VIRTUALITY_virtual | 1 |
| DW_VIRTUALITY_pure_virtual | 2 |

**Figure 27.** Virtuality encodings

## 7.12 Source Languages

The encodings for source languages are given in Figure 28. Names marked with † and their associated values are reserved, but the languages they represent are not supported in DWARF Version 2.

## 7.13 Address Class Encodings

The value of the common address class encoding DW_ADDR_none is 0.

| Language name | Value |
|---|---|
| DW_LANG_C89 | 0x0001 |
| DW_LANG_C | 0x0002 |
| DW_LANG_Ada83† | 0x0003 |
| DW_LANG_C_plus_plus | 0x0004 |
| DW_LANG_Cobol74† | 0x0005 |
| DW_LANG_Cobol85† | 0x0006 |
| DW_LANG_Fortran77 | 0x0007 |
| DW_LANG_Fortran90 | 0x0008 |
| DW_LANG_Pascal83 | 0x0009 |
| DW_LANG_Modula2 | 0x000a |
| DW_LANG_lo_user | 0x8000 |
| DW_LANG_hi_user | 0xffff |

**Figure 28.** Language encodings

## 7.14 Identifier Case

The encodings of the constants used in the DW_AT_identifier_case attribute are given in Figure 29.

| Identifier Case Name | Value |
|---|---|
| DW_ID_case_sensitive | 0 |
| DW_ID_up_case | 1 |
| DW_ID_down_case | 2 |
| DW_ID_case_insensitive | 3 |

**Figure 29.** Identifier case encodings

## 7.15 Calling Convention Encodings

The encodings for the values of the DW_AT_calling_convention attribute are given in Figure 30.

| Calling Convention Name | Value |
|---|---|
| DW_CC_normal | 0x1 |
| DW_CC_program | 0x2 |
| DW_CC_nocall | 0x3 |
| DW_CC_lo_user | 0x40 |
| DW_CC_hi_user | 0xff |

**Figure 30.** Calling convention encodings

## 7.16 Inline Codes

The encodings of the constants used in the DW_AT_inline attribute are given in Figure 31.

| Inline Code Name | Value |
|---|---|
| DW_INL_not_inlined | 0 |
| DW_INL_inlined | 1 |
| DW_INL_declared_not_inlined | 2 |
| DW_INL_declared_inlined | 3 |

**Figure 31.** Inline encodings

## 7.17 Array Ordering

The encodings for the values of the order attributes of arrays is given in Figure 32.

| Ordering name | Value |
|---|---|
| DW_ORD_row_major | 0 |
| DW_ORD_col_major | 1 |

**Figure 32.** Ordering encodings

## 7.18 Discriminant Lists

The descriptors used in the DW_AT_dicsr_list attribute are encoded as 1-byte constants. The defined values are presented in Figure 33.

| Descriptor Name | Value |
|---|---|
| DW_DSC_label | 0 |
| DW_DSC_range | 1 |

**Figure 33.** Discriminant descriptor encodings

## 7.19 Name Lookup Table

Each set of entries in the table of global names contained in the .debug_pubnames section begins with a header consisting of: a 4-byte length containing the length of the set of entries for this compilation unit, not including the length field itself; a 2-byte version identifier containing the value 2 for DWARF Version 2; a 4-byte offset into the .debug_info section; and a 4-byte length containing the size in bytes of the contents of the .debug_info section generated to represent this compilation unit. This header is followed by a series of tuples. Each tuple consists of a 4-byte offset followed by a string of non-null bytes terminated by one null byte. Each set is terminated by a 4-byte word containing the value 0.

## 7.20 Address Range Table

Each set of entries in the table of address ranges contained in the .debug_aranges section begins with a header consisting of: a 4-byte length containing the length of the set of entries for this compilation unit, not including the length field itself; a 2-byte version identifier containing the value 2 for DWARF Version 2; a 4-byte offset into the .debug_info section; a 1-byte unsigned integer containing the size in bytes of an address (or the offset portion of an address for segmented addressing) on the target system; and a 1-byte unsigned integer containing the size in bytes of a segment descriptor on the target system. This header is followed by a series of tuples. Each tuple consists of an address and a length, each in the size appropriate for an address on the target architecture. The first tuple following the header in each set begins at an offset that is a multiple of the size of a single tuple (that is, twice the size of an address). The header is padded, if necessary, to the appropriate boundary. Each set of tuples is terminated by a 0 for the address and 0 for the length.

## 7.21 Line Number Information

The sizes of the integers used in the line number and call frame information sections are as follows:

sbyte Signed 1-byte value.

ubyte Unsigned 1-byte value.

|  |  |
|---|---|
| uhalf | Unsigned 2-byte value. |
| sword | Signed 4-byte value. |
| uword | Unsigned 4-byte value. |

The version number in the statement program prologue is 2 for DWARF Version 2. The boolean values ‘‘true’’ and ‘‘false’’ used by the statement information program are encoded as a single byte containing the value 0 for ‘‘false,’’ and a non-zero value for ‘‘true.’’ The encodings for the pre-defined standard opcodes are given in Figure 34.

| Opcode Name | Value |
|---|---|
| DW_LNS_copy | 1 |
| DW_LNS_advance_pc | 2 |
| DW_LNS_advance_line | 3 |
| DW_LNS_set_file | 4 |
| DW_LNS_set_column | 5 |
| DW_LNS_negate_stmt | 6 |
| DW_LNS_set_basic_block | 7 |
| DW_LNS_const_add_pc | 8 |
| DW_LNS_fixed_advance_pc | 9 |

**Figure 34.** Standard Opcode Encodings

The encodings for the pre-defined extended opcodes are given in Figure 35.

| Opcode Name | Value |
|---|---|
| DW_LNE_end_sequence | 1 |
| DW_LNE_set_address | 2 |
| DW_LNE_define_file | 3 |

**Figure 35.** Extended Opcode Encodings

## 7.22 Macro Information

The source line numbers and source file indices encoded in the macro information section are represented as unsigned LEB128 numbers as are the constants in an DW_MACINFO_vendor_ext entry. The macinfo type is encoded as a single byte. The encodings are given in Figure 36.

| Macinfo Type Name | Value |
|---|---|
| DW_MACINFO_define | 1 |
| DW_MACINFO_undef | 2 |
| DW_MACINFO_start_file | 3 |
| DW_MACINFO_end_file | 4 |
| DW_MACINFO_vendor_ext | 255 |

**Figure 36.** Macinfo Type Encodings

## 7.23 Call Frame Information

The value of the CIE id in the CIE header is 0xffffffff. The initial value of the CIE version number is 1.

Call frame instructions are encoded in one or more bytes. The primary opcode is encoded in the high order two bits of the first byte (that is, opcode = byte >> 6). An operand or extended opcode may be encoded in the low order 6 bits. Additional operands are encoded in subsequent bytes. The instructions and their encodings are presented in Figure 37.

| Instruction | High<br>2 Bits | Low<br>6 Bits | Operand 1 | Operand 2 |
|---|---|---|---|---|
| DW_CFA_advance_loc | 0x1 | delta | | |
| DW_CFA_offset | 0x2 | register | ULEB128 offset | |
| DW_CFA_restore | 0x3 | register | | |
| DW_CFA_set_loc | 0 | 0x01 | address | |
| DW_CFA_advance_loc1 | 0 | 0x02 | 1-byte delta | |
| DW_CFA_advance_loc2 | 0 | 0x03 | 2-byte delta | |
| DW_CFA_advance_loc4 | 0 | 0x04 | 4-byte delta | |
| DW_CFA_offset_extended | 0 | 0x05 | ULEB128 register | ULEB128 offset |
| DW_CFA_restore_extended | 0 | 0x06 | ULEB128 register | |
| DW_CFA_undefined | 0 | 0x07 | ULEB128 register | |
| DW_CFA_same_value | 0 | 0x08 | ULEB128 register | |
| DW_CFA_register | 0 | 0x09 | ULEB128 register | ULEB128 register |
| DW_CFA_remember_state | 0 | 0x0a | | |
| DW_CFA_restore_state | 0 | 0x0b | | |
| DW_CFA_def_cfa | 0 | 0x0c | ULEB128 register | ULEB128 offset |
| DW_CFA_def_cfa_register | 0 | 0x0d | ULEB128 register | |
| DW_CFA_def_cfa_offset | 0 | 0x0e | ULEB128 offset | |
| DW_CFA_nop | 0 | 0 | | |
| DW_CFA_lo_user | 0 | 0x1c | | |
| DW_CFA_hi_user | 0 | 0x3f | | |

**Figure 37.** Call frame instruction encodings

## 7.24 Dependencies

The debugging information in this format is intended to exist in the .debug_abbrev, .debug_aranges, .debug_frame, .debug_info, .debug_line, .debug_loc, .debug_macinfo, .debug_pubnames and .debug_str sections of an object file. The information is not word-aligned, so the assembler must provide a way for the compiler to produce 2-byte and 4-byte quantities without alignment restrictions, and the linker must be able to relocate a 4-byte reference at an arbitrary alignment. In target architectures with 64-bit addresses, the assembler and linker must similarly handle 8-byte references at arbitrary alignments.

## 8. FUTURE DIRECTIONS

The UNIX International Programming Languages SIG is working on a specification for a set of interfaces for reading DWARF information, that will hide changes in the representation of that information from its consumers. It is hoped that using these interfaces will make the transition from DWARF Version 1 to Version 2 much simpler and will make it easier for a single consumer to support objects using either Version 1 or Version 2 DWARF.

A draft of this specification is available for review from UNIX International. The Programming Languages SIG wishes to stress, however, that the specification is still in flux.

## Appendix 1 -- Current Attributes by Tag Value

The list below enumerates the attributes that are most applicable to each type of debugging information entry. DWARF does not in general require that a given debugging information entry contain a particular attribute or set of attributes. Instead, a DWARF producer is free to generate any, all, or none of the attributes described in the text as being applicable to a given entry. Other attributes (both those defined within this document but not explicitly associated with the entry in question, and new, vendor-defined ones) may also appear in a given debugging entry. Therefore, the list may be taken as instructive, but cannot be considered definitive.

| TAG NAME | APPLICABLE ATTRIBUTES |
|---|---|
| DW_TAG_access_declaration | DECL†<br>DW_AT_accessibility<br>DW_AT_name<br>DW_AT_sibling |
| DW_TAG_array_type | DECL<br>DW_AT_abstract_origin<br>DW_AT_accessibility<br>DW_AT_byte_size<br>DW_AT_declaration<br>DW_AT_name<br>DW_AT_ordering<br>DW_AT_sibling<br>DW_AT_start_scope<br>DW_AT_stride_size<br>DW_AT_type<br>DW_AT_visibility |
| DW_TAG_base_type | DW_AT_bit_offset<br>DW_AT_bit_size<br>DW_AT_byte_size<br>DW_AT_encoding<br>DW_AT_name<br>DW_AT_sibling |
| DW_TAG_catch_block | DW_AT_abstract_origin<br>DW_AT_high_pc<br>DW_AT_low_pc<br>DW_AT_segment<br>DW_AT_sibling |

† DW_AT_decl_column, DW_AT_decl_file, DW_AT_decl_line.

## Appendix 1 (cont'd) -- Current Attributes by Tag Value

| TAG NAME | APPLICABLE ATTRIBUTES |
|---|---|
| DW_TAG_class_type | DECL<br>DW_AT_abstract_origin<br>DW_AT_accessibility<br>DW_AT_byte_size<br>DW_AT_declaration<br>DW_AT_name<br>DW_AT_sibling<br>DW_AT_start_scope<br>DW_AT_visibility |
| DW_TAG_common_block | DECL<br>DW_AT_declaration<br>DW_AT_location<br>DW_AT_name<br>DW_AT_sibling<br>DW_AT_visibility |
| DW_TAG_common_inclusion | DECL<br>DW_AT_common_reference<br>DW_AT_declaration<br>DW_AT_sibling<br>DW_AT_visibility |
| DW_TAG_compile_unit | DW_AT_base_types<br>DW_AT_comp_dir<br>DW_AT_identifier_case<br>DW_AT_high_pc<br>DW_AT_language<br>DW_AT_low_pc<br>DW_AT_macro_info<br>DW_AT_name<br>DW_AT_producer<br>DW_AT_sibling<br>DW_AT_stmt_list |
| DW_TAG_const_type | DW_AT_sibling<br>DW_AT_type |

## Appendix 1 (cont'd) -- Current Attributes by Tag Value

| TAG NAME | APPLICABLE ATTRIBUTES |
|---|---|
| DW_TAG_constant | DECL<br>DW_AT_accessibility<br>DW_AT_constant_value<br>DW_AT_declaration<br>DW_AT_external<br>DW_AT_name<br>DW_AT_sibling<br>DW_AT_start_scope<br>DW_AT_type<br>DW_AT_visibility |
| DW_TAG_entry_point | DW_AT_address_class<br>DW_AT_low_pc<br>DW_AT_name<br>DW_AT_return_addr<br>DW_AT_segment<br>DW_AT_sibling<br>DW_AT_static_link<br>DW_AT_type |
| DW_TAG_enumeration_type | DECL<br>DW_AT_abstract_origin<br>DW_AT_accessibility<br>DW_AT_byte_size<br>DW_AT_declaration<br>DW_AT_name<br>DW_AT_sibling<br>DW_AT_start_scope<br>DW_AT_visibility |
| DW_TAG_enumerator | DECL<br>DW_AT_const_value<br>DW_AT_name<br>DW_AT_sibling |
| DW_TAG_file_type | DECL<br>DW_AT_abstract_origin<br>DW_AT_byte_size<br>DW_AT_name<br>DW_AT_sibling<br>DW_AT_start_scope<br>DW_AT_type<br>DW_AT_visibility |

## Appendix 1 (cont'd) -- Current Attributes by Tag Value

| TAG NAME | APPLICABLE ATTRIBUTES |
|---|---|
| DW_TAG_formal_parameter | DECL<br>DW_AT_abstract_origin<br>DW_AT_artificial<br>DW_AT_default_value<br>DW_AT_is_optional<br>DW_AT_location<br>DW_AT_name<br>DW_AT_segment<br>DW_AT_sibling<br>DW_AT_type<br>DW_AT_variable_parameter |
| DW_TAG_friend | DECL<br>DW_AT_abstract_origin<br>DW_AT_friend<br>DW_AT_sibling |
| DW_TAG_imported_declaration | DECL<br>DW_AT_accessibility<br>DW_AT_import<br>DW_AT_name<br>DW_AT_sibling<br>DW_AT_start_scope |
| DW_TAG_inheritance | DECL<br>DW_AT_accessibility<br>DW_AT_data_member_location<br>DW_AT_sibling<br>DW_AT_type<br>DW_AT_virtuality |
| DW_TAG_inlined_subroutine | DECL<br>DW_AT_abstract_origin<br>DW_AT_high_pc<br>DW_AT_low_pc<br>DW_AT_segment<br>DW_AT_sibling<br>DW_AT_return_addr<br>DW_AT_start_scope |
| DW_TAG_label | DW_AT_abstract_origin<br>DW_AT_low_pc<br>DW_AT_name<br>DW_AT_segment<br>DW_AT_start_scope<br>DW_AT_sibling |

## Appendix 1 (cont'd) -- Current Attributes by Tag Value

| TAG NAME | APPLICABLE ATTRIBUTES |
|---|---|
| DW_TAG_lexical_block | DW_AT_abstract_origin<br>DW_AT_high_pc<br>DW_AT_low_pc<br>DW_AT_name<br>DW_AT_segment<br>DW_AT_sibling |
| DW_TAG_member | DECL<br>DW_AT_accessibility<br>DW_AT_byte_size<br>DW_AT_bit_offset<br>DW_AT_bit_size<br>DW_AT_data_member_location<br>DW_AT_declaration<br>DW_AT_name<br>DW_AT_sibling<br>DW_AT_type<br>DW_AT_visibility |
| DW_TAG_module | DECL<br>DW_AT_accessibility<br>DW_AT_declaration<br>DW_AT_high_pc<br>DW_AT_low_pc<br>DW_AT_name<br>DW_AT_priority<br>DW_AT_segment<br>DW_AT_sibling<br>DW_AT_visibility |
| DW_TAG_namelist | DECL<br>DW_AT_accessibility<br>DW_AT_abstract_origin<br>DW_AT_declaration<br>DW_AT_sibling<br>DW_AT_visibility |
| DW_TAG_namelist_item | DECL<br>DW_AT_namelist_item<br>DW_AT_sibling |
| DW_TAG_packed_type | DW_AT_sibling<br>DW_AT_type |

## Appendix 1 (cont'd) -- Current Attributes by Tag Value

| TAG NAME | APPLICABLE ATTRIBUTES |
|---|---|
| DW_TAG_pointer_type | DW_AT_address_class<br>DW_AT_sibling<br>DW_AT_type |
| DW_TAG_ptr_to_member_type | DECL<br>DW_AT_abstract_origin<br>DW_AT_address_class<br>DW_AT_containing_type<br>DW_AT_declaration<br>DW_AT_name<br>DW_AT_sibling<br>DW_AT_type<br>DW_AT_use_location<br>DW_AT_visibility |
| DW_TAG_reference_type | DW_AT_address_class<br>DW_AT_sibling<br>DW_AT_type |
| DW_TAG_set_type | DECL<br>DW_AT_abstract_origin<br>DW_AT_accessibility<br>DW_AT_byte_size<br>DW_AT_declaration<br>DW_AT_name<br>DW_AT_start_scope<br>DW_AT_sibling<br>DW_AT_type<br>DW_AT_visibility |
| DW_TAG_string_type | DECL<br>DW_AT_accessibility<br>DW_AT_abstract_origin<br>DW_AT_byte_size<br>DW_AT_declaration<br>DW_AT_name<br>DW_AT_segment<br>DW_AT_sibling<br>DW_AT_start_scope<br>DW_AT_string_length<br>DW_AT_visibility |

## Appendix 1 (cont'd) -- Current Attributes by Tag Value

| TAG NAME | APPLICABLE ATTRIBUTES |
|---|---|
| DW_TAG_structure_type | DECL<br>DW_AT_abstract_origin<br>DW_AT_accessibility<br>DW_AT_byte_size<br>DW_AT_declaration<br>DW_AT_name<br>DW_AT_sibling<br>DW_AT_start_scope<br>DW_AT_visibility |
| DW_TAG_subprogram | DECL<br>DW_AT_abstract_origin<br>DW_AT_accessibility<br>DW_AT_address_class<br>DW_AT_artificial<br>DW_AT_calling_convention<br>DW_AT_declaration<br>DW_AT_external<br>DW_AT_frame_base<br>DW_AT_high_pc<br>DW_AT_inline<br>DW_AT_low_pc<br>DW_AT_name<br>DW_AT_prototyped<br>DW_AT_return_addr<br>DW_AT_segment<br>DW_AT_sibling<br>DW_AT_specification<br>DW_AT_start_scope<br>DW_AT_static_link<br>DW_AT_type<br>DW_AT_visibility<br>DW_AT_virtuality<br>DW_AT_vtable_elem_location |

## Appendix 1 (cont'd) -- Current Attributes by Tag Value

| TAG NAME | APPLICABLE ATTRIBUTES |
|---|---|
| DW_TAG_subrange_type | DECL<br>DW_AT_abstract_origin<br>DW_AT_accessibility<br>DW_AT_byte_size<br>DW_AT_count<br>DW_AT_declaration<br>DW_AT_lower_bound<br>DW_AT_name<br>DW_AT_sibling<br>DW_AT_type<br>DW_AT_upper_bound<br>DW_AT_visibility |
| DW_TAG_subroutine_type | DECL<br>DW_AT_abstract_origin<br>DW_AT_accessibility<br>DW_AT_address_class<br>DW_AT_declaration<br>DW_AT_name<br>DW_AT_prototyped<br>DW_AT_sibling<br>DW_AT_start_scope<br>DW_AT_type<br>DW_AT_visibility |
| DW_TAG_template_type_param | DECL<br>DW_AT_name<br>DW_AT_sibling<br>DW_AT_type |
| DW_TAG_template_value_param | DECL<br>DW_AT_name<br>DW_AT_const_value<br>DW_AT_sibling<br>DW_AT_type |
| DW_TAG_thrown_type | DECL<br>DW_AT_sibling<br>DW_AT_type |
| DW_TAG_try_block | DW_AT_abstract_origin<br>DW_AT_high_pc<br>DW_AT_low_pc<br>DW_AT_segment<br>DW_AT_sibling |

## Appendix 1 (cont'd) -- Current Attributes by Tag Value

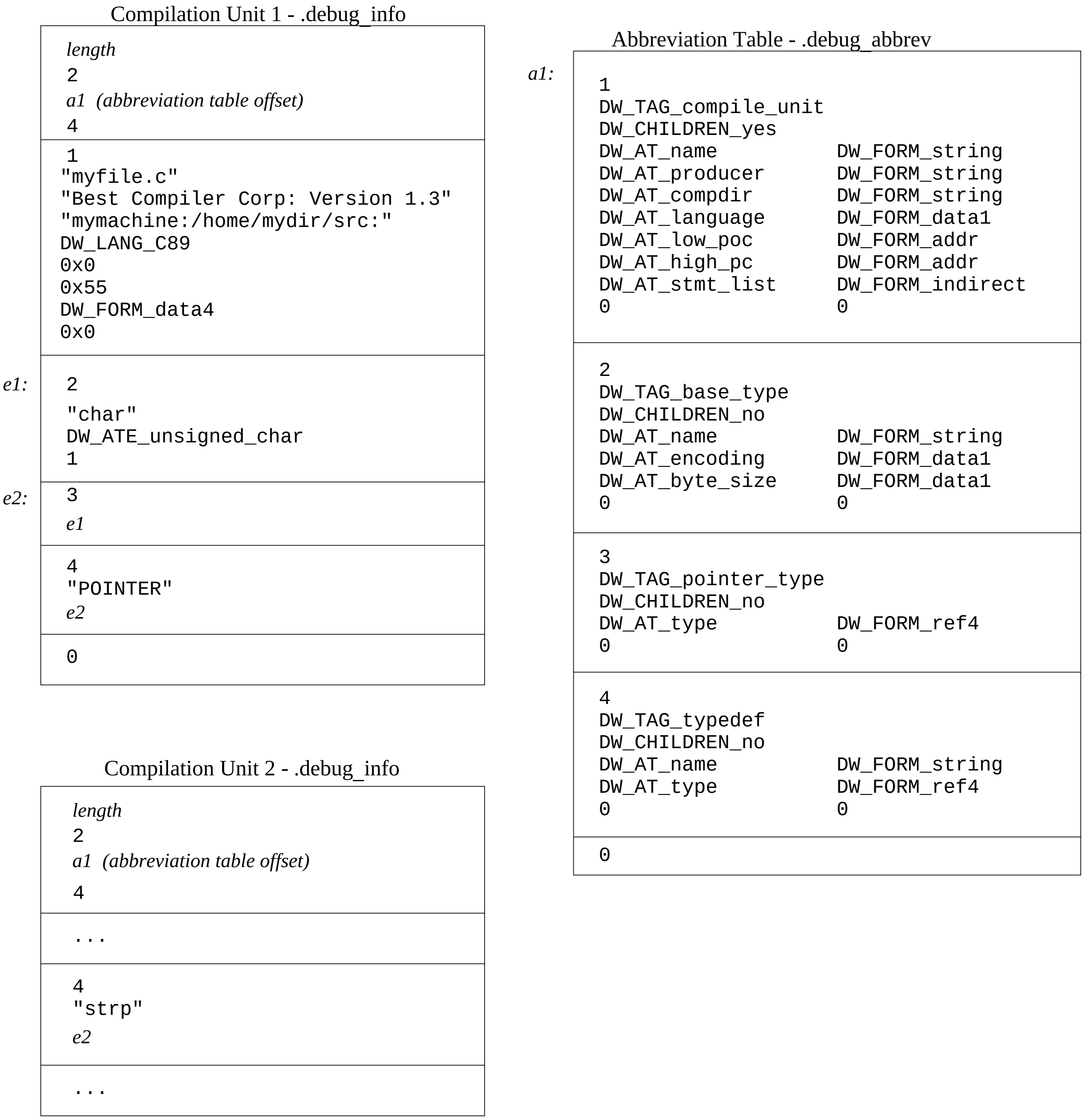
| TAG NAME | APPLICABLE ATTRIBUTES |
|---|---|
| DW_TAG_typedef | DECL<br>DW_AT_abstract_origin<br>DW_AT_accessibility<br>DW_AT_declaration<br>DW_AT_name<br>DW_AT_sibling<br>DW_AT_start_scope<br>DW_AT_type<br>DW_AT_visibility |
| DW_TAG_union_type | DECL<br>DW_AT_abstract_origin<br>DW_AT_accessibility<br>DW_AT_byte_size<br>DW_AT_declaration<br>DW_AT_friends<br>DW_AT_name<br>DW_AT_sibling<br>DW_AT_start_scope<br>DW_AT_visibility |
| DW_TAG_unspecified_parameters | DECL<br>DW_AT_abstract_origin<br>DW_AT_artificial<br>DW_AT_sibling |
| DW_TAG_variable | DECL<br>DW_AT_accessibility<br>DW_AT_constant_value<br>DW_AT_declaration<br>DW_AT_external<br>DW_AT_location<br>DW_AT_name<br>DW_AT_segment<br>DW_AT_sibling<br>DW_AT_specification<br>DW_AT_start_scope<br>DW_AT_type<br>DW_AT_visibility |

## Appendix 1 (cont'd) -- Current Attributes by Tag Value

| TAG NAME | APPLICABLE ATTRIBUTES |
|---|---|
| DW_TAG_variant | DECL<br>DW_AT_accessibility<br>DW_AT_abstract_origin<br>DW_AT_declaration<br>DW_AT_discr_list<br>DW_AT_discr_value<br>DW_AT_sibling |
| DW_TAG_variant_part | DECL<br>DW_AT_accessibility<br>DW_AT_abstract_origin<br>DW_AT_declaration<br>DW_AT_discr<br>DW_AT_sibling<br>DW_AT_type |
| DW_TAG_volatile_type | DW_AT_sibling<br>DW_AT_type |
| DW_TAG_with_statement | DW_AT_accessibility<br>DW_AT_address_class<br>DW_AT_declaration<br>DW_AT_high_pc<br>DW_AT_location<br>DW_AT_low_pc<br>DW_AT_segment<br>DW_AT_sibling<br>DW_AT_type<br>DW_AT_visibility |

## Appendix 2 -- Organization of Debugging Information

The following diagram depicts the relationship of the abbreviation tables contained in the `.debug_abbrev` section to the information contained in the `.debug_info` section. Values are given in symbolic form, where possible.



Compilation Unit 1 - .debug_info

```
length
2
a1  (abbreviation table offset)
4
---------------------------------
1
"myfile.c"
"Best Compiler Corp: Version 1.3"
"mymachine:/home/mydir/src:"
DW_LANG_C89
0x0
0x55
DW_FORM_data4
0x0
---------------------------------
e1:  2
     "char"
     DW_ATE_unsigned_char
     1
---------------------------------
e2:  3
     e1
---------------------------------
4
"POINTER"
e2
---------------------------------
0
```

Compilation Unit 2 - .debug_info

```
length
2
a1  (abbreviation table offset)
4
---------------------------------
...
---------------------------------
4
"strp"
e2
---------------------------------
...
```

Abbreviation Table - .debug_abbrev

```
a1:  1
     DW_TAG_compile_unit
     DW_CHILDREN_yes
     DW_AT_name          DW_FORM_string
     DW_AT_producer      DW_FORM_string
     DW_AT_compdir       DW_FORM_string
     DW_AT_language      DW_FORM_data1
     DW_AT_low_poc       DW_FORM_addr
     DW_AT_high_pc       DW_FORM_addr
     DW_AT_stmt_list     DW_FORM_indirect
     0                   0
---------------------------------------------
     2
     DW_TAG_base_type
     DW_CHILDREN_no
     DW_AT_name          DW_FORM_string
     DW_AT_encoding      DW_FORM_data1
     DW_AT_byte_size     DW_FORM_data1
     0                   0
---------------------------------------------
     3
     DW_TAG_pointer_type
     DW_CHILDREN_no
     DW_AT_type          DW_FORM_ref4
     0                   0
---------------------------------------------
     4
     DW_TAG_typedef
     DW_CHILDREN_no
     DW_AT_name          DW_FORM_string
     DW_AT_type          DW_FORM_ref4
     0                   0
---------------------------------------------
     0
```

## Appendix 3 -- Statement Program Examples

Consider this simple source file and the resulting machine code for the Intel 8086 processor:

```
1:  int
2:  main()
    0x239:              push pb
    0x23a:              mov bp,sp
3:  {
4:  printf("Omit needless words\n");
    0x23c:              mov ax,0xaa
    0x23f:              push ax
    0x240:              call _printf
    0x243:              pop cx
5:  exit(0);
    0x244:              xor ax,ax
    0x246:              push ax
    0x247:              call _exit
    0x24a:              pop cx
6:  }
    0x24b:              pop bp
    0x24c:              ret
7:
    0x24d:
```

If the statement program prologue specifies the following:

| | |
|---|---|
| minimum_instruction_length | 1 |
| opcode_base | 10 |
| line_base | 1 |
| line_range | 15 |

Then one encoding of the statement program would occupy 12 bytes (the opcode SPECIAL(*m*, *n*) indicates the special opcode generated for a line increment of *m* and an address increment of *n*):

| Opcode | Operand | Byte Stream |
|---|---|---|
| DW_LNS_advance_pc | LEB128(0x239) | 0x2, 0xb9, 0x04 |
| SPECIAL(2, 0) | | 0xb |
| SPECIAL(2, 3) | | 0x38 |
| SPECIAL(1, 8) | | 0x82 |
| SPECIAL(1, 7) | | 0x73 |
| DW_LNS_advance_pc | LEB128(2) | 0x2, 0x2 |
| DW_LNE_end_sequence | | 0x0, 0x1, 0x1 |

An alternate encoding of the same program using standard opcodes to advance the program counter would occupy 22 bytes:

| Opcode | Operand | Byte Stream |
|---|---|---|
| DW_LNS_fixed_advance_pc | 0x239 | 0x9, 0x39, 0x2 |
| SPECIAL(2, 0) | | 0xb |
| DW_LNS_fixed_advance_pc | 0x3 | 0x9, 0x3, 0x0 |
| SPECIAL(2, 0) | | 0xb |
| DW_LNS_fixed_advance_pc | 0x8 | 0x9, 0x8, 0x0 |
| SPECIAL(1, 0) | | 0xa |
| DW_LNS_fixed_advance_pc | 0x7 | 0x9, 0x7, 0x0 |
| SPECIAL(1, 0) | | 0xa |
| DW_LNS_fixed_advance_pc | 0x2 | 0x9, 0x2, 0x0 |
| DW_LNE_end_sequence | | 0x0, 0x1, 0x1 |

## Appendix 4 -- Encoding and decoding variable length data

Here are algorithms expressed in a C-like pseudo-code to encode and decode signed and unsigned numbers in LEB128:

### Encode an unsigned integer:

```
do
{
                        byte = low order 7 bits of value;
                        value >>= 7;
                        if (value != 0)         /* more bytes to come */
                                set high order bit of byte;
                        emit byte;
} while (value != 0);
```

### Encode a signed integer:

```
more = 1;
negative = (value < 0);
size = no. of bits in signed integer;
while(more)
{
                        byte = low order 7 bits of value;
                        value >>= 7;
                        /* the following is unnecessary if the implementation of >>=
                         * uses an arithmetic rather than logical shift for a signed
                         * left operand
                         */
                        if (negative)
                                /* sign extend */
                                value |= - (1 << (size - 7));
                        /* sign bit of byte is 2nd high order bit (0x40) */
                        if ((value == 0 && sign bit of byte is clear) ||
                                (value == -1 && sign bit of byte is set))
                                more = 0;
                        else
                                set high order bit of byte;
                        emit byte;
}
```

### Decode unsigned LEB128 number:

```
result = 0;
shift = 0;
while(true)
{
                       byte = next byte in input;
                       result |= (low order 7 bits of byte << shift);
                       if (high order bit of byte == 0)
                               break;
                       shift += 7;
}
```

### Decode signed LEB128 number:

```
result = 0;
shift = 0;
size = no. of bits in signed integer;
while(true)
{
                       byte = next byte in input;
                       result |= (low order 7 bits of byte << shift);
                       shift += 7;
                       /* sign bit of byte is 2nd high order bit (0x40) */
                       if (high order bit of byte == 0)
                               break;
}
if ((shift < size) && (sign bit of byte is set))
                       /* sign extend */
                       result |= - (1 << shift);
```

## Appendix 5 -- Call Frame Information Examples

The following example uses a hypothetical RISC machine in the style of the Motorola 88000.

- Memory is byte addressed.
- Instructions are all 4-bytes each and word aligned.
- Instruction operands are typically of the form:

  <destination reg> <source reg> <constant>

- The address for the load and store instructions is computed by adding the contents of the source register with the constant.
- There are 8 4-byte registers:

  R0 always 0  
  R1 holds return address on call  
  R2-R3 temp registers (not preserved on call)  
  R4-R6 preserved on call  
  R7 stack pointer.

- The stack grows in the negative direction.

The following are two code fragments from a subroutine called `foo` that uses a frame pointer (in addition to the stack pointer.) The first column values are byte addresses.

```
        ;; start prologue
foo     sub     R7, R7, <fsize>         ; Allocate frame
foo+4   store   R1, R7, (<fsize>-4)     ; Save the return address
foo+8   store   R6, R7, (<fsize>-8)     ; Save R6
foo+12  add     R6, R7, 0               ; R6 is now the Frame ptr
foo+16  store   R4, R6, (<fsize>-12)    ; Save a preserve reg.
        ;; This subroutine does not change R5
        ...
        ;; Start epilogue (R7 has been returned to entry value)
foo+64  load    R4, R6, (<fsize>-12)    ; Restore R4
foo+68  load    R6, R7, (<fsize>-8)     ; Restore R6
foo+72  load    R1, R7, (<fsize>-4)     ; Restore return address
foo+76  add     R7, R7, <fsize>         ; Deallocate frame
foo+80  jump    R                                                   ; Return
foo+84
```

The table for the `foo` subroutine is as follows. It is followed by the corresponding fragments from the `.debug_frame` section.

| Loc | CFA | R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 |
|---|---|---|---|---|---|---|---|---|---|---|
| foo | [R7]+0 | s | u | u | u | s | s | s | s | r1 |
| foo+4 | [R7]+fsize | s | u | u | u | s | s | s | s | r1 |
| foo+8 | [R7]+fsize | s | u | u | u | s | s | s | s | c4 |
| foo+12 | [R7]+fsize | s | u | u | u | s | s | c8 | s | c4 |
| foo+16 | [R6]+fsize | s | u | u | u | s | s | c8 | s | c4 |
| foo+20 | [R6]+fsize | s | u | u | u | c12 | s | c8 | s | c4 |
| foo+64 | [R6]+fsize | s | u | u | u | c12 | s | c8 | s | c4 |
| foo+68 | [R6]+fsize | s | u | u | u | s | s | c8 | s | c4 |
| foo+72 | [R7]+fsize | s | u | u | u | s | s | s | s | c4 |
| foo+76 | [R7]+fsize | s | u | u | u | s | s | s | s | r1 |
| foo+80 | [R7]+0 | s | u | u | u | s | s | s | s | r1 |

notes:

1. R8 is the return address
2. s = same_value rule
3. u = undefined rule
4. rN = register(N) rule
5. cN = offset(N) rule

Common Information Entry (CIE):

```
cie      32                       ; length
cie+4    0xffffffff               ; CIE_id
cie+8    1                        ; version
cie+9    0                        ; augmentation
cie+10   4                        ; code_alignment_factor
cie+11   4                        ; data_alignment_factor
cie+12   8                        ; R8 is the return addr.
cie+13   DW_CFA_def_cfa (7, 0)    ; CFA = [R7]+0
cie+16   DW_CFA_same_value (0)    ; R0 not modified (=0)
cie+18   DW_CFA_undefined (1)     ; R1 scratch
cie+20   DW_CFA_undefined (2)     ; R2 scratch
cie+22   DW_CFA_undefined (3)     ; R3 scratch
cie+24   DW_CFA_same_value (4)    ; R4 preserve
cie+26   DW_CFA_same_value (5)    ; R5 preserve
cie+28   DW_CFA_same_value (6)    ; R6 preserve
cie+30   DW_CFA_same_value (7)    ; R7 preserve
cie+32   DW_CFA_register (8, 1)   ; R8 is in R1
cie+35   DW_CFA_nop               ; padding
cie+36
```

Frame Description Entry (FDE):

```
fde      40                                 ; length
fde+4    cie                                ; CIE_ptr
fde+8    foo                                ; initial_location
fde+12   84                                 ; address_range
fde+16   DW_CFA_advance_loc(1)              ; instructions
fde+17   DW_CFA_def_cfa_offset(<fsize>/4)   ; assuming <fsize> < 512
fde+19   DW_CFA_advance_loc(1)
fde+20   DW_CFA_offset(8,1)
fde+22   DW_CFA_advance_loc(1)
fde+23   DW_CFA_offset(6,2)
fde+25   DW_CFA_advance_loc(1)
fde+26   DW_CFA_def_cfa_register(6)
fde+28   DW_CFA_advance_loc(1)
fde+29   DW_CFA_offset(4,3)
fde+31   DW_CFA_advance_loc(11)
fde+32   DW_CFA_restore(4)
fde+33   DW_CFA_advance_loc(1)
fde+34   DW_CFA_restore(6)
fde+35   DW_CFA_def_cfa_register(7)
fde+37   DW_CFA_advance_loc(1)
fde+38   DW_CFA_restore(8)
fde+39   DW_CFA_advance_loc(1)
fde+40   DW_CFA_def_cfa_offset(0)
fde+42   DW_CFA_nop                         ; padding
fde+43   DW_CFA_nop                         ; padding
fde+44
```

# III

# Relocatable Object Module Format (OMF)

# TIS Portable Formats Specification, Version 1.1
# OMF:  Relocatable Object Module Format

This document represents the TIS OMF specification, the 32-bit Relocatable Object Module Format standard.  OMF is the official definition for the standard relocatable object module format (OMF) for 32-bit applications and tools for Intel Architectures.  This document presents the heritage of the format with a focus on its current 32-bit aspects.  It is published and controlled by the Tool Interface Standards (TIS) Committee, an open industry standards body, and will be revised as clarifications and technical information become available to the TIS Committee.

This OMF represents the consolidation of different OMFs in use today into a single industry standard specification.  The information included within this document has been compiled from the following documents:  *The MS-DOS Encyclopedia* by Microsoft Press, the *PharLap 386/Link Reference Manual*, the Intel 8086 object module specification (*Intel Technical Specification* 121748-001), the *IBM OS/2 16/32-bit Object Module Format (OMF) and Linear eXecutable Module Format (LX), Revision 7 (dated April 24, 1993)*, and internal Microsoft documents.

Where there were conflicts among these source documents, the TIS Committee resolved them.  Certain issues regarding the different OMFs that contribute to this standard have been clarified for Version 1.1. After an extensive review, the TIS Committee has made the following changes:

- The name string subfields in various records are more consistent.

- The public name fields in the COMDAT, LINSYM, and NBKPAT records were revised.

- The LINNUM record was revised.

- The COMDEF record was revised to include Borland VIRDEF record information.

- The library names hashing algorithm in Appendix 2 was revised.

This document is intended for individuals who have a background knowledge of how source code is converted into an executable file.  For details on a particular vendor-extension of the OMF, refer to the appropriate vendor's documentation.

Note that this specification is not applicable for vendors choosing to build products with the Executable and Linkable Format (ELF), because ELF is both a linkable and loadable specification.

# OMF: Relocatable Object Module Format

## Table of Contents

## The Object Record Format

### Record Format

All object records conform to the following format:

| | | | |
|---|---|---|---|
| | | <----------------------Record Length in Bytes----------------> | |
| 1 | 2 | <variable> | 1 |
| Record Type | Record Length | Record Contents | Checksum or 0 |

The Record Type field is a 1-byte field containing the hexadecimal number that identifies the type of object record. The format is determined by the least significant bit of the Record type field. An odd Record Type indicates that certain numeric fields within the record contain 32-bit values; an even Record Type indicates that those fields contain 16-bit values. The affected fields are described with each record. Note that this principle does not govern the Use32/Use16 segment attribute (which is set in the ACBP byte of SEGDEF records); it simply specifies the size of certain numeric fields within the record. It is possible to use 16-bit OMF records to generate 32-bit segments, or vice versa.

The Record Length field is a 2-byte field that gives the length of the remainder of the object record in bytes (excluding the bytes in the Record Type and Record Length fields). The record length is stored with the low-order byte first. An entire record occupies 3 bytes plus the number of bytes in the Record Length field.

The Record Contents field varies in size and format, depending on the record type.

The Checksum field is a 1-byte field that contains the negative sum (modulo 256) of all other bytes in the record. In other words, the checksum byte is calculated so that the low-order byte of the sum of all the bytes in the record, including the checksum byte, equals 0. Overflow is ignored. Some compilers write a 0 byte rather than computing the checksum, so either form should be accepted by programs that process object modules.

> ***Note:*** *The maximum size of the entire record (unless otherwise noted for specific record types) is 1024 bytes.*

## Frequent Object Record Subfields

The contents of each record are determined by the record type, but certain subfields appear frequently enough to be explained separately. The format of such fields is below.

### Names

A name string is encoded as an 8-bit unsigned count followed by a string of count characters, refered to as*count, char* format. The character set is usually some ASCII subset. A null name is specified by a single byte of 0 (indicating a string of length 0).

### Indexed References

Certain items are ordered by occurrence and are referenced by index. The first occurrence of the item has index number 1. Index fields may contain 0 (indicating that they are not present) or values from 1 through 7FFF. The index number field in an object record can be either 1 or 2 bytes long. If the number is in the range 0–7FH, the high-order bit (bit 7) is 0 and the low-order bits contain the index number, so the field is only 1 byte long. If the index number is in the range 80–7FFFH, the field is 2 bytes long. The high-order bit of the first byte in the field is set to 1, and the high-order byte of the index number (which must be in the range 0–7FH) fits in the remaining 7 bits. The low-order byte of the index number is specified in the second byte of the field. The code to decode an index is:

```
if (first_byte & 0x80)
   index_word = (first_byte & 7F) * 0x100 + second_byte;
else
   index_word = first_byte;
```

### Type Indexes

Type Index fields occupy 1 or 2 bytes and occur in PUBDEF, LPUBDEF, COMDEF, LCOMDEF, EXTDEF, and LEXTDEF records. These type index fields were used in old versions of the OMF to reference TYPDEF records. This usage is obsolete. This field is usually left empty (encoded as 1 byte with value 0). However some linkers may use this for debug information or other purposes.

### Ordered Collections

Certain records and record groups are ordered so that the records may be referred to with indexes (the format of indexes is described in the "Indexed References" section). The same format is used whether an index refers to names, logical segments, or other items.

The overall ordering is obtained from the order of the records within the file together with the ordering of repeated fields within these records. Such ordered collections are referenced by index, counting from 1 (index 0 indicates unknown or not specified).

For example, there may be many LNAMES records within a module, and each of those records may contain many names. The names are indexed starting at 1 for the first name in the first LNAMES record encountered while reading the file, 2 for the second name in the first record, and so forth, with the highest index for the last name in the last LNAMES record encountered.

The ordered collections are:

**Names** Ordered by occurrence of LNAMES records and names within each. Referenced as a name index.

**Logical Segments** Ordered by occurrence of SEGDEF records in file. Referenced as a segment index.

**Groups** Ordered by occurrence of GRPDEF records in file. Referenced as a group index.

**External Symbols** Ordered by occurrence of EXTDEF, COMDEF, LEXTDEF, and LCOMDEF records and symbols within each. Referenced as an external name index (in FIXUP subrecords).

### Numeric 2-Byte and 4-Byte Fields

Words and double words (16- and 32-bit quantities, respectively) are stored in little endian byte order (lowest address is least significant).

Certain records, notably SEGDEF, PUBDEF, LPUBDEF, LINNUM, LEDATA, LIDATA, FIXUPP, and MODEND, contain size, offset, and displacement values that may be 32-bit quantities for Use32 segments. The encoding is as follows:

- When the least-significant bit of the record type byte is set (that is, the record type is an odd number), the numeric fields are 4 bytes.
- When the least-significant bit of the record type byte is clear, the fields occupy 2 bytes. The values are zero-extended when applied to Use32 segments.

***Note:*** *See the description of SEGDEF records for an explanation of Use16/Use32 segments.*

## Order of Records

The sequence in which the types of object records appear in an object module is fairly flexible in some respects. Several record types are optional, and if the type of information they carry is unnecessary, they are omitted from the object module. In addition, most object record types can occur more than once in the same object module. And because object records are variable in length, it is often possible to choose between combining information into one large record or breaking it down into several smaller records of the same type.

An important constraint on the order in which object records appear is the need for some types of object records to refer to information contained in other records. Because the linker processes the records sequentially, object records containing such information must precede the records that refer to the information. For example, two types of object records, SEGDEF and GRPDEF, refer to the names contained in an LNAMES record. Thus, an LNAMES record must appear before any SEGDEF or GRPDEF records that refer to it so that the names in the LNAMES record are known to the linker by the time it processes the SEGDEF or GRPDEF records.

The record order is chosen so that the number of linker passes through an object module are minimized. Most linkers make two passes through the object modules: the first pass may be cut short by the presence of the Link Pass Separator COMENT record; the second pass processes all records.

For greatest linking speed, all symbolic information should occur at the start of the object module. This order is recommended but not mandatory. The general ordering is:

### Identifier Record(s)

THEADR or LHEADR record

***Note**: This must be the first record.*

### Records Processed by Pass 1

The following records may occur in any order but they *must* precede the Link Pass Separator if it is present:

COMENT records identifying object format and extensions

COMENT records other than Link Pass Separator comment

LNAMES or LLNAMES records providing ordered name list

SEGDEF records providing ordered list of program segments

GRPDEF records providing ordered list of logical segments

TYPDEF records (obsolete)

ALIAS records

PUBDEF records locating and naming public symbols

LPUBDEF records locating and naming private symbols

COMDEF, LCOMDEF, EXTDEF, LEXTDEF, and CEXTDEF records

***Note:** This group of records is indexed together, so external name index fields in FIXUPP records may refer to any of the record types listed.*

### Link Pass Separator (Optional)

COMENT class A2 record is used to indicate that Pass 1 of the linker is complete. When this record is encountered, many linkers stop their first pass over the object file. Records preceding the link pass separator define the symbolic information for the file.

For greater linking speed, all LIDATA, LEDATA, FIXUPP, BAKPAT, INCDEF, and LINNUM records should come after the A2 COMENT record, but this is not required. Pass 2 should begin again at the start of the object module so that these records are processed in Pass 2 regardless of where they are placed in the object module.

### Records Ignored by Pass 1 and Processed by Pass 2

The following records may come before or after the Link Pass Separator:

LIDATA, LEDATA, or COMDAT records followed by applicable FIXUPP records

FIXUPP records containing only THREAD subrecords

BAKPAT and NBKPAT FIXUPP records

COMENT class A0, subrecord type 03 (INCDEF) records containing incremental compilation information for FIXUPP and LINNUM records

LINNUM and LINSYM records providing line number and program code or data association

### Terminator

MODEND record indicating end of module with optional start address

## Record Specifics

The following is a list of record types that have been implemented and are described within the body of this document. Details of each implemented record (form and content) are presented in the following sections. The records are listed sequentially by hex value. Conflicts between various OMFs that overlap in their use of record types or fields are marked. For information on obsolete records, please refer to Appendix 3.

### Currently Implemented Records

| | | |
|---|---|---|
| **80H** | **THEADR** | **Translator Header Record** |
| **82H** | **LHEADR** | **Library Module Header Record** |
| **88H** | **COMENT** | **Comment Record (Including all comment class extensions)** |
| **8AH/8BH** | **MODEND** | **Module End Record** |
| **8CH** | **EXTDEF** | **External Names Definition Record** |
| **90H/91H** | **PUBDEF** | **Public Names Definition Record** |
| **94H/95H** | **LINNUM** | **Line Numbers Record** |
| **96H** | **LNAMES** | **List of Names Record** |
| **98H/99H** | **SEGDEF** | **Segment Definition Record** |
| **9AH** | **GRPDEF** | **Group Definition Record** |
| **9CH/9DH** | **FIXUPP** | **Fixup Record** |
| **A0H/A1H** | **LEDATA** | **Logical Enumerated Data Record** |
| **A2H/A3H** | **LIDATA** | **Logical Iterated Data Record** |
| **B0H** | **COMDEF** | **Communal Names Definition Record** |
| **B2H/B3H** | **BAKPAT** | **Backpatch Record** |
| **B4H** | **LEXTDEF** | **Local External Names Definition Record** |
| **B6H/B7H** | **LPUBDEF** | **Local Public Names Definition Record** |
| **B8H** | **LCOMDEF** | **Local Communal Names Definition Record** |
| **BCH** | **CEXTDEF** | **COMDAT External Names Definition Record** |
| **C2H/C3H** | **COMDAT** | **Initialized Communal Data Record** |
| **C4H/C5H** | **LINSYM** | **Symbol Line Numbers Record** |
| **C6H** | **ALIAS** | **Alias Definition Record** |
| **C8H/C9H** | **NBKPAT** | **Named Backpatch Record** |
| **CAH** | **LLNAMES** | **Local Logical Names Definition Record** |
| **CCH** | **VERNUM** | **OMF Version Number Record** |
| **CEH** | **VENDEXT** | **Vendor-specific OMF Extension Record** |
| **F0H** | | **Library Header Record**<br>Although this is not actually an OMF record type, the presence of a record with F0H as the first byte indicates that the module is a library. The format of a library file is given in Appendix 2. |
| **F1H** | | **Library End Record** |

## 80H THEADR—Translator Header Record

### Description

The THEADR record contains the name of the object module. This name identifies an object module within an object library or in messages produced by the linker.

### History

Unchanged from the original Intel 8086 specification.

### Record Format

| 1 | 2 | 1 | <-------String Length------> | 1 |
|---|---|---|---|---|
| 80 | Record Length | String Length | Name String | Checksum |

The String Length byte gives the number of characters in the name string; the name string itself is ASCII. This name is usually that of the file that contains a program's source code (if supplied by the language translator), or may be specified directly by the programmer (for example, TITLE pseudo-operand or assembler NAME directive).

***Notes***

*The name string is always present; a null name is allowed but not recommended (because it doesn't provide much information for a debugging program).*

*The name string indicates the full path and filename of the file that contained the source code for the module.*

*This record, or an LHEADR record must occur as the first object record. More than one header record is allowed (as a result of an object bind, or if the source arose from multiple files as a result of include processing).*

### Example

The following THEADR record was generated by the Microsoft C Compiler:

```
        0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000   80   09   00   07   68   65   6C   6C   6F   2E   63   CB                         ...hello.c.
```

Byte 00H contains 80H, indicating a THEADR record.

Bytes 01-02H contain 0009H, the length of the remainder of the record.

Bytes 03-0AH contain the T-module name. Byte 03H contains 07H, the length of the name, and bytes 04H through 0AH contain the name itself (HELLO.C).

Byte 0BH contains the Checksum field, 0CBH.

## 82H LHEADR—Library Module Header Record

### Description

This record is very similar to the THEADR record. It is used to indicate the name of a module within a library file (which has an internal organization different from that of an object module).

### History

This record type was defined in the original Intel 8086 specification with the same format but with a different purpose, so its use for libraries should be considered a Microsoft extension.

### Record Format

| 1 | 2 | 1 | <-------String Length------> | 1 |
|---|---|---|---|---|
| 82 | Record Length | String Length | Name String | Checksum |

***Note:*** *The THEADR, and LHEADR records are handled identically. See Appendix 2 for a complete description of these library file format.*

## 88H COMENT—Comment Record

### Description

The COMENT record contains a character string that may represent a plain text comment, a symbol meaningful to a program accessing the object module, or even binary-encoded identification data. An object module can contain any number of COMENT records.

### History

Before the VENDEXT record was added for TIS, the COMENT record was the primary way of extending the OMF. These extensions were added or changed for 32-bit linkers and continue to be supported in this standard. The comment classes that have been added or changed are 9D, A0, A1, A2, A4, AA, B0, and B1.

Comment class A2 was added for Microsoft C version 5.0. Histories for comment classes A0, A3, A4, A6, A7, and A8 are given later in this section.

68000 and big-endian comments were added for Microsoft C version 7.0.

### Record Format

The comment records are actually a group of items, classified by comment class.

| 1 | 2 | 1 | 1 | <---------Record Length Minus 3--------> | 1 |
|---|---|---|---|---|---|
| 88 | Record Length | Comment Type | Comment Class | Commentary Byte String (optional) | Checksum |

### Comment Type

The Comment Type field is bit significant; its layout is

| <--------------------------1 byte ------------------------------------------------------------------------------------> | | | | | | | |
|---|---|---|---|---|---|---|---|
| NP | NL | 0 | 0 | 0 | 0 | 0 | 0 |

where

**NP** (no purge bit) is set if the comment is to be preserved by utility programs that manipulate object modules. This bit can protect an important comment, such as a copyright message, from deletion.

**NL** (no list bit) is set if the comment is not to be displayed by utility programs that list the contents of object modules. This bit can hide a comment.

The remaining bits are unused and should be set to 0.

### Comment Class and Commentary Byte String

The Comment Class field is an 8-bit numeric field that conveys information by its value (accompanied by a null byte string) or indicates the information to be found in the accompanying byte string. The byte string's length is determined from the record length, not by an initial count byte.

The values that have been defined (including obsolete values) are the following:

**0** **Translator** Translator; it may name the source language or translator. We recommend that the translator name and version, plus the optimization level used for compilation, be recorded here. Other compiler or assembler options can be included, although current practice seems to be to place these under comment class 9D.

**1** **Intel copyright** Ignored.

**2 – 9B** **Intel reserved** These values are reserved for Intel products. Values from 9C through FF are ignored by Intel products.

**81** **Library specifier—obsolete** Replaced by comment class 9F; contents are identical to 9F.

**9C** **MS-DOS version—obsolete** The commentary byte string field is a 2 byte string that specifies the MS-DOS version number. This comment class is not supported by Microsoft LINK.

**9D** **Memory model** This information is currently generated by the Microsoft C compiler for use by the XENIX linker; it is ignored by the MS-DOS and OS/2 versions of Microsoft LINK. The byte string consists of from one to three ASCII characters and indicates the following:

| | |
|---|---|
| **0, 1, 2, or 3** | 8086, 80186, 80286, or 80386 instructions generated, respectively |
| **O** | Optimization performed on code |
| **s, m, c, l, or h** | Small, medium, compact, large, or huge model |
| **A, B, C, D** | 68000, 68010, 68020, or 68030 instructions generated, respectively |

**9E** **DOSSEG** Sets Microsoft LINK's DOSSEG switch. The byte string is null. This record is included in the startup module in each language library. It directs the linker to use the standardized segment ordering, according to the naming conventions documented with MS-DOS, OS/2, and accompanying language products.

**9F** **Default library search name** The byte string contains a library filename (without a lead count byte and without an extension), which is searched in order to resolve external references within the object module. The default library search can be overridden with LINK's /NODEFAULTLIBRARYSEARCH switch.

| | | | |
|---|---|---|---|
| **A0** | **OMF extensions** | | This class consists of a set of records, identified by subtype (first byte of commentary string). Values supported by some linkers are: |
| | **01** | **IMPDEF** | Import definition record. See the IMPDEF section for a complete description. |
| | **02** | **EXPDEF** | Export definition record. See the EXPDEF section for a complete description. |
| | **03** | **INCDEF** | Incremental compilation record. See the INCDEF section for a complete description. |
| | **04** | **Protected memory library** | 32-bit linkers only; relevant only to 32-bit dynamic-link libraries (DLLs). This comment record is inserted in an object module by the compiler when it encounters the _loadds construct in the source code for a DLL. The linker then sets a flag in the header of the executable file (DLL) to indicate that the DLL should be loaded in such a way that its shared data is protected from corruption. The _loadds keyword tells the compiler to emit modified function prolog code, which loads the DS segment register. (Normal functions don't need this.)<br>When the flag is set in the .EXE header, the loader loads the selector of a protected memory area into DS while performing run-time fixups (relocations). All other DLLs and applications get the regular DGROUP selector, which doesn't allow access to the protected memory area set up by the operating system. |
| | **05** | **LNKDIR** | Microsoft C++ linker directives record. See the LNKDIR section for a complete description. |
| | **06** | **Big-endian** | The target for this OMF is a big-endian machine, as opposed to little-endian. "Big-endian" describes an architecture for which the most significant byte of a multibyte value has the smallest address. "Little-endian" describes an architecture for which the least significant byte of a multibyte value has the smallest address. |
| | **07** | **PRECOMP** | When the Microsoft symbol and type information for this object file is emitted, the directory entry for $$TYPES is to be emitted as sstPreComp instead of sstTypes. |
| | **08-FF** | | Reserved. |

***Note**: The presence of any unrecognized subtype causes the linker to generate a fatal error.*

**A1** **"New OMF" extension**

This comment class is now used solely to indicate the version of the symbolic debug information. If this comment class is not present, the version of the debug information produced is defined by the linker. For example, Microsoft LINK defaults to the oldest format of Microsoft symbol and type information.

This comment class was previously used to indicate that the obsolete method of communal representation through TYPDEF and EXTDEF pairs was not used and that COMDEF records were used instead. In current linkers, COMDEF records are always enabled, even without this comment record present.

The byte string is currently empty, but the planned future contents will be a version number (8-bit numeric field) followed by an ASCII character string indicating the symbol style. Values will be:

**n,'C','V'** **Microsoft symbol and type style**

**n,'D','X'** **AIX style**

**n,'H','L'** **IBM PM Debugger style**

**A2** **Link Pass Separator**

This record conveys information to the linker about the organization of the file. The value of the first byte of the commentary string specifies the comment subtype. Currently, a single subtype is defined:

**01** Indicates the start of records generated from Pass 2 of the linker. Additional bytes may follow, with their number determined by the Record Length field, but they will be ignored by the linker.

See the "Order of Records" section for information on which records must come before and after this comment.

> ***Warning:*** *It is assumed that this comment will not be present in a module whose MODEND record contains a program starting address.*

***Note**: This comment class may become obsolete with the advent of COMDAT records.*

**A3** **LIBMOD** Library module comment record. Ignored by the linker; used only by the librarian. See the LIBMOD section for a complete description.

**A4** **EXESTR** Executable string. See the EXESTR section for a complete description.

**A6** **INCERR** Incremental compilation error. See the INCERR section for a complete description.

**A7** **NOPAD** No segment padding. See the NOPAD section for a complete description.

**A8** **WKEXT** Weak Extern record. See the WKEXT section for a complete description.

**A9** **LZEXT** Lazy Extern record. See the LZEXT section for a complete description.

**DA** **Comment** For random comment.

**DB** **Compiler** For pragma comment(compiler); version number.

**DC** **Date** For pragma comment(date stamp).

**DD** **Timestamp** For pragma comment(timestamp).

**DF** **User** For pragma comment(user). Sometimes used for copyright notices.

| | | |
|---|---|---|
| **E9** | **Dependency file (Borland)** | Used to show the include files that were used to build this .OBJ file. |
| **FF** | **Command line (Microsoft QuickC)** | Shows the compiler options chosen. May be obsolete. This record is also used by Phoenix Technology Ltd. for library comments. |
| **C0H-FFH** | **-** | Comment classes within this range that are not otherwise used are reserved for user-defined comment classes. In general, the VENDEXT record replaces the need for new user-defined comment classes. |

***Notes***

*Microsoft LIB ignores the Comment Type field.*

*A COMENT record can appear almost anywhere in an object module. Only two restrictions apply:*

- *A COMENT record cannot be placed between a FIXUPP record and the LEDATA or LIDATA record to which it refers.*
- *A COMENT record cannot be the first or last record in an object module. (The first record must always be THEADR or LHEADR and the last must always be MODEND.)*

### Examples

The following three examples are typical COMENT records taken from an object module generated by the Microsoft C Compiler.

This first example is a language-translator comment:

```
      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000  88   07   00   00   00   4D   53   20   43   6E                                 .....MS Cn
```

Byte 00H contains 88H, indicating that this is a COMENT record.

Bytes 01-02H contain 0007H, the length of the remainder of the record.

Byte 03H (the Comment Type field ) contains 00H. Bit 7 (no purge) is set to 0, indicating that this COMENT record may be purged from the object module by a utility program that manipulates object modules. Bit 6 (no list) is set to 0, indicating that this comment need not be excluded from any listing of the module's contents. The remaining bits are all 0.

Byte 04H (the Comment Class field) contains 00H, indicating that this COMENT record contains the name of the language translator that generated the object module.

Bytes 05H through 08H contain the name of the language translator, Microsoft C.

Byte 09H contains the Checksum field, 6EH.

The second example contains the name of an object library to be searched by default when Microsoft LINK processes the object module containing this COMENT record:

```
      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000  88   09   00   00   9F   53   4C   49   42   46   50   10                  .....SLIBFP
```

Byte 04H (the Comment Class field) contains 9FH, indicating that this record contains the name of a library for LINK to use to resolve external references.

Bytes 05-0AH contain the library name, SLIBFP.  In this example, the name refers to the Microsoft C Compiler's floating-point function library, SLIBFP.LIB.

The last example indicates that Microsoft LINK should write the most recent format of Microsoft symbol and type information to the executable file.

```
      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000  88   06   00   00   A1   01   43   56   37                                         .....CV7
```

Byte 04H indicates the comment class, 0A1H.

Bytes 05-07H, which contain the comment string, are ignored by LINK.

## 88H IMPDEF—Import Definition Record (Comment Class A0, Subtype 01)

### Description

This record describes the imported names for a module.

### History

This comment class and subtype is a Microsoft extension added for OS/2 and Windows.

### Record Format

One import symbol is described; the subrecord format is

| 1 | 1 | <variable> | <variable> | 2 or <variable> |
|---|---|---|---|---|
| 01 | Ordinal Flag | Internal Name | Module Name | Entry Ident |

where:

**01** Identifies the subtype as an IMPDEF. It determines the form of the Entry Ident field.

**Ordinal Flag** Is a byte; if 0, the import is identified by name. If nonzero, it is identified by ordinal. It determines the form of the Entry Ident field.

**Internal Name** Is in *count, char* string format and is the name used within this module for the import symbol. This name will occur again in an EXTDEF record.

**Module Name** Is in *count, char* string format and is the name of the module (a DLL) that supplies an export symbol matching this import.

**Entry Ident** Is an ordinal or the name used by the exporting module for the symbol, depending upon the Ordinal Flag.

If this field is an ordinal (Ordinal Flag is nonzero), it is a 16-bit word. If this is a name and the first byte of the name is 0, then the exported name is the same as the imported name (in the Internal Name field). Otherwise, it is the imported name in *count, char* string format (as exported by Module Name).

***Note**: IMPDEF records are created by an import library utility, IMPLIB, which builds an "import library" from a module definition file or DLL.*

## 88H EXPDEF—Export Definition Record (Comment Class A0, Subtype 02)

### Description

This record describes the exported names for a module.

### History

This comment class and subtype is a Microsoft extension added for Microsoft C version 5.1.

### Record Format

One exported entry point is described; the subrecord format is

| 1 | 1 | <variable> | <variable> | 2 |
|---|---|---|---|---|
| 02 | Exported<br>Flag | Exported<br>Name | Internal<br>Name | Export<br>Ordinal |
| | | | | <conditional> |

where:

**02** Identifies the subtype as an EXPDEF.

**Exported Flag** Is a bit-significant 8-bit field with the following format:

<-------------------------------------------------------1 byte---------------------------------------------------->

| Ordinal<br>Bit | Resident<br>Name | No<br>Data | Parm Count |
|---|---|---|---|
| 1 | 1 | 1 | <------------------------5 bits-------------------------> |

**Ordinal Bit** Is set if the item is exported by ordinal; in this case the Export Ordinal field is present.

**Resident Name** Is set if the exported name is to be kept resident by the system loader; this is an optimization for frequently used items imported by name.

**No Data** Is set if the entry point does not use initialized data (either instanced or global).

**Parm Count** Is the number of parameter words. The Parm Count field is set to 0 for all but callgates to 16-bit segments.

**Exported Name** Is in *count, char* string format. Name to be used when the entry point is imported by name.

**Internal Name** Is in *count, char* string format. If the name length is 0, the internal name is the same as the Exported Name field. Otherwise, it is the name by which the entry point is known within this module. This name will appear as a PUBDEF or LPUBDEF name.

**Export Ordinal** Is present if the Ordinal Bit field is set; it is a 16-bit numeric field whose value is the ordinal used (must be nonzero).

***Note:*** *EXPDEFs are produced by many compilers when the keyword _export is used in a source file.*

## 88H INCDEF—Incremental Compilation Record (Comment Class A0, Subtype 03)

### Description

This record is used for incremental compilation. Every FIXUPP and LINNUM record following an INCDEF record will adjust all external index values and line number values by the appropriate delta. The deltas are cumulative if there is more than one INCDEF record per module.

### History

This comment class subtype is a Microsoft extension added for Microsoft QuickC version 2.0.

### Record Format

The subrecord format is:

| 1 | 2 | 2 | <variable> |
|---|---|---|---|
| 03 | EXTDEF Delta | LINNUM Delta | Padding |

The EXTDEF Delta and LINNUM Delta fields are signed.

Padding (zeros) is added by Microsoft QuickC to allow for expansion of the object module during incremental compilation and linking.

> ***Note**: Negative deltas are allowed.*

## 88H LNKDIR—Microsoft C++ Directives Record (Comment Class A0, Subtype 05)

### Description

This record is used by the compiler to pass directives and flags to the linker.

### History

This comment class and subtype is a Microsoft extension added for Microsoft C 7.0.

### Record Format

The subrecord format is:

| 1 | 1 | 1 | 1 |
|---|---|---|---|
| 05 | Bit Flags | Pseudocode Version | CodeView Version |

### Bit Flags Field

The format of the Bit Flags byte is:

| 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 (bits) |
|---|---|---|---|---|---|---|---|---|
| 05 | 0 | 0 | 0 | 0 | 0 | Run MPC | Omit CodeView $PUBLICS | New .EXE |

The low-order bit, if set, indicates that the linker should output the new .EXE format; this flag is ignored for all but linking of pseudocode (p-code) applications. (Pseudocode requires a segmented executable.)

The second low-order bit indicates that the linker should not output the $PUBLICS subsection of the Microsoft symbol and type (CodeView) information.

The third low-order bit indicates the need to run the Microsoft Make Pseudocode Utility (MPC) over the object file to enable creation of an executable file.

### Pseudocode Version Field

This is a one-byte field indicating the pseudocode interpreter version number.

### CodeView Version Field

This is a one-byte field indicating the CodeView version number.

> ***Note:*** *The presence of this record in an object module will indicate the presence of global symbols records. The linker will not emit a $PUBLICS section for those modules with this comment record and a $SYMBOLS section.*

## 88H LIBMOD—Library Module Name Record (Comment Class A3)

### Description

The LIBMOD comment record is used only by the librarian not by the linker. It gives the name of an object module within a library, allowing the librarian to preserve the source filename in the THEADR record and still identify the module names that make up the library. Since the module name is the base name of the .OBJ file that was built into the library, it may be completely different from the final library name.

### History

This comment class and subtype is a Microsoft extension added for LIB version 3.07 in version 5.0 of its macro assembler (MASM).

### Record Format

The subrecord format is:

| 1 | <variable> |
|---|---|
| A3 | Module Name |

The record contains only the ASCII string of the module name, in *count, char* format. The module name has no path and no extension, just the base of the module name.

> ***Notes***
>
> *Microsoft LIB adds a LIBMOD record when an .OBJ file is added to a library and strips the LIBMOD record when an .OBJ file is removed from a library, so typically this record exists only in .LIB files.*
>
> *There will be one LIBMOD record in the library file for each object module that was combined to build the library.*
>
> *IBM LINK386 ignores LIBMOD comment records.*

## 88H EXESTR—Executable String Record (Comment Class A4)

### Description

The EXESTR comment record implements these ANSI and XENIX/UNIX features in Microsoft C:

```
#pragma comment(exestr, <char-sequence>)
#ident string
```

### History

This comment class and subtype is a Microsoft extension added for Microsoft C 5.1.

### Record Format

The subrecord format is:

| 1 | <variable> |
|---|---|
| A4 | Arbitrary Text |

The linker will copy the text in the Arbitrary Text field byte for byte to the end of the executable file. The text will not be included in the program load image.

***Notes***

*If Microsoft symbol and type information is present, the text will not be at the end of the file but somewhere before so as not to interfere with the Microsoft symbol and type information signature.*

*There is no limit to the number of EXESTR comment records.*

## 88H INCERR—Incremental Compilation Error (Comment Class A6)

### Description

This comment record will cause the linker to terminate with a fatal error similar to "invalid object—error encountered during incremental compilation."

This behavior is useful when an incremental compilation fails and the user tries to link manually. The object module cannot be deleted, in order to preserve the base for the next incremental compilation.

### History

This comment class and subtype is a Microsoft extension added for Microsoft QuickC 2.0.

### Record Format

The subrecord format is:

| 1 | |
|---|---|
| A6 | No Fields |

## 88H NOPAD—No Segment Padding (Comment Class A7)

### Description

This comment record identifies a set of segments that are to be excluded from the padding imposed with the /PADDATA or /PADCODE options.

### History

This comment class and subtype is a Microsoft extension added for COBOL. It was added to Microsoft LINK to support MicroFocus COBOL version 1.2; it was added permanently in LINK version 5.11 to support Microsoft COBOL version 3.0.

### Record Format

The subrecord format is:

| 1 | 1 or 2 |
|---|---|
| A7 | SEGDEF Index |
|  | <--------------------repeated--------------------> |

The SEGDEF Index field is the standard OMF index type of 1 or 2 bytes. It may be repeated.

> ***Note:*** *IBM LINK386 ignores NOPAD comment records.*

## 88H WKEXT—Weak Extern Record (Comment Class A8)

### Description

This record marks a set of external names as "weak," and for every weak extern, the record associates another external name to use as the default resolution.

### History

This comment class and subtype is a Microsoft extension added for Microsoft Basic version 7.0. There is no construct in Microsoft Basic that produces it, but the record type is manually inserted into Microsoft Basic library modules.

The first user-accessible construct to produce a weak extern was added for Microsoft MASM version 6.0.

See the following "Notes" section for details on how and why this record is used in Microsoft's Basic and MASM.

### Record Format

The subrecord format is:

| 1 | 1 or 2 | 1 or 2 |
|---|---|---|
| A8 | Weak EXTDEF Index | Default Resolution EXTDEF Index |

<----------------repeated------------------------->

The Weak EXTDEF Index field is the 1- or 2-byte index to the EXTDEF of the extern that is weak.

The Default Resolution EXTDEF Index field is the 1- or 2-byte index to the EXTDEF of the extern that will be used to resolve the extern if no "stronger" link is found to resolve it.

#### *Notes*

*There are two ways to cancel the "weakness" of a weak extern; both result in the extern becoming a "strong" extern (the same as an EXTDEF). They are:*

- *If a PUBDEF for the weak extern is linked in*
- *If an EXTDEF for the weak extern is found in another module (including libraries)*

*If a weak extern becomes strong, then it must be resolved with a matching PUBDEF, just like a regular EXTDEF. If a weak extern has not become strong by the end of the linking process, then the default resolution is used.*

*If two weak externs for the same symbol in different modules have differing default resolutions, many linkers will emit a warning.*

*Weak externs do not query libraries for resolution; if an extern is still weak when libraries are searched, it stays weak and gets the default resolution. However, if a library module is linked in for other reasons (say, to resolve strong externs) and there are EXTDEFs for symbols that were weak, the symbols become strong.*

## Example

Assume that there is a weak extern for "var" with a default resolution name of "con". If there is a PUBDEF for "var" in some library module that would not otherwise be linked in, then the library module is not linked in, and any references to "var" are resolved to "con". However, if the library module is linked in for other reasons—for example, to resolve references to a strong extern named "bletch"— then "var" will be resolved by the PUBDEF from the library, not to the default resolution "con".

WKEXTs are best understood by explaining why they were added in the first place. The minimum BASIC run-time library in the past consisted of a large amount of code that was always linked in, even for the smallest program. Most of this code was never called directly by the user, but it was called indirectly from other routines in other libraries, so it had to be linked in to resolve the external references.

For instance, the floating-point library was linked in even if the user's program did not use floating-point operations, because the PRINT library routine contained calls to the floating-point library for support to print floating-point numbers.

The solution was to make the function calls between the libraries into weak externals, with the default resolution set to a small stub routine. If the user never used a language construct or feature that needed the additional library support, then no strong extern would be generated by the compiler and the default resolution (to the stub routine) would be used. However, if the user accessed the library's routines or used constructs that required the library's support, a strong extern would be generated by the compiler to cancel the effect of the weak extern, and the library module would be linked in. This required that the compiler know a lot about which libraries are needed for which constructs, but the resulting executable was much smaller.

***Note:***

*The construct in Microsoft MASM 6.0 that produces a weak extern is*

*EXTERN  var(con): byte*

*which makes "con" the default resolution for weak extern "var".*

## 88H LZEXT—Lazy Extern Record (Comment Class A9)

### Description

This record marks a set of external names as "lazy," and for every lazy extern, the record associates another external name to use as the default resolution.

### History

This comment class and subtype is a Microsoft extension added for Microsoft C 7.0, but was not implemented in the Microsoft C 7.0 linker.

### Record Format

The subrecord format is:

| 1 | 1 or 2 | 1 or 2 |
|---|---|---|
| A9 | Lazy EXTDEF Index | Default Resolution EXTDEF Index |

<---------------------repeated---------------------->

The Lazy EXTDEF Index field is the 1- or 2-byte index to the EXTDEF of the extern that is lazy.

The Default Resolution EXTDEF Index field is the 1- or 2-byte index to the EXTDEF of the extern that will be used to resolve the extern if no "stronger" link is found to resolve it.

***Notes***

*There are two ways to cancel the "laziness" of a lazy extern; both result in the extern becoming a "strong" extern (the same as an EXTDEF.) They are:*

- *If a PUBDEF for the lazy extern is linked in*
- *If an EXTDEF for the lazy extern is found in another module (including libraries)*

*If a lazy extern becomes strong, it must be resolved with a matching PUBDEF, just like a regular EXTDEF. If a lazy extern has not become strong by the end of the linking process, then the default resolution is used.*

*If two weak externs for the same symbol in different modules have differing default resolutions, many linkers will emit a warning.*

*Unlike weak externs, lazy externs do query libraries for resolution; if an extern is still lazy when libraries are searched, it stays lazy and gets the default resolution.*

*IBM LINK386 ignores LZEXT comment records.*

## 8AH or 8BH MODEND—Module End Record

### Description

The MODEND record denotes the end of an object module. It also indicates whether the object module contains the main routine in a program, and it can optionally contain a reference to a program's entry point.

### History

Record type 8BH is was added for 32-bit linkers; it has a Target Displacement field of 32 bits rather than 16 bytes.

### Record Format

| 1 | 2 | 1 | 1 | 1 or 2 | 1 or 2 | 2 or 4 | 1 |
|---|---|---|---|---|---|---|---|
| 8A<br>or<br>8B | Record Length | Module Type | End Data | Frame Datum | Target Datum | Target Displacement | Checksum |

<---------------Start Address subfield, conditional------------->

where:

#### Module Type Field

The Module Type byte is bit significant; its layout is

| MATTR | | Segment | | | | | |
|---|---|---|---|---|---|---|---|
| Main | Start | Bit | 0 | 0 | 0 | 0 | X |

<----------2 bits--------->

where:

**MATTR** Is a 2-bit field.

**Main** Is set if the module is a main program module.

**Start** Is set if the module contains a start address; if this bit is set, the field starting with the End Data byte is present and specifies the start address.

**Segment Bit** Is reserved. Only 0 is supported by MS-DOS and OS/2.

**X** Is set if the Start Address subfield contains a relocatable address reference that the linker must fix up. (The original Intel 8086 specification allows this bit to be 0, to indicate that the start address is an absolute physical address that is stored as a 16-bit frame number and 16-bit offset, but this capability is not supported by certain linkers. This bit should always be set; however, the value will be ignored.

**Start Address**

The Start Address subfield is present only if the Start bit in the Module Type byte is set. Its format is identical to the Fix Data, Frame Datum, Target Datum, and Target Displacement fields in a FIXUP subrecord of a FIXUPP record. Bit 2 of the End Data field, which corresponds to the P bit in a Fix Data field, must be 0. The Target Displacement field (if present) is a 4-byte field if the record type is 8BH and a 2-byte field otherwise. This value provides the initial contents of CS:(E)IP.

If overlays are used, the start address must be given in the MODEND record of the root module.

***Notes***

*A MODEND record can appear only as the last record in an object module.*

*It is assumed that the Link Pass Separator comment record (COMENT A2, subtype 01) will not be present in a module whose MODEND record contains a program starting address. If there are overlays, the linker needs to see the starting address on Pass 1 to define the symbol $$MAIN.*

**Example**

Consider the MODEND record of a simple HELLO.ASM program:

```
       0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000  8A   07   00   C1   00   01   01   00   00                                  AC.....
```

Byte 00H contains 8AH, indicating a MODEND record.

Bytes 01-02H contain 0007H, the length of the remainder of the record.

Byte 03H contains 0C1H (11000001B). Bit 7 is set to 1, indicating that this module is the main module of the program. Bit 6 is set to 1, indicating that a Start Address subfield is present. Bit 0 is set to 1, indicating that the address referenced in the Start Address subfield must be fixed up by the linker.

Byte 04H (End Data in the Start Address subfield) contains 00H. As in a FIXUPP record, bit 7 indicates that the frame for this fixup is specified explicitly, and bits 6 through 4 indicate that a SEGDEF index specifies the frame. Bit 3 indicates that the target reference is also specified explicitly, and bits 2 through 0 indicate that a SEGDEF index also specifies the target.

Byte 05H (Frame Datum in the Start Address subfield) contains 01H. This is a reference to the first SEGDEF record in the module, which in this example corresponds to the _TEXT segment. This reference tells the linker that the start address lies in the _TEXT segment of the module.

Byte 06H (Target Datum in the Start Address subfield) contains 01H. This also is a reference to the first SEGDEF record in the object module, which corresponds to the _TEXT segment. For example, Microsoft LINK uses the following Target Displacement field to determine where in the _TEXT segment the address lies.

Bytes 07-08H (Target Displacement in the Start Address subfield) contain 0000H. This is the offset (in bytes) of the start address.

Byte 09H contains the Checksum field, 0ACH.

## 8CH EXTDEF—External Names Definition Record

### Description

The EXTDEF record contains a list of symbolic external references—that is, references to symbols defined in other object modules. The linker resolves external references by matching the symbols declared in EXTDEF records with symbols declared in PUBDEF records.

### History

In the Intel specification and older linkers, the Type Index field was used as an index into TYPDEF records. This is no longer true; the field now encodes Microsoft symbol and type information (see Appendix 1 for details.) Many linkers ignore the old style TYPDEF.

### Record Format

| 1 | 2 | 1 | <String Length> | 1 or 2 | 1 |
|---|---|---|---|---|---|
| 8C | Record Length | String Length | External Name String | Type Index | Checksum |

This record provides a list of unresolved references, identified by name and with optional associated type information. The external names are ordered by occurrence jointly with the COMDEF and LEXTDEF records, and referenced by an index in other records (FIXUPP records); the name may not be null. Indexes start from 1.

String Length is a 1-byte field containing the length of the name field that follows it.

The Type Index field is encoded as an index field and contains debug information.

***Notes***

*For Microsoft compilers, all referenced functions of global scope and all referenced variables explicitly declared "extern" will generate an EXTDEF record.*

*Many linkers impose a limit of 1023 external names and restrict the name length to a value between 1 and 7FH.*

*Any EXTDEF records in an object module must appear before the FIXUPP records that reference them.*

*Resolution of an external reference is by name match (case sensitive) and symbol type match. The search looks for a matching name in the following sequence:*

1. *Searches PUBDEF and COMDEF records.*
2. *If linking a segmented executable, searches imported names (IMPDEF).*
3. *If linking a segmented executable and not a DLL, searches for an exported name (EXPDEF) with the same name—a self-imported alias.*
4. *Searches for the symbol name among undefined symbols. If the reference is to a weak extern, the default resolution is used. If the reference is to a strong extern, it's an undefined external, and the linker generates an error.*

*All external references must be resolved at link time (using the above search order). Even though the linker produces an executable file for an unsuccessful link session, an error bit is set in the header that prevents the loader from running the executable.*

**Example**

Consider this EXTDEF record generated by the Microsoft C Compiler:

```
      0  1  2  3   4   5  6  7   8  9  A   B  C   D   E  F
0000  8C 25 00 0A  5F  5F 61 63  72 74 75  73 65  64  00 05  .%.__acrtused..
0010  5F 6D 61 69  6E  00 05 5F  70 75 74  73 00  08  5F 5F  _main.._puts..__
0020  63 68 6B 73  74  6B 00 A5                              chkstk..
```

Byte 00H contains 8CH, indicating that this is an EXTDEF record.

Bytes 01-02H contain 0025H, the length of the remainder of the record.

Bytes 03-26H contain a list of external references. The first reference starts in byte 03H, which contains 0AH, the length of the name __acrtused. The name itself follows in bytes 04-0DH. Byte 0EH contains 00H, which indicates that the symbol's type is not defined by any TYPDEF record in this object module. Bytes 0F-26H contain similar references to the external symbols _main, _puts, and __chkstk.

Byte 27H contains the Checksum field, 0A5H.

## 90H or 91H PUBDEF—Public Names Definition Record

### Description

The PUBDEF record contains a list of public names. It makes items defined in this object module available to satisfy external references in other modules with which it is bound or linked.

The symbols are also available for export if so indicated in an EXPDEF comment record.

### History

Record type 91H was added for 32-bit linkers; it has a Public Offset field of 32 bits rather than 16 bits.

### Record Format

| 1 | 2 | 1 or 2 | 1 or 2 | 2 | 1 | <String Length> | 2 or 4 | 1 or 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 90 or 91 | Record Length | Base Group Index | Base Segment Index | Base Frame | String Length | Public Name String | Public Offset | Type Index | Checksum |
|  |  |  |  | <conditional> | <--------------------repeated---------------------> |  |  |  |  |

### Base Group, Base Segment, and Base Frame Fields

The Base Group and Base Segment fields contain indexes specifying previously defined SEGDEF and GRPDEF records. The group index may be 0, meaning that no group is associated with this PUBDEF record.

The Base Frame field is present only if the Base Segment field is 0, but the contents of the Base Frame field are ignored.

The Base Segment Index is normally nonzero and no Base Frame field is present.

According to the Intel 8086 specification, if both the segment and group indexes are 0, the Base Frame field contains a 16-bit paragraph (when viewed as a linear address); this may be used to define public symbols that are absolute. Absolute addressing is not fully supported by some linkers—it can be used for read-only access to absolute memory locations; however, writing to absolute memory locations may not work in some linkers.

### Public Name String, Public Offset, and Type Index Fields

The Public Name String field is in *count, char* format and cannot be null. The maximum length of a Public Name is 255 bytes.

The Public Offset field is a 2- or 4-byte numeric field containing the offset of the location referred to by the public name. This offset is assumed to lie within the group, segment, or frame specified in the Base Group, Base Segment, or Base Frame fields.

The Type Index field is encoded in index format and contains either debug information or 0. The use of this field is determined by the OMF producer and typically provides type information for the referenced symbol.

***Notes***

*All defined functions and initialized global variables generate PUBDEF records in most compilers. No PUBDEF record will be generated, however, for instantiated inline functions in C++.*

*Any PUBDEF records in an object module must appear after the GRPDEF and SEGDEF records to which they refer. Because PUBDEF records are not themselves referenced by any other type of object record, they are generally placed near the end of an object module.*

*Record type 90H uses 16-bit encoding of the Public Offset field, but it is zero-extended to 32 bits if applied to Use32 segments.*

**Examples**

The following two examples show PUBDEF records created by Microsoft's macro assembler, MASM.

The first example is the record for the statement:

```
PUBLIC  GAMMA
```

The PUBDEF record is:

```
      0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0000  90  0C  00  00  01  05  47  41  4D  4D  41  02  00  00  F9      .....GAMMA.....
```

Byte 00H contains 90H, indicating a PUBDEF record.

Bytes 01-02H contain 000CH, the length of the remainder of the record.

Bytes 03-04H represent the Base Group, Base Segment, and Base Frame fields. Byte 03H (the group index) contains 0, indicating that no group is associated with the name in this PUBDEF record. Byte 04H (the segment index) contains 1, a reference to the first SEGDEF record in the object module. This is the segment to which the name in this PUBDEF record refers.

Bytes 05-0AH represent the Public Name String field. Byte 05H contains 05H (the length of the name), and bytes 06-0AH contain the name itself, GAMMA.

Bytes 0B-0CH contain 0002H, the Public Offset field. The name GAMMA thus refers to the location that is offset two bytes from the beginning of the segment referenced by the Base Group, Base Segment, and Base Frame fields.

Byte 0DH is the Type Index field. The value of the Type Index field is 0, indicating that no data type is associated with the name GAMMA.

Byte 0EH contains the Checksum field, 0F9H.

The next example is the PUBDEF record for the following absolute symbol declaration:

```
          PUBLIC    ALPHA
ALPHA     EQU       1234h
```

The PUBDEF record is:

```
        0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
  0000  90   0E   00   00   00   00   00   05   41   4C   50   48   41   34   12   00 ...ALPHA4...
  0010  B1
```

Bytes 03-06H (the Base Group, Base Segment, and Base Frame fields) contain a group index of 0 (byte 03H) and a segment index of 0 (byte 04H). Because both the group index and segment index are 0, a frame number also appears in the Base Group, Base Segment, and Base Frame fields. In this instance, the frame number (bytes 05-06H) also happens to be 0.

Bytes 07-0CH (the Public Name String field) contain the name ALPHA, preceded by its length.

Bytes 0D-0EH (the Public Offset field) contain 1234H. This is the value associated with the symbol ALPHA in the assembler EQU directive. If ALPHA is declared in another object module with the declaration

```
EXTRN    ALPHA:ABS
```

any references to ALPHA in that object module are fixed up as absolute references to offset 1234H in frame 0. In other words, ALPHA would have the value 1234H.

Byte 0FH (the Type Index field) contains 0.

## 94H or 95H LINNUM—Line Numbers Record

### Description

The LINNUM record relates line numbers in source code to addresses in object code.

### History

Record type 95H is added for 32-bit linkers; allowing for 32-bit debugger style-specific information.

> ***Note:*** *For instantiated inline functions in Microsoft C 7.0, line numbers are output in LINSYM records with a reference to the function name instead of the segment.*

### Record Format

| 1 | 2 | 1 or 2 | 1 or 2 | <variable> | 1 |
|---|---|---|---|---|---|
| 94<br>or 95 | Record<br>Length | Base<br>Group | Base<br>Segment | Debugger<br>Style-specific<br>Information | Checksum |
|  |  |  |  | <---------repeated--------> |  |

### Base Group and Base Segment Fields

The Base Group and Base Segment fields contain indexes specifying previously defined GRPDEF and SEGDEF records.

***Notes***

*The debugger style-specific information is indicated by comment class A1.*

*Although the complete Intel 8086 specification allows the Base Group and Base Segment fields to refer to a group or to an absolute segment as well as to a relocatable segment, some linkers commonly restrict references in this field to relocatable segments.*

*The following discussion uses the Microsoft debugger style-specific information. The debugger style-specific information field in the LINNUM record is composed as follows:*

| 2 | 2 or 4 |
|---|---|
| Line<br>Number | Line Number<br>Offset |

<--------------------repeated-------------------->

*For Microsoft LINK LINNUM records, the Line Number field contains a 16-bit quantity, in the range 0 through 7FFF and is, as its name indicates, a line number in the source code. The Line Number Offset field contains a 2-byte or 4-byte quantity that gives the translated code or data's start byte in the program segment defined by the SEGDEF index (4 bytes if the record type is 95H; 2 bytes for type 94H).*

*The Line Number and Line Number Offset fields can be repeated, so a single LINNUM record can specify multiple line numbers in the same segment.*

*Line Number 0 has a special meaning: it is used for the offset of the first byte after the end of the function. This is done so that the length of the last line (in bytes) can be determined.*

*The source file corresponding to a line number group is determined from the THEADR record.*

*Any LINNUM records in an object module must appear after the SEGDEF records to which they refer. Because LINNUM records are not themselves referenced by any other type of object record, they are generally placed near the end of an object module.*

*Also see the INCDEF record which is used to maintain line numbers after incremental compilation.*

**Example**

The following LINNUM record was generated by the Microsoft C Compiler:

```
        0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000   94   0F   00   00   01   02   00   00   00   03   00   08   00   04   00   0F   ...........
0010   00   3C                                                                       ..
```

Byte 00H contains 94H, indicating that this is a LINNUM record.

Bytes 01-02H contain 000FH, the length of the remainder of the record.

Bytes 03-04H represent the Base Group and Base Segment fields. Byte 03H (the Base Group field) contains 00H, as it must. Byte 04H (the Base Segment field) contains 01H, indicating that the line numbers in this LINNUM record refer to code in the segment defined in the first SEGDEF record in this object module.

Bytes 05-06H (the Line Number field) contain 0002H, and bytes 07-08H (the Line Number Offset field) contain 0000H. Together, they indicate that source-code line number 0002 corresponds to offset 0000H in the segment indicated in the Base Group and Base Segment fields.

Similarly, the two pairs of Line Number and Line Number Offset fields in bytes 09-10H specify that line number 0003 corresponds to offset 0008H and that line number 0004 corresponds to offset 000FH.

Byte 11H contains the Checksum field, 3CH.

## 96H LNAMES—List of Names Record

### Description

The LNAMES record is a list of names that can be referenced by subsequent SEGDEF and GRPDEF records in the object module.

The names are ordered by occurrence and referenced by index from subsequent records. More than one LNAMES record may appear. The names themselves are used as segment, class, group, overlay, and selector names.

### History

This record has not changed since the original Intel 8086 OMF specification.

### Record Format

| 1 | 2 | 1 | <---String Length---> | 1 |
|---|---|---|---|---|
| 96 | Record Length | String Length | Name String | Checksum |

< --------------------repeated--------------------- >

Each name appears in *count, char* format, and a null name is valid. The character set is ASCII. Names can be up to 255 characters long.

***Notes***

*Any LNAMES records in an object module must appear before the records that refer to them. Because it does not refer to any other type of object record, an LNAMES record usually appears near the start of an object module.*

*Previous versions limited the name string length to 254 characters.*

### Example

The following LNAMES record contains the segment and class names specified in all three of the following full-segment definitions:

```
_TEXT     SEGMENT byte public 'CODE'
_DATA     SEGMENT word public 'DATA'
_STACK    SEGMENT para public 'STACK'
```

The LNAMES record is:

```
       0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0000   96  25  00  00  04  43  4F  44  45  04  44  41  54  41  05  53  .%...CODE.DATA.S.
0010   54  41  43  4B  05  5F  44  41  54  41  06  5F  53  54  41  43  TACK._DATA._STAC
0020   4B  05  5F  54  45  58  54  8B                                  K._TEXT.
```

Byte 00H contains 96H, indicating that this is an LNAMES record.

Bytes 01-02H contain 0025H, the length of the remainder of the record.

Byte 03H contains 00H, a zero-length name.

Byte 04H contains 04H, the length of the class name CODE, which is found in bytes 05-08H. Bytes 09-26H contain the class names DATA and STACK and the segment names _DATA, _STACK, and _TEXT, each preceded by 1 byte that gives its length.

Byte 27H contains the Checksum field, 8BH.

## 98H or 99H SEGDEF—Segment Definition Record

### Description

The SEGDEF record describes a logical segment in an object module. It defines the segment's name, length, and alignment, and the way the segment can be combined with other logical segments at bind, link, or load time.

Object records that follow a SEGDEF record can refer to it to identify a particular segment. The SEGDEF records are ordered by occurrence, and are referenced by segment indexes (starting from 1) in subsequent records.

### History

Record type 99H was added for 32-bit linkers: the Segment Length field is 32 bits rather than 16 bits. There is one newly implemented alignment type (page alignment), the B bit flag of the ACBP byte indicates a segment of 4 GB, and the P bit flag of the ACBP byte is the Use16/Use32 flag.

Starting with version 2.4, Microsoft LINK ignores the Overlay Name Index field. In versions 2.4 and later, command-line parameters to Microsoft LINK, rather than information contained in object modules, determine the creation of run-time overlays.

The length does not include COMDAT records. If selected, their size is added.

### Record Format

| 1 | 2 | <variable> | 2 or 4 | 1 or 2 | 1 or 2 | 1 or 2 | 1 |
|---|---|---|---|---|---|---|---|
| 98<br>or 99 | Record<br>Length | Segment<br>Attributes | Segment<br>Length | Segment<br>Name Index | Class<br>Name Index | Overlay<br>Name Index | Checksum |

### Segment Attributes Field

The Segment Attributes field is a variable-length field; its layout is:

| <---3 bits---> | <---3 bits---> | <---1 bit---> | <---1 bit---> | <---2 bytes----> | <----1 byte----> |
|---|---|---|---|---|---|
| A | C | B | P | Frame Number | Offset |
| | | | | <conditional> | <conditional> |

The fields have the following meanings:

**A** **Alignment**

A 3-bit field that specifies the alignment required when this program segment is placed within a logical segment. Its values are:

**0** Absolute segment.

**1** Relocatable, byte aligned.

**2** Relocatable, word (2-byte, 16-bit) aligned.

**3** Relocatable, paragraph (16-byte) aligned.

**4** Relocatable, aligned on a page boundary. (The original Intel 8086 specification defines a page to be 256 bytes. The IBM implementation of OMF uses a 4096-byte or 4K page size).

**5** Relocatable, aligned on a double word (4-byte) boundary.

**6** Not supported.

**7** Not defined.

The new values for 32-bit linkers are A=4 and A=5. Double word alignment is expected to be useful as 32-bit memory paths become more prevalent. Page-align is useful for certain hardware-defined items (such as page tables) and error avoidance.

> ***Note:*** *If A=0, the conditional Frame Number and Offset fields are present and indicate the starting address of the absolute segment. Microsoft LINK ignores the Offset field.*
>
> ***Conflict****: The original Intel 8086 specification included additional segment-alignment values not supported by Microsoft; alignment 5 now conflicts with the following Microsoft LINK extensions:*
>
> ***5*** *"unnamed absolute portion of memory address space"*
>
> ***6*** *"load-time locatable (LTL), paragraph aligned if not part of any group"*

**C** **Combination**

This 3-bit field describes how the linker can combine the segment with other segments. Under MS-DOS, segments with the same name and class can be combined in two ways: they can be concatenated to form one logical segment, or they can be overlapped. In the latter case, they have either the same starting address or the same ending address, and they describe a common area in memory. Values for the C field are:

**0** **Private.** Do not combine with any other program segment.

**1** **Reserved.**

**2** **Public.** Combine by appending at an offset that meets the alignment requirement.

**3** **Reserved.**

**4** Same as C=2 (public).

**5** **Stack.** Combine as for C=2. This combine type forces byte alignment.

**6** **Common.** Combine by overlay using maximum size.

**7** Same as C=2 (public).

***Conflict****: The original Intel 8086 specification lists C=1 as Common, not C=6.*

**B** **Big**

Used as the high-order bit of the Segment Length field. If this bit is set, the segment length value must be 0. If the record type is 98H and this bit is set, the segment is exactly 64K long. If the record type is 99H and this bit is set, the segment is exactly $2^{32}$ bytes or 4 GB long.

**P** This bit corresponds to the bit field for segment descriptors, known as the B bit for data segments and the D bit for code segments in Intel documentation.

If 0, then the segment is no larger than 64K (if data), and 16-bit addressing and operands are the default (if code). This is a Use16 segment.

If nonzero, then the segment may be larger than 64K (if data), and 32-bit addressing and operands are the default (if code). This is a Use32 segment.

***Note**: This is the only method for defining Use32 segments in the TIS OMF.*

### Segment Length Field

The Segment Length field is a 2- or 4-byte numeric quantity and specifies the number of bytes in this program segment. For record type 98H, the length can be from 0 to 64K; if a segment is exactly 64K, the segment length should be 0, and the B field in the ACBP byte should be 1. For record type 99H, the length can be from 0 to 4 GB; if a segment is exactly 4 GB in size, the segment length should be 0 and the B field in the ACBP byte should be 1.

### Segment Name Index, Class Name Index, Overlay Name Index Fields

The three name indexes (Segment Name Index, Class Name Index, and Overlay Name Index) refer to names that appeared in previous LNAMES record(s). The linker ignores the Overlay Name Index field. The full name of a segment consists of the segment and class names, and segments in different object modules are normally combined according to the A and C values if their full names are identical. These indexes must be nonzero, although the name itself may be null.

The Segment Name Index field identifies the segment with a name. The name need not be unique—other segments of the same name will be concatenated onto the first segment with that name. The name may have been assigned by the programmer, or it may have been generated by a compiler.

The Class Name Index field identifies the segment with a class name (such as CODE, FAR_DATA, or STACK). The linker places segments with the same class name into a contiguous area of memory in the run-time memory map.

The Overlay Name Index field identifies the segment with a run-time overlay. It is ignored by many linkers.

***Notes***

*Many linkers impose a limit of 255 SEGDEF records per object module.*

*Certain name/class combinations are reserved for debug information and have special significance to the linker, such as $$TYPES and $$SYMBOLS. See Appendix 1 for more information.*

***Conflicts:** The TIS-defined OMF has Use16/Use32 stored as the P bit of the ACBP field. The P bit does not specify the access for the segment. For Microsoft LINK the access is specified in the .DEF file.*

### Examples

The following examples of Microsoft assembler SEGMENT directives show the resulting values for the A field in the corresponding SEGDEF object record:

```
aseg    SEGMENT at 400h                ; A = 0
bseg    SEGMENT byte public 'CODE'     ; A = 1
cseg    SEGMENT para stack 'STACK'     ; A = 3
```

The following examples of assembler SEGMENT directives show the resulting values for the C field in the corresponding SEGDEF object record:

```
aseg    SEGMENT at 400H              ; C = 0
bseg    SEGMENT public 'DATA'        ; C = 2
cseg    SEGMENT stack 'STACK'        ; C = 5
dseg    SEGMENT common 'COMMON'      ; C = 6
```

In this first example, the segment is byte aligned:

```
      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
 0000 98   07   00   28   11   00   07   02   01   1E                                 ....(.....
```

Byte 00H contains 98H, indicating that this is a SEGDEF record.

Bytes 01-02H contain 0007H, the length of the remainder of the record.

Byte 03H contains 28H (00101000B), the ACBP byte. Bits 7-5 (the A field) contain 1 (001B), indicating that this segment is relocatable and byte aligned. Bits 4-2 (the C field) contain 2 (010B), which represents a public combine type. (When this object module is linked, this segment will be concatenated with all other segments with the same name.) Bit 1 (the B field) is 0, indicating that this segment is smaller than 64K. Bit 0 (the P field) is ignored and should be 0, as it is here.

Bytes 04-05H contain 0011H, the size of the segment in bytes.

Bytes 06-08H index the list of names defined in the module's LNAMES record. Byte 06H (the Segment Name Index field) contains 07H, so the name of this segment is the seventh name in the LNAMES record. Byte 07H (the Class Name Index field) contains 02H, so the segment's class name is the second name in the LNAMES record. Byte 08H (the Overlay Name Index field) contains 1, a reference to the first name in the LNAMES record. (This name is usually null, as MS-DOS ignores it anyway.)

Byte 09H contains the Checksum field, 1EH.

The second SEGDEF record declares a word-aligned segment. It differs only slightly from the first.

```
      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
 0000 98   07   00   48   0F   00   05   03   01   01                                 .. H......
```

Bits 7-5 (the A field) of byte 03H (the ACBP byte) contain 2 (010B), indicating that this segment is relocatable and word aligned.

Bytes 04-05H contain the size of the segment, 000FH.

Byte 06H (the Segment Name Index field) contains 05H, which refers to the fifth name in the previous LNAMES record.

Byte 07H (the Class Name Index field) contains 03H, a reference to the third name in the LNAMES record.

## 9AH GRPDEF—Group Definition Record

### Description

This record causes the program segments identified by SEGDEF records to be collected together (grouped). For OS/2, the segments are combined into a logical segment that is to be addressed through a single selector. For MS-DOS, the segments are combined within the same 64K frame in the run-time memory map.

### History

The special group name "FLAT" has been added for 32-bit linkers.

### Record Format

| 1 | 2 | 1 or 2 | 1 | 1 or 2 | 1 |
|---|---|---|---|---|---|
| 9A | Record Length | Group Name Index | FF Index | Segment Definition | Checksum |

<--------- repeated --------->

### Group Name Field

The Group Name field contains an index into a previously defined LNAMES name and must be nonzero.

Groups from different object modules are combined if their names are identical.

### Group Components

The group's components are segments, specified as indexes into previously defined SEGDEF records.

The first byte of each group component is a type field for the remainder of the component. Certain linkers require a type value of FFH and always assume that the component contains a segment index value. See the "Notes" section below for other types defined by Intel.

The component fields are usually repeated so that all the segments constituting a group can be included in one GRPDEF record.

#### *Notes*

*Most linkers impose a limit of 31 GRPDEF records in a single object module and limit the total number of group definitions across all object modules to 31.*

*This record is frequently followed by a THREAD FIXUPP record.*

*A common grouping using the Group Definition Record would be to group the default data segments.*

*Most linkers do special handling of the pseudo-group name FLAT. All address references to this group are made as offsets from the Virtual Zero Address, which is the start of the memory image of the executable.*

*The additional group component types defined by the original Intel 8086 specification and the fields that follow them are:*

| | |
|---|---|
| ***FE*** | *External Index* |
| ***FD*** | *Segment Name Index, Class Name Index, Overlay Name Index* |
| ***FB*** | *LTL Data field, Maximum Group Length, Group Length* |
| ***FA*** | *Frame Number, Offset* |

*None of these types are supported by Microsoft LINK or IBM LINK386.*

**Example**

The example of a GRPDEF record below corresponds to the following assembler directive:

```
tgroup  GROUP seg1,seg2,seg3
```

The GRPDEF record is:

```
       0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000   9A   08   00   06   FF   01   FF   02   FF   03   55                            .....U
```

Byte 00H contains 9AH, indicating that this is a GRPDEF record.

Bytes 01-02H contain 0008H, the length of the remainder of the record.

Byte 03H contains 06H, the Group Name Index field. In this instance, the index number refers to the sixth name in the previous LNAMES record in the object module. That name is the name of the group of segments defined in the remainder of the record.

Bytes 04-05H contain the first of three group component descriptor fields. Byte 04H contains the required 0FFH, indicating that the subsequent field is a segment index. Byte 05H contains 01H, a segment index that refers to the first SEGDEF record in the object module. This SEGDEF record declared the first of three segments in the group.

Bytes 06-07H represent the second group component descriptor, this one referring to the second SEGDEF record in the object module.

Similarly, bytes 08-09H are a group component descriptor field that references the third SEGDEF record.

Byte 0AH contains the Checksum field, 55H.

## 9CH or 9DH FIXUPP—Fixup Record

### Description

The FIXUPP record contains information that allows the linker to resolve (fix up) and eventually relocate references between object modules. FIXUPP records describe the LOCATION of each address value to be fixed up, the TARGET address to which the fixup refers, and the FRAME relative to which the address computation is performed.

### History

Record type 9DH was added for 32-bit linkers; it has a Target Displacement field of 32 bits rather than 16 bits, and the Location field of the Locat word has been extended to 4 bits (using the previously unused higher order S bit) to allow new LOCATION values of 9, 11, and 13.

### Record Format

| 1 | 2 | < ----------------- from Record Length-----------------> | 1 |
|---|---|---|---|
| 9C<br>or 9D | Record<br>Length | THREAD subrecord or<br>FIXUP subrecord | Checksum |
|  |  | < -------------------------- repeated ---------------------> |  |

Each subrecord in a FIXUPP object record either defines a thread for subsequent use, or refers to a data location in the nearest previous LEDATA or LIDATA record. The high-order bit of the subrecord determines the subrecord type: if the high-order bit is 0, the subrecord is a THREAD subrecord; if the high-order bit is 1, the subrecord is a FIXUP subrecord. Subrecords of different types can be mixed within one object record.

Information that determines how to resolve a reference can be specified explicitly in a FIXUP subrecord, or it can be specified within a FIXUP subrecord by a reference to a previous THREAD subrecord. A THREAD subrecord describes only the method to be used by the linker to refer to a particular target or frame. Because the same THREAD subrecord can be referenced in several subsequent FIXUP subrecords, a FIXUPP object record that uses THREAD subrecords may be smaller than one in which THREAD subrecords are not used.

THREAD subrecords can be referenced in the same object record in which they appear and also in subsequent FIXUPP object records.

### THREAD Subrecord

There are four FRAME threads and four TARGET threads; not all need be defined, and they can be redefined by later THREAD subrecords in the same or later FIXUPP object records. The FRAME threads are used to specify the Frame Datum field in a later FIXUP subrecord; the TARGET threads are used to specify the Target Datum field in a later FIXUP subrecord.

A THREAD subrecord does not require that a previous LEDATA or LIDATA record occur.

The layout of the THREAD subrecord is as follows:

| <-------------------------------------- 1 byte | | | | --------------------------------------- > | <----------- 1 or 2 bytes ------------> |
|---|---|---|---|---|---|
| 0 | D | 0 | Method | Thred | Index |
| 1 | 1 | 1 | 3 | 2 (bits) | <------------conditional-----------> |

where:

**0** The high-order bit is 0 to indicate that this is a THREAD subrecord.

**D** Is 0 for a TARGET thread, 1 for a FRAME thread.

**Method** Is a 3-bit field.

For TARGET threads, only the lower two bits of the field are used; the high-order bit of the method is derived from the P bit in the Fix Data field of FIXUP subrecords that refer to this thread. (The full list of methods is given here for completeness.) This field determines the kind of index required to specify the Target Datum field.

**T0** Specified by a SEGDEF index.

**T1** Specified by a GRPDEF index.

**T2** Specified by a EXTDEF index.

**T3** Specified by an explicit frame number (not supported by Microsoft LINK or IBM LINK386).

**T4** Specified by a SEGDEF index only; the displacement in the FIXUP subrecord is assumed to be 0.

**T5** Specified by a GRPDEF index only; the displacement in the FIXUP subrecord is assumed to be 0.

**T6** Specified by a EXTDEF index only; the displacement in the FIXUP subrecord is assumed to be 0.

The index type specified by the TARGET thread method is encoded in the Index field.

For FRAME threads, the Method field determines the Frame Datum field of subsequent FIXUP subrecords that refer to this thread. Values for the Method field are:

**F0** The FRAME is specified by a SEGDEF index.

**F1** The FRAME is specified by a GRPDEF index.

**F2** The FRAME is specified by a EXTDEF index. Microsoft LINK and IBM LINK386 determine the FRAME from the external name's corresponding PUBDEF record in another object module, which specifies either a logical segment or a group.

**F3** Invalid. (The FRAME is identified by an explicit frame number; this is not supported by any current linker.)

**F4** The FRAME is determined by the segment index of the previous LEDATA or LIDATA record (that is, the segment in which the location is defined).

**F5** The FRAME is determined by the TARGET's segment, group, or external index.

**F6** Invalid.

***Note:*** *The Index field is present for FRAME methods F0, F1, and F2 only.*

**Thred** A 2-bit field that determines the thread number (0 through 3, for the four threads of each kind).

**Index** A conditional field that contains an index value that refers to a previous SEGDEF, GRPDEF, or EXTDEF record. The field is present only if the thread method is 0, 1, or 2. (If method 3 were supported by the linker, the Index field would contain an explicit frame number.)

### FIXUP Subrecord

A FIXUP subrecord gives the how/what/why/where/who information required to resolve or relocate a reference when program segments are combined or placed within logical segments. It applies to the nearest previous LEDATA or LIDATA record, which must be defined before the FIXUP subrecord. The FIXUP subrecord is as follows:

| 2 | 1 | 1 or 2 | 1 or 2 | 2 or 4 |
|---|---|---|---|---|
| Locat | Fix<br>Data | Frame<br>Datum | Target<br>Datum | Target<br>Displacement |
|  | <conditional> | <conditional> | <conditional> |  |

where the Locat field has an unusual format. Contrary to the usual byte order in Intel data structures, the most significant bits of the Locat field are found in the low-order byte, rather than the high-order byte, as follows:

| < ---------- low-order byte -------- > |  |  | < --------high-order byte--------- > |
|---|---|---|---|
| 1 | M | Location | Data Record Offset |
| 1 | 1 | 4 | 10 (bits) |

where:

**1** The high-order bit of the low-order byte is set to indicate a FIXUP subrecord.

**M** Is the mode; M=1 for segment-relative fixups, and M=0 for self-relative fixups.

**Location** Is a 4-bit field that determines what type of LOCATION is to be fixed up:

**0** Low-order byte (8-bit displacement or low byte of 16-bit offset).

**1** 16-bit offset.

**2** 16-bit base—logical segment base (selector).

**3** 32-bit Long pointer (16-bit base:16-bit offset).

**4** High-order byte (high byte of 16-bit offset). Microsoft LINK and IBM LINK386 do not support this type.

**5** 16-bit loader-resolved offset, treated as Location=1.

***Conflict**: The PharLap implementation of OMF uses Location=5 to indicate a 32-bit offset, where IBM and Microsoft use Location=9.*

**6** Not defined, reserved.

***Conflict**: The PharLap implementation of OMF uses Location=6 to indicate a 48-bit pointer (16-bit base:32-bit offset), where IBM and Microsoft use Location=11.*

**7** Not defined, reserved.

**8** Not defined, reserved.

**9** 32-bit offset.

**10** Not defined, reserved.

**11** 48-bit pointer (16-bit base:32-bit offset).

**12** Not defined, reserved.

**13** 32-bit loader-resolved offset, treated as Location=9 by the linker.

**Data Record Offset** Indicates the position of the LOCATION to be fixed up in the LEDATA or LIDATA record immediately preceding the FIXUPP record. This offset indicates either a byte in the Data Bytes field of an LEDATA record or a data byte in the Content field of a Data Block field in an LIDATA record.

The Fix Data bit layout is

| F | Frame | T | P | Targt |
|---|---|---|---|---|
| 1 | 3 | 1 | 1 | 2 (bits) |

and is interpreted as follows:

**F** If F=1, the FRAME is given by a FRAME thread whose number is in the Frame field (modulo 4). There is no Frame Datum field in the subrecord.

If F=0, the FRAME method (in the range F0 to F5) is explicitly defined in this FIXUP subrecord. The method is stored in the Frame field.

**Frame** A 3-bit numeric field, interpreted according to the F bit. The Frame Datum field is present and is an index field for FRAME methods F0, F1, and F2 only.

**T** If T=1, the TARGET is defined by a TARGET thread whose thread number is given in the 2-bit Targt field. The Targt field contains a number between 0 and 3 that refers to a previous THREAD subrecord containing the TARGET method. The P bit, combined with the two low-order bits of the Method field in the THREAD subrecord, determines the TARGET method.

If T=0, the TARGET is specified explicitly in this FIXUP subrecord. In this case, the P bit and the Targt field can be considered a 3-bit field analogous to the Frame field.

**P** Determines whether the Target Displacement field is present.

If P=1, there is no Target Displacement field.

If P=0, the Target Displacement field is present. It is a 4-byte field if the record type is 9DH; it is a 2-byte field otherwise.

**Targt** A 2-bit numeric field, which gives the lower two bits of the TARGET method (if T=0) or gives the TARGET thread number (if T=1).

### Frame Datum, Target Datum, and Target Displacement Fields

The Frame Datum field is an index field that refers to a previous SEGDEF, GRPDEF, or EXTDEF record, depending on the FRAME method.

Similarly, the Target Datum field contains a segment index, a group index, or an external name index, depending on the TARGET method.

The Target Displacement field, a 16-bit or 32-bit field, is present only if the P bit in the Fix Data field is set to 0, in which case the Target Displacement field contains the offset used in methods 0, 1, and 2 of specifying a TARGET.

***Notes***

*FIXUPP records are used to fix references in the immediately preceding LEDATA, LIDATA, or COMDAT record.*

*The Frame field is the translator's way of telling the linker the contents of the segment register used for the reference; the TARGET is the item being referenced whose address was not completely resolved by the translator. In protected mode, the only legal segment register values are selectors; every segment and group of segments is mapped through some selector and addressed by an offset within the underlying memory defined by that selector.*

### Examples

For good examples of the usage of the FIXUP record, consult *The MS-DOS Encyclopedia*.

## A0H or A1H LEDATA—Logical Enumerated Data Record

### Description

This record provides contiguous binary data—executable code or program data—that is part of a program segment. The data is eventually copied into the program's executable binary image by the linker.

The data bytes may be subject to relocation or fixing up as determined by the presence of a subsequent FIXUPP record, but otherwise they require no expansion when mapped to memory at run time.

### History

Record type A1H was added for 32-bit linkers; it has an Enumerated Data Offset field of 32 bits rather than 16 bits.

### Record Format

| 1 | 2 | 1 or 2 | 2 or 4 | <from Record Length> | 1 |
|---|---|---|---|---|---|
| A0<br>or A1 | Record<br>Length | Segment<br>Index | Enumerated<br>Data Offset | Data<br>Bytes | Checksum |

### Segment Index Field

The Segment Index field must be nonzero and is the index of a previously defined SEGDEF record. This is the segment into which the data in this LEDATA record is to be placed.

### Enumerated Data Offset Field

The Enumerated Data Offset field is either a 2- or 4-byte field (depending on the record type) that determines the offset at which the first data byte is to be placed relative to the start of the SEGDEF segment. Successive data bytes occupy successively higher locations.

### Data Bytes Field

The maximum number of data bytes is 1024, so that a FIXUPP Location field, which is 10 bits, can reference any of these data bytes.

***Notes***

*Record type A1H has the offset stored as a 32-bit value. Record type A0H encodes the offset value as a 16-bit numeric field (zero-extended if applied to a Use32 segment).*

*If an LEDATA record requires a fixup, a FIXUPP record must immediately follow the LEDATA record.*

*Code for functions is output in LEDATA records currently. The segment for code is usually named _TEXT (or* module*_TEXT, depending on the memory model), unless #pragma alloc_text is used to specify a different code segment for the specified functions.*

*For instantiated functions in Microsoft C++, code will simply be output in COMDAT records that refer to the function and identify the function's segment.*

*Data, usually generated by initialized variables (global or static), is output in LEDATA/LIDATA records referring to either a data segment or, possibly, a segment created for a based variable.*

### Example

The following LEDATA record contains a simple text string:

```
        0    1    2    3    4    5    6   7    8   9   A    B    C    D    E    F
0000   A0   13   00   02   00   00   48  65   6C  6C  6F   2C   20   77   6F   72  ...... Hello, wor
0010   6C   64   0D   0A   24   A8                                                 ld..$.
```

Byte 00H contains 0A0H, which identifies this as an LEDATA record.

Bytes 01-02H contain 0013H, the length of the remainder of the record.

Byte 03H (the Segment Index field) contains 02H, a reference to the second SEGDEF record in the object module.

Bytes 04-05H (the Enumerated Data Offset field) contain 0000H. This is the offset, from the base of the segment indicated by the Segment Index field, at which the data in the Data Bytes field will be placed when the program is linked. Of course, this offset is subject to relocation by the linker because the segment declared in the specified SEGDEF record may be relocatable and may be combined with other segments declared in other object modules.

Bytes 06-14H (the Data Bytes field) contain the actual data.

Byte 15H contains the Checksum field, 0A8H.

## A2H or A3H LIDATA—Logical Iterated Data Record

### Description

Like the LEDATA record, the LIDATA record contains binary data—executable code or program data. The data in an LIDATA record, however, is specified as a repeating pattern (iterated), rather than by explicit enumeration.

The data in an LIDATA record can be modified by the linker if the LIDATA record is followed by a FIXUPP record, although this is not recommended.

### History

Record type A3H was added for 32-bit linkers; it has Iterated Data Offset and Repeat Count fields of 32 bits rather than 16 bits.

### Record Format

| 1 | 2 | 1 or 2 | 2 or 4 | <from Record Length> | 1 |
|---|---|---|---|---|---|
| A2 or A3 | Record Length | Segment Index | Iterated Data Offset | Data Block | Checksum |
|  |  |  |  | <----------repeated--------> |  |

### Segment Index and Iterated Data Offset Fields

The Segment Index and Iterated Data Offset fields (2 or 4 bytes) are the same as for an LEDATA record. The index must be nonzero. This indicates the segment and offset at which the data in this LIDATA record is to be placed when the program is loaded.

### Data Block Field

The data blocks have the following form:

| 2 or 4 | 2 | <from Block Count> |
|---|---|---|
| Repeat Count | Block Count | Content |

### Repeat Count Field

The Repeat Count field is a 16-bit or 32-bit value that determines the number of times the Content field is to be repeated. The Repeat Count field is 32 bits only if the record type is A3H.

> ***Conflict:*** *The PharLap implementation of OMF uses a 16-bit repeat count even in 32-bit records.*

### Block Count Field

The Block Count field is a 16-bit word whose value determines the interpretation of the Content field, as follows:

**0** Indicates that the Content field that follows is a 1-byte count value followed by count data bytes. The data bytes will be mapped to memory, repeated as many times as are specified in the Repeat Count field.

**!= 0** Indicates that the Content field that follows is composed of one or more Data Block fields. The value in the Block Count field specifies the number of Data Block fields (recursive definition).

***Notes***

*The Microsoft C Compiler generates LIDATA records for initialized data. For example:*

```
static int a[100] = { 1, };
```

*A FIXUPP record may occur after the LIDATA record; however, the fixup is applied before the iterated data block is expanded. It is a translator error for a fixup to reference any of the Count fields.*

**Example 1**

```
02 00 02 00 03 00 00 00 02 40 41 02 00 00 00 02 50 51
```

is an iterated data block with 16-bit repeat counts that expands to:

```
40 41 40 41 40 41 50 51 50 51 40 41 40 41 40 41 50 51 50 51
```

Here, the outer data block has a repeat count of 2 and a block count of 2 (which means to repeat twice the result of expanding the two inner data blocks). The first inner data block has repeat count = 3, block count = 0. The content is 2 bytes of data (40 41); the repeat count expands the data to a string of 6 bytes. The second (and last) inner data block has a repeat count = 2, block count = 0, content 2 bytes of data (50 51). This expands to 4 bytes, which is concatenated with the 6 bytes from the first inner data block. The resulting 10 bytes are then expanded by 2 (the repeat count of the outer data block) to form the 20-byte sequence illustrated.

**Example 2**

This sample LIDATA record corresponds to the following assembler statement, which declares a 10-element array containing the strings ALPHA and BETA:

```
db      10 dup('ALPHA','BETA')
```

The LIDATA record is

```
        0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0000    A2  1B  00  01  00  00  0A  00  02  00  01  00  00  00  05  41  ...............A
0010    4C  50  48  41  01  00  00  00  04  42  45  54  41  A9          LPHA.....BETA.
```

Byte 00H contains 0A2H, identifying this as an LIDATA record.

Bytes 01-02H contain 1BH, the length of the remainder of the record.

Byte 03H (the Segment Index field) contains 01H, a reference to the first SEGDEF record in this object module, indicating that the data declared in this LIDATA record is to be placed into the segment described by the first SEGDEF record.

Bytes 04-05H (the Iterated Data Offset field) contain 0000H, so the data in this LIDATA record is to be located at offset 0000H in the segment designated by the segment.

Bytes 06-1CH represent an iterated data block:

- Bytes 06-07H contain the repeat count, 000AH, which indicates that the Content field of this iterated data block is to be repeated 10 times.
- Bytes 08-09H (the block count for this iterated data block) contain 0002H, which indicates that the Content field of this iterated data block (bytes 0A-1CH) contains two nested iterated data block fields (bytes 0A-13H and bytes 14-1CH).
- Bytes 0A-0BH contain 0001H, the repeat count for the first nested iterated data block. Bytes 0C-0DH contain 0000H, indicating that the Content field of this nested iterated data block contains data rather than more nested iterated data blocks. The Content field (bytes 0E-13H) contains the data; byte 0EH contains 05H, the number of subsequent data bytes; and bytes 0F-13H contain the actual data (the string ALPHA).
- Bytes 14-1CH represent the second nested iterated data block, which has a format similar to that of the block in bytes 0A-13H. This second nested iterated data block represents the 4-byte string BETA.
- Byte 1DH is the Checksum field, 0A9H.

## B0H COMDEF—Communal Names Definition Record

### Description

The COMDEF record is an extension to the basic set of 8086 object record types. It declares a list of one or more communal variables (uninitialized static data or data that may match initialized static data in another compilation unit).

The size of such a variable is the maximum size defined in any module naming the variable as communal or public. The placement of communal variables is determined by the data type using established conventions (noted below).

### History

The COMDEF record was introduced by version 3.5 of Microsoft LINK.

This record is also used by Borland to support unique instantiatians of virtual tables, “out of inlines” and various thunks the C++ compiler generates. Borland’s documentation refers to this record as the VIRDEF record.

### Record Format

| 1 | 2 | 1 | <String Length> | 1 or 2 | 1 | <from Data Type> | 1 |
|---|---|---|---|---|---|---|---|
| B0 | Record Length | String Length | Communal Name | Type Index | Data Type | Communal Length | Checksum |

<-----------------------------------repeated------------------------------------------>

### Communal Name Field

The name is in *count, char* format, and the name may be null. NEAR and FAR communals from different object files are matched at bind or link time if their names agree; the variable's size is the maximum of the sizes specified (subject to some constraints, as documented below).

### Type Index Field

This field encodes symbol information; it is parsed as an index field (1 or 2 bytes) and is not inspected by linkers.

### Data Type and Communal Length Fields

The Data Type field indicates the contents of the Communal Length field. All Data Type values for NEAR data indicate that the Communal Length field has only one numeric value: the amount of memory to be allocated for the communal variable. All Data Type values for FAR data indicate that the Communal Length field has two numeric values: the first is the number of elements, and the second is the element size.

The Data Type field is one of the following hexadecimal values:

**1 to 5FH** Interpreted as a Borland segment index.

**61H** FAR data; the length is specified as the number of the elements followed by the element size in bytes

**62H** NEAR data; the length is specified as the number of bytes

The Communal Length field is a single numeric field or a pair of numeric fields (as specified by the Data Type field), encoded as follows:

| Value Range | Number<br>of Bytes | Allocation |
|---|---|---|
| 0 through 128 | 1 | This byte contains the value |
| 0 to 64K-1 | 3 | First byte is 81H, followed by a 16-bit word whose value is used |
| 0 to 16 MB-1 | 4 | First byte is 84H, followed by a 3-byte value |
| -2 GB-1 to 2 GB-1 | 5 | First byte is 88H, followed by a 4-byte value |

Groups of Communal Name, Type Index, Data Type, and Communal Length fields can be repeated so that more than one communal variable can be declared in the same COMDEF record.

***Notes***

*If a public or exported symbol with the same name is found in another module to which this module is bound or linked, certain linkers will give the error "symbol defined more than once."*

*Communal variables cannot be resolved to dynamic links (that is, imported symbols).*

*The records are ordered by occurrence, together with the items named in EXTDEF and LEXTDEF records (for reference in FIXUP subrecords).*

*In older linkers, object modules that contain COMDEF records are required to also contain one COMENT record with comment class 0A1H, indicating that Microsoft extensions to the Intel object record specification are included in the object module. This COMENT record is no longer required; linkers always interpret COMDEF records.*

**Example**

The following COMDEF record was generated by Microsoft C Compiler version 4.0 for these public variable declarations:

```
int     var;                         /* 2-byte integer */
char    var2[32768];                 /* 32768-byte array */
char    far var3[10][2][20];         /* 400-byte array */
```

The COMDEF record is:

```
          0   1   2   3   4   5   6   7   8   9   A   B   C   D   E    F
0000  B0  20  00  04  5F  66  6F  6F  00  62  02  05  5F  66  6F  6F  .  .._var.b.._var
0010  32  00  62  81  00  80  05  5F  66  6F  6F  33  00  61  81  90  2.b...._var3.a..
0020  01  01  99                                                      ...
```

Byte 00H contains 0B0H, indicating that this is a COMDEF record.

Bytes 01-02H contain 0020H, the length of the remainder of the record.

Bytes 03-0AH, 0B-15H, and 16-21H represent three declarations for the communal variables var, var2, and var3. The Microsoft C compiler prepends an underscore to each of the names declared in the source code, so the symbols represented in this COMDEF record are _var, _var2, and _var3.

Byte 03H contains 04H, the length of the first Communal Name field in this record. Bytes 04-07H contain the name itself (_var). Byte 08H (the Type Index field) contains 00H, as required. Byte 09H (the Data Type field) contains 62H, indicating that this is a NEAR variable. Byte 0AH (the Communal Length field) contains 02H, the size of the variable in bytes.

Byte 0BH contains 05H, the length of the second Communal Name field. Bytes 0C-10H contain the name _var2. Byte 11H is the Type Index field, which again contains 00H, as required. Byte 12H (the Data Type field) contains 62H, indicating that _var2 is a NEAR variable.

Bytes 13-15H (the Communal Length field) contain the size in bytes of the variable. The first byte of the Communal Length field (byte 13H) is 81H, indicating that the size is represented in the subsequent two bytes of data—bytes 14-15H, which contain the value 8000H.

Bytes 16-1BH represent the Communal Name field for _var3, the third communal variable declared in this record. Byte 1CH (the Type Index field) again contains 00H as required. Byte 1DH (the Data Type field) contains 61H, indicating that this is a FAR variable. This means the Communal Length field is formatted as a Number of Elements field (bytes 1E-20H, which contain the value 0190H) and an Element Size field (byte 21H, which contains 01H). The total size of this communal variable is thus 190H times 1, or 400 bytes.

Byte 22H contains the Checksum field, 99H.

## B2H or B3H BAKPAT—Backpatch Record

### Description

This record is for backpatches to LOCATIONs that cannot be conveniently handled by a FIXUPP record at reference time (for example, forward references in a one-pass compiler). It is essentially a specialized fixup.

### History

Record type B2H is a Microsoft extension that was added for QuickC version 1.0. Record type B3H is the 32-bit equivalent: the Offset and Value fields are 32 bits rather than 16 bits.

### Record Format

| 1 | 2 | 1 or 2 | 1 | 2 or 4 | 2 or 4 | 1 |
|---|---|---|---|---|---|---|
| B2<br>or B3 | Record<br>Length | Segment<br>Index | Location<br>Type | Offset | Value | Checksum |

<---------------repeated------------->

### Segment Index Field

Segment index to which all "backpatch" FIXUPP records are to be applied. Note that, in contrast to FIXUPP records, these records do not need to follow the data record to be fixed up. Hence, the segment to which the backpatch applies must be specified explicitly.

### Location Type Field

Type of LOCATION to be patched; the only valid values are:

| | |
|---|---|
| **0** | 8-bit low-order byte |
| **1** | 16-bit offset |
| **2** | 32-bit offset, record type B3H only (not supported yet) |
| **9** | 32-bit offset, per the IBM implementation of OMF |

### Offset and Value Fields

These fields are 32 bits for record type B3H, and 16 bits for B2H.

The Offset field specifies the LOCATION to be patched (as an offset into the SEGDEF record whose index is Segment Index).

The associated Value field is added to the LOCATION being patched (unsigned addition, ignoring overflow). The Value field is a fixed length (16 bits or 32 bits, depending on the record type) to make object-module processing easier.

#### ***Notes***

*BAKPAT records can occur anywhere in the object module following the SEGDEF record to which they refer. They do not have to immediately follow the appropriate LEDATA record as FIXUPP records do.*

*These records are buffered by the linker in Pass 2 until the end of the module, after the linker applies all other FIXUPP records. Most linkers then processes these records as fixups.*

### Example

To generate a self-relative address whose TARGET is a forward reference (JZ forwardlabel), the translator can insert the negative offset of the next instruction (-*) from the start of the SEGDEF record, followed by an additive backpatch (meaning that the backpatch is added to the original value and the sum replaces the original value) whose Value is the offset of the TARGET of the jump, which is done last.

## B4H or B5H  LEXTDEF—Local External Names Definition Record

### Description

This record is identical in form to the EXTDEF record described earlier.  However, the symbols named in this record are not visible outside the module in which they are defined.

### History

This record is an extension to the original set of 8086 object record types.  It was added for Microsoft C 5.0. There is no semantic difference between the B4H and B5H types.

### Record Format

| 1 | 2 | 1 | <String Length> | 1 or 2 | 1 |
|---|---|---|---|---|---|
| B4<br>B5 | Record<br>Length | String<br>Length | External<br>Name String | Type<br>Index | Checksum |

<-------------------------repeated---------------------->

#### *Notes*

*These records are associated with LPUBDEF and LCOMDEF records and ordered with the EXTDEF records by occurrence, so that they may be referenced by an external name index for fixups.*

*The name string, when stored in the linker's internal data structures, is encoded with spaces and digits at the beginning of the name.*

### Example

This record type is produced in Microsoft C from static functions, such as:

```
static int var() { }
```

## B6H or B7H  LPUBDEF—Local Public Names Definition Record

### Description

This record is identical in form to the PUBDEF record described earlier.  However, the symbols named in this record are not visible outside the module in which they are defined.

### History

This record is an extension to the original set of 8086 object record types. It was added for Microsoft C 5.0. Record type B7H has been added for 32-bit linkers:  the Local Offset field is 32 bits rather than 16 bits.

### Record Format

| 1 | 2 | 1 or 2 | 1 or 2 | 2 | 1 | <String Length> | 2 or 4 | 1 or 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| B6 or B7 | Record Length | Base Group | Base Segment | Base Frame | String Length | Local Name String | Local Offset | Type Index | Checksum |

<conditional><--------------------repeated--------------------->

***Note:***  *In Microsoft C, the static keyword on functions or initialized variables produces LPUBDEF records.  Uninitialized static variables produce LCOMDEF records.*

## B8H LCOMDEF—Local Communal Names Definition Record

### Description

This record is identical in form to the COMDEF record described previously. However, the symbols named in this record are not visible outside the module in which they are defined.

### History

This record is an extension to the original set of 8086 object record types. It was added for Microsoft C 5.0.

### Record Format

| 1 | 2 | 1 | <String Length> | 1 or 2 | 1 | <from Data Type> | 1 |
|---|---|---|---|---|---|---|---|
| B8 | Record Length | String Length | Communal Name | Type Index | Data Type | Communal Length | Checksum |

<----------------------------------repeated----------------------------------------->

**Note:** *In Microsoft C, uninitialized static variables produce an LCOMDEF record.*

## BCH CEXTDEF—COMDAT External Names Definition Record

### Description

This record serves the same purpose as the EXTDEF record described earlier. However, the symbol named is referred to through a Logical Name Index field. Such a Logical Name Index field is defined through an LNAMES or LLNAMES record.

### History

The record is an extension to the original set of 8086 object record types. It was added for Microsoft C 7.0.

### Record Format

| 1 | 2 | 1 or 2 | 1 or 2 | 1 |
|---|---|---|---|---|
| BC | Record Length | Logical Name Index | Type Index | Checksum |

<------------repeated------------->

***Notes***

*A CEXTDEF can precede the COMDAT to which it will be resolved. In this case, the location of the COMDAT is not known at the time the CEXTDEF is seen.*

*This record is produced when a FIXUPP record refers to a COMDAT symbol.*

## C2H or C3H COMDAT—Initialized Communal Data Record

### Description

The purpose of the COMDAT record is to combine logical blocks of code and data that may be duplicated across a number of compiled modules.

### History

The record is an extension to the original set of 8086 object record types. It was added for Microsoft C 7.0.

### Record Format

| 1 | 2 | 1 | 1 | 1 | 2 or 4 | 1 or 2 | 1 or 2 | 1 or 2 [1]<br><var> [2] | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|
| C2 or C3 | Record Length | Flags | Attributes | Align | Enumerated Data Offset | Type Index | Public Base | Public Name | Data | Checksum |
| | | | | | | | | | <repeated> | |

### Flags Field

This field contains the following defined bits:

**01H** Continuation bit. If clear, this COMDAT record establishes a new instance of the COMDAT variable; otherwise, the data is a continuation of the previous COMDAT of the symbol.

**02H** Iterated data bit. If clear, the Data field contains enumerated data; otherwise, the Data field contains iterated data, as in an LIDATA record.

**04H** Local bit (effectively an "LCOMDAT"). This is used in preference to LLNAMES.

**08H** Data in code segment. If the application is overlaid, this COMDAT must be forced into the root text. Also, do not apply FARCALLTRANSLATION to this COMDAT.

*Note: This flag bit is not supported by IBM LINK386.*

### Attributes Field

This field contains two 4-bit fields: the Selection Criteria to be used and the Allocation Type, which is an ordinal specifying the type of allocation to be performed. Values are:

**Selection Criteria (High-Order 4 Bits):**

| Bit | Selection Criteria | |
|---|---|---|
| **00H** | No match | Only one instance of this COMDAT allowed. |
| **10H** | Pick Any | Pick any instance of this COMDAT. |
| **20H** | Same Size | Pick any instance, but instances must have the same length or the linker will generate an error. |

| | | |
|---|---|---|
| **30H** | Exact Match | Pick any instance, but checksums of the instances must match or the linker will generate an error. Fixups are ignored. |
| **40H – F0H** | | Reserved. |

**Allocation Type (Low-Order 4 bits):**

| **Bit** | **Allocation** | |
|---|---|---|
| **00H** | Explicit | Allocate in the segment specified in the ensuing Base Group, Base Segment, and Base Frame fields. |
| **01H** | Far Code | Allocate as CODE16. The linker will create segments to contain all COMDATs of this type. |
| **02H** | Far Data | Allocate as DATA16. The linker will create segments to contain all COMDATs of this type. |
| **03H** | Code32 | Allocate as CODE32. The linker will create segments to contain all COMDATs of this type. |
| **04H** | Data32 | Allocate as DATA32. The linker will create segments to contain all COMDATs of this type. |
| **05H - 0FH** | | Reserved. |

### Align Field

These codes are based on the ones used by the SEGDEF record:

| | |
|---|---|
| **0** | Use value from SEGDEF |
| **1** | Byte aligned |
| **2** | Word aligned |
| **3** | Paragraph (16 byte) aligned |
| **4** | Page aligned. (The original Intel specification uses 256-byte pages, the IBM OMF implementation uses 4096-byte pages.) |
| **5** | Double word (4 byte) aligned |
| **6** | Not defined |
| **7** | Not defined |

### Enumerated Data Offset Field

This field specifies an offset relative to the beginning location of the symbol specified in the Public Name Index field and defines the relative location of the first byte of the Data field. Successive data bytes in the Data field occupy higher locations of memory. This works very much like the Enumerated Data Offset field in an LEDATA record, but instead of an offset relative to a segment, this is relative to the beginning of the COMDAT symbol.

### Type Index Field

The Type Index field is encoded in index format; it contains either debug information or an old-style TYPDEF index. If this index is 0, there is no associated type data. Old-style TYPDEF indexes are ignored by most linkers. Linkers do not perform type checking.

### Public Base Field

This field is conditional and is identical to the public base fields (Base Group, Base Segment, and Base Frame) stored in the PUBDEF record. This field is present only if the Allocation Type field specifies Explicit allocation.

### Public Name Field

[1] Microsoft LINK recognizes this field as a regular logical name index (1 or 2 bytes).

[2] IBM LINK386 recognizes this field as a regular length-prefixed name.

### Data Field

The Data field provides up to 1024 consecutive bytes of data. If there are fixups, they must be emitted in a FIXUPP record that follows the COMDAT record. The data can be either enumerated or iterated, depending on the Flags field.

> ***Notes***
>
> *Record type C3H has an Enumerated Data Offset field of 32 bits.*
>
> *While creating addressing frames, most linkers add the COMDAT data to the appropriate logical segments, adjusting their sizes. At that time, the offset at which the data that goes inside the logical segment is calculated. Next, the linker creates physical segments from adjusted logical segments and reports any 64K boundary overflows.*
>
> *If the allocation type is not explicit, COMDAT code and data is accumulated by the linker and broken into segments, so that the total can exceed 64K.*
>
> *In Pass 2, only the selected occurrence of COMDAT data will be stored in virtual memory, fixed, and later written into the .EXE file.*
>
> *COMDATs are allocated in the order of their appearance in the .OBJ files if no explicit ordering is given.*
>
> *A COMDAT record cannot be continued across modules. A COMDAT record can be duplicated in a single module.*
>
> *If any COMDAT record on a given symbol has the local bit set, all COMDAT records on that symbol have that bit set.*

## C4H or C5H LINSYM—Symbol Line Numbers Record

### Description

This record will be used to output line numbers for functions specified through COMDAT records. Each LINSYM record is associated with a preceding COMDAT record.

### History

This record is an extension to the original set of 8086 object record types. It was added for Microsoft C 7.0.

### Record Format

| 1 | 2 | 1 | 1 or 2 [1]<br><var> [2] | 2 | 2 or 4 | 1 |
|---|---|---|---|---|---|---|
| C4 or C5 | Record Length | Flags | Public Name | Line Number | Line Number Offset | Checksum |

<----------repeated---------->

### Flags Field

This field contains one defined bit:

**01H** Continuation bit. If clear, this COMDAT record establishes a new instance of the COMDAT variable; otherwise, the data is a continuation of the previous COMDAT of the symbol.

### Public Name Field

[1] Microsoft LINK recognizes this field as a regular logical name index indicating the name of the base of the LINSYM record.

[2] IBM LINK386 recognizes this field as a length-preceded name indicating the name of the base of the LINSYM record.

### Line Number Field

An unsigned number in the range 0 to 65,535.

### Line Number Offset Field

The offset relative to the base specified by the symbol name base. The size of this field depends on the record type.

#### *Notes*

*Record type C5H is identical to C4H except that the Line Number Offset field is 4 bytes instead of 2.*

*This record is used to output line numbers for functions specified through COMDAT records. Often, the residing segment as well as the relative offsets of such functions is unknown at compile time, in that the linker is the final arbiter of such information. For such cases, most compilers will generate this record to specify the line number/offset pairs relative to a symbolic name.*

*This record will also be used to discard duplicate LINNUM information. If the linker encounters two or more LINSYM records with matching symbolic names (corresponding to multiple COMDAT records with the same name), the linker will keep the one that corresponds to the retained COMDAT.*

*LINSYM records must follow the COMDATs to which they refer. A LINSYM on a given symbol refers to the most recent COMDAT on the same symbol. LINSYMs inherit the "localness" of their COMDATs.*

## C6H ALIAS—Alias Definition Record

### Description

This record has been introduced to support link-time aliasing, a method by which compilers or assemblers may direct the linker to substitute all references to one symbol for another.

### History

The record is an extension to the original set of 8086 object record types for FORTRAN version 5.1 (Microsoft LINK version 5.13).

### Record Format

| 1 | 2 | <variable> | <variable> | 1 |
|---|---|---|---|---|
| C6 | Record Length | Alias Name | Substitute Name | Checksum |

<----------------------repeated------------------------>

#### Alias Name Field

A regular length-preceded name of the alias symbol.

#### Substitute Name Field

A regular length-preceded name of the substitute symbol.

#### *Notes*

*This record consists of two symbolic names: the alias symbol and the substitute symbol. The alias symbol behaves very much like a PUBDEF in that it must be unique. If a PUBDEF of an alias symbol is encountered later, the PUBDEF overrides the alias. If another ALIAS record with a different substitute symbol is encountered, a warning is emitted by most linkers, and the second substitute symbol is used.*

*When attempting to satisfy an external reference, if an ALIAS record whose alias symbol matches is found, the linker will halt the search for alias symbol definitions and will attempt to satisfy the reference with the substitute symbol.*

*All ALIAS records must appear before the Link Pass 2 record.*

## C8H or C9H  NBKPAT—Named Backpatch Record

### Description

The Named Backpatch record is similar to a BAKPAT record, except that it refers to a COMDAT record by logical name index rather than an LIDATA or LEDATA record. NBKPAT records must immediately follow the COMDAT/FIXUPP block to which they refer.

### History

This record is an extension to the original set of 8086 object record types.  It was added for Microsoft C 7.0.

### Record Format

| 1 | 2 | 1 | 1 or 2 [1]<br><var> [2] | 2 or 4 | 2 or 4 | 1 |
|---|---|---|---|---|---|---|
| C8<br>or C9 | Record<br>Length | Location<br>Type | Public<br>Name | Offset | Value | Checksum |

<-------repeated-------->

### Location Type Field

*Type of location to be patched; the only valid values are:*

| | |
|---|---|
| *0* | *8-bit byte* |
| *1* | *16-bit word* |
| *2* | *32-bit double word, record type C9H only* |

### Public Name Field

[1] Microsoft LINK recognizes this field as a regular logical name index of the COMDAT record to be back patched.

[2] IBM LINK386 recognizes this field as a length-preceded name of the COMDAT record to be back patched.

### Offset and Value Fields

These fields are 32 bits for record type C8H, 16 bits for C9H.

The Offset field specifies the location to be patched, as an offset into the COMDAT.

The associated Value field is added to the location being patched (unsigned addition, ignoring overflow).  The Value field is a fixed length (16 bits or 32 bits, depending on the record type) to make object module processing easier.

## CAH LLNAMES—Local Logical Names Definition Record

### Description

The LLNAMES record is a list of local names that can be referenced by subsequent SEGDEF and GRPDEF records in the object module.

The names are ordered by their occurrence, with the names in LNAMES records and referenced by index from subsequent records. More than one LNAMES and LLNAMES record may appear. The names themselves are used as segment, class, group, overlay, COMDAT, and selector names.

### History

This record is an extension to the original set of 8086 object record types. It was added for Microsoft C 7.0.

### Record Format

| 1 | 2 | 1 | <----String Length-----> | 1 |
|---|---|---|---|---|
| CA | Record Length | String Length | Name String | Checksum |

<-------------------repeated-------------------->

Each name appears in *count, char* format, and a null name is valid. The character set is ASCII. Names can be up to 255 characters long.

***Notes***

*Any LLNAMES records in an object module must appear before the records that refer to them.*

*Previous versions limited the name string length to 254 characters.*

## CCH VERNUM - OMF Version Number Record

### Description

The VERNUM record contains the version number of the object format generated. The version number is used to identify what version of the TIS-sponsored OMF was generated.

### History

This is a new record that was approved by the Tool Interface Standards (TIS) Committee, an open industry standards body.

### Record Format

| 1 | 2 | 1 | <----String Length-----> | 1 |
|---|---|---|---|---|
| CC | Record Length | Version Length | Version String | Checksum |

The version string consists of 3 numbers separated by periods (.) as follows:

<TIS Version Base>.<Vendor Number>.<Version>

The TIS Version Base is the base version of the OMF being used. This number is provided by the TIS Committee. The Vendor Number is assigned by TIS to allow extensions specific to a vendor. Finally, the Version is the Vendor-specific version. A Vendor Number or Version of zero (0) is reserved for TIS. For example, a version string of 1.0.0 indicates a TIS compliant version of the OMF without vendor additions.

## CEH  VENDEXT  -  Vendor-specific OMF Extension Record

### Description

The VENDEXT record allows vendor-specific extensions to the OMF.  All vendor-specific extensions use this record.

### History

This is a new record that was approved by the Tool Interface Standards (TIS) Committee, an open industry standards body.

### Record Format

| 1 | 2 | 2 | <from record length> | 1 |
|---|---|---|---|---|
| CE | Record Length | Vendor Number | Extension Bytes | Checksum |

The Vendor Number is assigned by the TIS Committee.  Zero (0) is reserved.  The Extension Bytes provide OMF extension information.

## Appendix 1: Microsoft Symbol and Type Extensions

Microsoft symbol and type information is stored on a per-module basis in specially-named logical segments. These segments are defined in the usual way (SEGDEF records), but the linker handles them specially, and they do not end up as segments in the .EXE file. These segment names are reserved:

| Segment Name | Class Name | Combine Type |
|---|---|---|
| $$TYPES | DEBTYP | Private |
| $$SYMBOLS | DEBSYM | Private |

The segment $$IMPORT should also be considered a reserved name, although it is not used anymore. This segment was not part of any object files but was emitted by the linker to get the loader to automatically do fixups for Microsoft symbol and type information. The linker emitted a standard set of imports, not just ones referenced by the program being linked. Use of this segment may be revisited in the future.

Microsoft symbol and type information-specific data is stored in LEDATA records for the $$TYPES and $$SYMBOLS segments, in various proprietary formats. The $$TYPES segment contains information on user-defined variable types; $$SYMBOLS contains information about nonpublic symbols: stack, local, procedure, block start, constant, and register symbols and code labels.

For instantiated functions in Microsoft C 7.0, symbol information for Microsoft symbol and type information will be output in COMDAT records that refer to segment $$SYMBOLS and have decorated names based on the function names. Type information will still go into the $$TYPES segment in LEDATA records.

All OMF records that specify a Type Index field, including EXTDEF, PUBDEF, and COMDEF records, use Microsoft symbol and type information values. Because many types are common, Type Index values in the range 0 through 511 (1FFH) are reserved for a set of predefined primitive types. Indexes in the range 512 through 32767 (200H-7FFFH) index into the set of type definitions in the module's $$TYPES segment, offset by 512. Thus 512 is the first new type, 513 the second, and so on.

## Appendix 2: Library File Format

The first record in the library is a header that looks like any other object module format record.

***Note:*** *Libraries under MS-DOS are always multiples of 512-byte blocks.*

**Library Header Record (*n* bytes)**

| 1 | 2 | 4 | 2 | 1 | <n - 10> |
|---|---|---|---|---|---|
| Type<br>(F0H) | Record Length<br>(Page Size Minus 3) | Dictionary<br>Offset | Dictionary Size<br>in Blocks | Flags | Padding |

The first byte of the record identifies the record's type, and the next two bytes specify the number of bytes remaining in the record. Note that this word field is byte-swapped (that is, the low-order byte precedes the high-order byte). The record type for this library header is F0H (240 decimal).

The Record Length field specifies the page size within the library. Modules in a library always start at the beginning of a page. Page size is determined by adding three to the value in the Record Length field (thus the header record always occupies exactly one page). Legal values for the page size are given by 2 to the *n*, where *n* is greater than or equal to 4 and less than or equal to 15.

The four bytes immediately following the Record Length field are a byte-swapped long integer specifying the byte offset within the library of the first byte of the first block of the dictionary.

The next two bytes are a byte-swapped word field that specifies the number of blocks in the dictionary.

Note: the MS-DOS Library Manager cannot create a library whose dictionary would require more than 251 512-byte pages.

The next byte contains flags describing the library. The current flag definition is:

0x01 = case sensitive (applies to both regular and extended dictionaries)

All other values are reserved for future use and should be 0.

The remaining bytes in the library header record are not significant. This record deviates from the typical OMF record in that the last byte is not used as a checksum on the rest of the record.

Immediately following the header is the first object module in the library. It, in turn, is succeeded by all other object modules in the library. Each module is in object module format (that is, it starts with a LHEADR record and ends with a MODEND record). Individual modules are aligned so as to begin at the beginning of a new page. If, as is commonly the case, a module does not occupy a number of bytes that is exactly a multiple of the page size, then its last block will be padded with as many null bytes as are required to fill it.

Following the last object module in the library is a record that serves as a marker between the object modules and the dictionary. It also resembles an OMF record.

**Library End Record (marks end of objects and beginning of dictionary)**

| 1 | 2 | <n> |
|---|---|---|
| Type (F1H) | Record Length | Padding |

The record's Type field contains F1H (241 decimal), and its Record Length field is set so that the dictionary begins on a 512-byte boundary. The record contains no further useful information; the remaining bytes are insignificant. As with the library header, the last byte is not a checksum.

## Dictionary

The remaining blocks in the library compose the dictionary. The number of blocks in the dictionary is given in the library header. The dictionary provides rapid searching for a name using a two-level hashing scheme.

Due to the hashing algorithm, the number of dictionary blocks must be a prime number, and within each block is a prime number of buckets. Whereas a librarian can choose the prime number less than 255 for the dictionary blocks, the number of buckets within a block is fixed at 37.

To search for a name within the blocks, two hashing indices and two hash deltas are computed. A block index and block delta controls how to go from one block to the other, and a bucket index and bucket delta controls how to search buckets within a block. Each bucket within a block corresponds to a single string.

A block is 512 bytes long and the first 37 bytes correspond to the 37 buckets. To find the string corresponding to a bucket, multiply the value stored in the byte by two and use that as an index into the block. At this location in the block lies an unsigned byte value for the string length, followed by the string characters (not 0-terminated), which in turn is followed by a two-byte little-endian-format module number in which the module in the library defining this string can be found. Thus, all strings start at even locations in the block.

Byte 38 in a block records the free space left for storing strings in the block, and is an index of the same format as the bucket indices; that is, multiply the bucket index by two to find the next available slot in the block. If byte 38 has the value 255, there is no space left.

### Dictionary Record (length is the dictionary size in 512-byte blocks)

| 37 | 1 | <variable> | 2 | <conditional> |
|---|---|---|---|---|
| HTAB | FFLAG | Name | Block Number | Align Byte |

< -------------------------------- repeated ------------------------------------>

Entries consist of the following: the first byte is the length of the symbol to follow, the following bytes are the text of the symbol, and the last two bytes are a byte-swapped word field that specifies the page number (counting the library header as page 0) at which the module defining the symbol begins.

All entries may have at most one trailing null byte in order to align the next entry on a word boundary.

Module names are stored in the LHEADR record of each module.

## Extended Dictionary

The extended dictionary is optional and indicates dependencies between modules in the library. Versions of LIB earlier than 3.09 do not create an extended dictionary. The extended dictionary is placed at the end of the library.

The dictionary is preceded by these values:

BYTE =0xF2 Extended Dictionary header
WORD length of extended dictionary in bytes excluding first three bytes

Start of extended dictionary:

WORD number of modules in library = N

Module table, indexed by module number, with N + 1 fixed-length entries:

WORD module page number
WORD offset from start of extended dictionary to list of required modules

Last entry is null.

### Dictionary Hashing Algorithm

Pseudocode for creating a library and inserting names into a dictionary is listed below.

```
    typedef unsigned short hash_value;
    typedef struct {
       hash_value block_x, block_d, bucket_x, bucket_d;
       } hash;
    typedef unsigned char block[512];
    unsigned short nblocks = choose some prime number such that it is
       likely that all the names will fit in those blocks (the value must be
       greater than 1 and less than 255);
    const int nbuckets = 37;
    const int freespace = nbuckets+1;
MORE_BLOCKS: ;
    // Allocate storage for the dictionary:
    block *blocks = malloc(nblocks*sizeof(block));
    // Zero out each block.
    memset(blocks,0,nblocks*sizeof(block));
    // Initialize freespace pointers.
    for (int i = 0; i < nblocks; i++)
       blocks[i][freespace] = freespace/2;

    for N <- each name you want to insert in the library do {
       int length_of_string = strlen(N);       // # of characters.
       // Hash the name, producing the four values (see below
       // for hashing algorithm):
       hash h = compute_hash(N, nblocks);
       hash_value start_block = h.block_x, start_bucket = h.bucket_x;
       // Space required:
       //         1 for len byte; string text; 2 bytes for module number;
       //         1 possible byte for pad.
       int space_required=1+length_of_string+2;
       if (space_required % 2) space_required++;      // Make sure even.
    NEXT_BLOCK: ;
       // Obtain pointer to block:
       unsigned char *bp = blocks[h.block_x];
       boolean success = FALSE;
       do {
            if (bp[h.bucket_x] == 0) {
                   if (512-bp[freespace]*2 < space_required) break;
                  // Found space.
                   bp[h.bucket_x] = bp[freespace];
                  int store_at = 2*bp[h.bucket_x];
                  bp[store_at] = length_of_string;
                  bp[store_at+1..store_at+length_of_string] = string characters;
                  int mod_location = store_at+length_of_string;
                  // Put in the module page number, LSB format.
                  bp[mod_location] = module_page_number % 256;
                  bp[mod_location+1] = module_page_number / 256;
                  bp[freespace] += space_required/2;
                  // In case we are right at the end of the block,
                  // set block to full.
                  if (bp[freespace] == 0) bp[freespace] = 0xff;
                  success = TRUE;
                  break;
                  }
            h.bucket_x = (h.bucket_x+h.bucket_d) % nbuckets;
            } while (h.bucket_x != start_bucket);
       if (!success) {
            // If we got here, we found no bucket.  Go to the next block.
            h.block_x = (h.block_x + h.block_d) % nblocks;
            if (h.block_x == start_block) {
                  // We got back to the start block; there is no space
                  // anywhere.  So increase the number of blocks to the
                  // next prime number and start all over with all names.
                  do nblocks++; while (nblocks is not prime);
                  free(blocks);
                  goto MORE_BLOCKS;
                  }
          // Whenever you can't fit a string in a block, you must mark
          // the block as full, even though there may be enough space
          // to handle a smaller string.  This is because the linker,
          // after failing to find a string in a block, will decide
          // the string is undefined if the block has any space left.
            bp[freespace] = 0xff;
            goto NEXT_BLOCK;
            }
        }
```

The order of applying the deltas to advance the hash indices is critical, due to the behavior of the linker. For example, it would not be correct to check a block to see if there is enough space to store a string before checking the block's buckets, because this is not the way the linker functions. The linker does not restart at bucket 0 when it moves to a new block. It resumes at the bucket last used in the previous block. Thus, the librarian must move through the buckets, even though there is not enough room for the string, so that the final bucket index is the same one the linker arrives at when it finishes searching the block.

The algorithm to compute the four hash indices is listed below.

```
hash compute_hash(const unsigned char*  name, int blocks) {
    int len = strlen(name);
    const unsigned char *pb = name, *pe = name+len;
    const int blank = 0x20;    // ASCII blank.
    hash_value
         // Left-to-right scan:
        block_x = len | blank, bucket_d = block_x,
        // Right-to-left scan:
        block_d = 0, bucket_x = 0;
    #define rotr(x,bits) ((x << 16-bits) | (x >>    bits))
    #define rotl(x,bits) ((x <<    bits) | (x >> 16-bits))
    while (1) {
        // blank -> convert to LC.
        unsigned short cback = *--pe | blank;
        bucket_x = rotr(bucket_x,2) ^ cback;
        block_d = rotl(block_d,2) ^ cback;
        if (--len == 0) break;
        unsigned short cfront = *pb++ | blank;
        block_x = rotl(block_x,2) ^ cfront;
        bucket_d = rotr(bucket_d,2) ^ cfront;
        }
    hash h;
    h.block_x = block_x % blocks;
    h.block_d = _max(block_d % blocks,1);
    h.bucket_x = bucket_x % nbuckets;
    h.bucket_d = _max(bucket_d % nbuckets,1);
    return h;
    }
```

## Appendix 3: Obsolete Records and Obsolete Features of Existing Records

This appendix contains a complete list of records that have been defined in the past but are not part of the TIS OMF. These record types are followed by a descriptive paragraph from the original Intel 8086 specification. When linkers encounter these records, they are free to process them, ignore them, or generate an error.

### Obsolete Records

**6EH RHEADR R-Module Header Record**

This record serves to identify a module that has been processed (output) by Microsoft LINK-86/LOCATE-86. It also specifies the module attributes and gives information on memory usage and need.

**70H REGINT Register Initialization Record**

This record provides information about the 8086 register/register-pairs: CS and IP, SS and SP, DS and ES. The purpose of this information is for a loader to set the necessary registers for initiation of execution.

**72H REDATA Relocatable Enumerated Data Record**

This record provides contiguous data from which a portion of an 8086 memory image may eventually be constructed. The data may be loaded directly by an 8086 loader, with perhaps some base fixups. The record may also be called a Load-Time Locatable (LTL) Enumerated Data Record.

**74H RIDATA Relocatable Iterated Data Record**

This record provides contiguous data from which a portion of an 8086 memory image may eventually be constructed. The data may be loaded directly by an 8086 loader, but data bytes within the record may require expansion. The record may also be called a Load-Time Locatable (LTL) Iterated Data Record.

**76H OVLDEF Overlay Definition Record**

This record provides the overlay's name, its location in the object file, and its attributes. A loader may use this record to locate the data records of the overlay in the object file.

**78H ENDREC End Record**

This record is used to denote the end of a set of records, such as a block or an overlay.

**7AH BLKDEF Block Definition Record**

This record provides information about blocks that were defined in the source program input to the translator that produced the module. A BLKDEF record will be generated for every procedure and for every block that contains variables. This information is used to aid debugging programs.

**7CH BLKEND Block End Record**

This record, together with the BLKDEF record, provides information about the scope of variables in the source program. Each BLKDEF record must be followed by a BLKEND record. The order of the BLKDEF, debug symbol records, and BLKEND records should reflect the order of declaration in the source module.

**7EH** **DEBSYM** **Debug Symbols Record**

This record provides information about all local symbols, including stack and based symbols. The purpose of this information is to aid debugging programs.

**84H** **PEDATA** **Physical Enumerated Data Record**

This record provides contiguous data, from which a portion of an 8086 memory image may be constructed. The data belongs to the "unnamed absolute segment" in that it has been assigned absolute 8086 memory addresses and has been divorced from all logical segment information.

**86H** **PIDATA** **Physical Iterated Data Record**

This record provides contiguous data, from which a portion of an 8086 memory image may be constructed. It allows initialization of data segments and provides a mechanism to reduce the size of object modules when there is repeated data to be used to initialize a memory image. The data belongs to the "unnamed absolute segment."

**8EH** **TYPDEF** This record contains details about the type of data represented by a name declared in a PUBDEF or EXTDEF record. For more details on this record, refer to its description later in this appendix.

**92H** **LOCSYM** **Local Symbols Record**

This record provides information about symbols that were used in the source program input to the translator that produced the module. This information is used to aid debugging programs. This record has a format identical to the PUBDEF record.

**9EH** **(none)** **Unnamed record**

This record number was the only even number not defined by the original Intel 8086 specification. Apparently it was never used.

**A4H** **LIBHED** **Library Header Record**

This record is the first record in a library file. It immediately precedes the modules (if any) in the library. Following the modules are three more records in the following order: LIBNAM, LIBLOC, and LIBDIC.

**A6H** **LIBNAM** **Library Module Names Record**

This record lists the names of all the modules in the library. The names are listed in the same sequence as the modules appear in the library.

**A8H** **LIBLOC** **Library Module Locations Record**

This record provides the relative location, within the library file, of the first byte of the first record (either a THEADR or LHEADR or RHEADR record) of each module in the library. The order of the locations corresponds to the order of the modules in the library.

**AAH** **LIBDIC** **Library Dictionary Record**

This record gives all the names of public symbols within the library. The public names are separated into groups; all names in the *n*th group are defined in the *n*th module of the library.

## 8EH TYPDEF—Type Definition Record

### Description

The TYPDEF record contains details about the type of data represented by a name declared in a PUBDEF or an EXTDEF record. This information may be used by a linker to validate references to names, or it may be used by a debugger to display data according to type.

Although the original Intel 8086 specification allowed for many different type specifications, such as scalar, pointer, and mixed data structure, many linkers used TYPDEF records to declare only communal variables. Communal variables represent globally shared memory areas—for example, FORTRAN common blocks or uninitialized public variables in Microsoft C. This function is served by the COMDEF record.

The size of a communal variable is declared explicitly in the TYPDEF record. If a communal variable has different sizes in different object modules, the linker uses the largest declared size when it generates an executable module.

### History

Starting with Microsoft LINK version 3.5, the COMDEF record should be used for declaration of communal variables. However, for compatibility, later versions of Microsoft LINK recognize TYPDEF records as well as COMDEF records.

### Record Format

| 1 | 2 | <variable> | 1 | <variable> | 1 |
|---|---|---|---|---|---|
| 8E | Record Length | Name | 0 (EN) | Leaf Descriptor | Checksum |

The name field of a TYPDEF record is in *count, char* format and is always ignored. It is usually a 1-byte field containing a single 0 byte.

The Eight-Leaf Descriptor field in the original Intel 8086 specification was a variable-length (and possibly repeated) field that contained as many as eight "leaves" that could be used to describe mixed data structures. Microsoft uses a stripped-down version of the Eight-Leaf Descriptor, of which the first byte, the EN byte, is always set to 0.

The Leaf Descriptor field is a variable-length field that describes the type and size of a variable. The two possible variable types are NEAR and FAR.

If the field describes a NEAR variable (one that can be referenced as an offset within a default data segment), the format of the Leaf Descriptor field is:

| 1 | 1 | <variable> |
|---|---|---|
| 62H | Variable Type | Length in Bits |

The 1-byte field containing 62H signifies a NEAR variable.
The Variable Type field is a 1-byte field that specifies the variable type:

| | |
|---|---|
| **77H** | Array |
| **79H** | Structure |
| **7BH** | Scalar |

This field must contain one of the three values given above, but the specific value is ignored by most linkers.

The Length in Bits field is a variable-length field that indicates the size of the communal variable. Its format depends on the size it represents.

If the first byte of the size is 128 (80H) or less, then the size is that value. If the first byte of the size is 81H, then a 2-byte size follows. If the first byte of the size is 84H, then a 3-byte size follows. If the first byte of the size is 88H, then a 4-byte size follows.

If the Leaf Descriptor field describes a FAR variable (one that must be referenced with an explicit segment and offset), the format is:

| 1 | 1 | <variable> | <variable> |
|---|---|---|---|
| 61H | Variable Type (77H) | Number of Elements | Element Type Index |

The 1-byte field containing 61H signifies a FAR variable.

The 1-byte variable type for a FAR communal variable is restricted to 77H (array). (As with the NEAR Variable Type field, the linker ignores this field, but it must have the value 77H.)

The Number of Elements field is a variable-length field that contains the number of elements in the array. It has the same format as the Length in Bits field in the Leaf Descriptor field for a NEAR variable.

The Element Type Index field is an index field that references a previous TYPDEF record. A value of 1 indicates the first TYPDEF record in the object module, a value of 2 indicates the second TYPDEF record, and so on. The TYPDEF record referenced must describe a NEAR variable. This way, the data type and size of the elements in the array can be determined.

***Note**: Microsoft LINK limits the number of TYPDEF records in an object module to 256.*

### Examples

The following three examples of TYPDEF records were generated by Microsoft C Compiler version 3.0. (Later versions use COMDEF records.)

The first sample TYPDEF record corresponds to the public declaration:

```
int      var;     /* 16-bit integer */
```

The TYPDEF record is:

```
      0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0000  8E  06  00  00  00  62  7B  10  7F                                .....b{..
```

Byte 00H contains 8EH, indicating that this is a TYPDEF record.

Bytes 01-02H contain 0006H, the length of the remainder of the record.

Byte 03H (the name field) contains 00H, a null name.

Bytes 04-07H represent the Eight-Leaf Descriptor field. The first byte of this field (byte 04H) contains 00H. The remaining bytes (bytes 05-07H) represent the Leaf Descriptor field:

- Byte 05H contains 62H, indicating that this TYPDEF record describes a NEAR variable.
- Byte 06H (the Variable Type field) contains 7BH, which describes this variable as scalar.
- Byte 07H (the Length in Bits field) contains 10H, the size of the variable in bits.

Byte 08H contains the Checksum field, 7FH.

The next example demonstrates how the variable size contained in the Length in Bits field of the Leaf Descriptor field is formatted:

```
char    var2[32768];   /* 32 KB array */
```

The TYPDEF record is:

```
        0    1    2    3    4    5    6    7    8    9    A    B    C   D   E   F
  0000  8E   09   00   00   00   62   7B   84   00   00   04   04                      .....bc{.....
```

The Length in Bits field (bytes 07-0AH) starts with a byte containing 84H, which indicates that the actual size of the variable is represented as a 3-byte value (the following three bytes). Bytes 08-0AH contain the value 040000H, the size of the 32K array in bits.

This third Microsoft C statement, because it declares a FAR variable, causes two TYPDEF records to be generated:

```
char    far  var3[10][2][20];    /* 400-element FAR array*/
```

The two TYPDEF records are:

```
        0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
  0000  8E   06   00   00   62   7B   08   87   8E   09   00   00   00   00   61   77  ....bc{......aw
  0010  81   90   01   01   7E                                                       .....|
```

Bytes 00-08H contain the first TYPDEF record, which defines the data type of the elements of the array (NEAR, scalar, 8 bits in size).

Bytes 09-14H contain the second TYPDEF record. The Leaf Descriptor field of this record declares that the variable is FAR (byte 0EH contains 61H) and an array (byte 0FH, the variable type, contains 77H).

> ***Note****: Because this TYPDEF record describes a FAR variable, bytes 10-12H represent a Number of Elements field. The first byte of the field is 81H, indicating a 2-byte value, so the next two bytes (bytes 11-12H) contain the number of elements in the array, 0190H (400D).*

Byte 13H (the Element Type Index field) contains 01H, which is a reference to the first TYPDEF record in the object module—in this example, the one in bytes 00-08H.

## PharLap Extensions to The SEGDEF Record (Obsolete Extension)

The following describes an obsolete extension to the SEGDEF record.

In the PharLap 32-bit OMF, there is an additional optional field that follows the Overlay Name Index field. The reserved bits should always be 0. The format of this field is

| < ------------------------------5 bits------------------------------ > | <--1 bit--> | < ---------2 bits ----- > |
|---|---|---|
| Reserved | U | AT |

where **AT** is the access type for the segment and has the following possible values

| | |
|---|---|
| **0** | Read only |
| **1** | Execute only |
| **2** | Execute/read |
| **3** | Read/write |

and **U** is the Use16/Use32 bit for the segment and has the following possible values:

| | |
|---|---|
| **0** | Use16 |
| **1** | Use32 |

**Conflict:** The PharLap OMF uses a 16-bit Repeat Count field, even in 32-bit records.