DWARF Debugging Information Format

UNIX International
Programming Languages SIG
Revision: Version 2, Draft 4 (October 12, 1992)

The material in this document represents work in progress of the UNIX International Programming Languages SIG. SIG members may duplicate and distribute the information as needed within their organizations. Further distribution is discouraged until the working group agrees that the work is mature enough for broader distribution.

UNAPPROVED DRAFT

Copyright © 1992 UNIX International, Inc.

Permission to use, copy, modify, and distribute this documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name UNIX International not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. UNIX International makes no representations about the suitability of this documentation for any purpose. It is provided "as is" without express or implied warranty.

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.

Trademarks:

Intel386 is a trademark of Intel Corporation.
UNIX® is a registered trademark of UNIX System Laboratories in the United States and other countries.

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++, FORTRAN 77, 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 debugging information, itself. Section two describes the overall structure of that information and attributes that are common to many or all of the different debugging information entries. Section three describes the specific debugging information entries and how they communicate the necessary information about the source program to a debugger.

The second piece of the DWARF description is the way the debugging information is encoded and represented in an object file. The DWARF encoding is presented in section four.

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++, Fortran, Modula2, Pascal). Therefore the document provides vendors a way to define their own debugging information tags, attributes, fundamental types, type modifiers, location operations, language names and calling conventions, 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 4 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.
- 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.

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 section 3.

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

TAG_array_type	TAG_class_type
TAG_common_block	TAG_common_inclusion
TAG_compile_unit	TAG_entry_point
TAG_enumeration_type	TAG_formal_parameter
TAG_global_subroutine	TAG_global_variable
TAG_imported_declaration	TAG_inheritance
TAG_inlined_subroutine	TAG_label
TAG_lexical_block	TAG_local_variable
TAG_member	TAG_module
TAG_namelist	TAG_pointer_type
TAG_ptr_to_member_type	TAG_reference_type
TAG_set_type	TAG_string_type
TAG_structure_type	TAG_subrange_type
TAG_subroutine	TAG_subroutine_type
TAG_typedef	TAG_union_type
TAG_unspecified_parameters	TAG_variant
TAG_variant_part	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 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 4.4).

AT_abstract_origin	AT_accessibility
AT_address_class	AT_artificial
AT_bit_offset	AT_bit_size
AT_byte_size	AT_calling_convention
AT_common_reference	AT_comp_dir
AT_const_value	AT_containing_type
AT_data_member_offset	AT_decl_column
AT_decl_file	AT_decl_line
AT_declaration	AT_default_value
AT_discr	AT_discr_list
AT_discr_value	AT_element_list
AT_frame_base	AT_friends
AT_fund_type	AT_high_pc
AT_identifier_case	AT_import
AT_inline	AT_is_optional
AT_language	AT_location
AT_low_pc	AT_lower_bound
AT_macro_info	AT_member
AT_mod_fund_type	AT_mod_u_d_type
AT_name	AT_namelist_items
AT_ordering	AT_priority
AT_producer	AT_prototyped
AT_return_addr	AT_segment
AT_sibling	AT_start_scope
AT_static_link	AT_stride_size
AT_string_length	AT_stmt_list
AT_subscr_data	AT_upper_bound
AT_use_location	AT_user_def_type
AT_variable_parameter	AT_virtuality
AT_visibility	AT_vtable_elem_location

Figure 2. Attribute names

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. Reference data use any of the constant data forms.

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 of debugging information entries is represented by the physical ordering of the debugging information entries and through the concepts of child and sibling entries. A debugging information entry is defined either to have child entries or not to have child entries (see section 4.3.3). 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 siblings ends with a null entry.

Conceptually, the debugging information entries form a graph whose nodes are declarations and whose arcs either point to the next declaration owned by the same parent (sibling) or the first declaration owned by that declaration (first child). The ordering of the entries is like a traversal of the graph that always traverses the first child arc of each node before the sibling arc.

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.

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 function 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. Location descriptions also appear inside `AT_subscr_data` attribute values. There, form is distinguished by the attribute's format code.

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. It is recommend that this mapping be defined by the ABI[1] authoring committee for each architecture.

1. `OP_REG0`, `OP_REG1`, ..., `OP_REG31`
   The `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. `OP_REGX`
   The `OP_REGX` operation has a single unsigned LEB128 literal operand that encodes the name of a register.

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. The 8-byte operations are only valid on 64-bit architectures.

1. `OP_LIT0`, `OP_LIT1`, ..., `OP_LIT31`
   The `OP_LIT`*n* operations encode the unsigned literal values from 0 through 31, inclusive.

2. `OP_ADDR`
   The `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. `OP_CONST1U`
   The single operand of the `OP_CONST1U` operation provides a 1-byte unsigned integer constant.

---

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

4. OP_CONST1S
   The single operand of the OP_CONST1S operation provides a 1-byte signed integer constant.
5. OP_CONST2U
   The single operand of the OP_CONST2U operation provides a 2-byte unsigned integer constant.
6. OP_CONST2S
   The single operand of the OP_CONST2S operation provides a 2-byte signed integer constant.
7. OP_CONST4U
   The single operand of the OP_CONST4U operation provides a 4-byte unsigned integer constant.
8. OP_CONST4S
   The single operand of the OP_CONST4S operation provides a 4-byte signed integer constant.
9. OP_CONST8U
   The single operand of the OP_CONST8U operation provides an 8-byte unsigned integer constant.
10. OP_CONST8S
    The single operand of the OP_CONST8S operation provides an 8-byte signed integer constant.
11. OP_CONSTU
    The single operand of the OP_CONSTU operation provides an unsigned LEB128 integer constant.
12. OP_CONSTS
    The single operand of the 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. OP_FBREG
   The OP_FBREG operation provides a signed LEB128 offset from the address specified by the location descriptor in the 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 loclist that adjusts the offset according to changes in the stack pointer as the PC changes.)*
2. OP_BREG0, OP_BREG1, ..., OP_BREG31
   The single operand of the OP_BREG*n* operations provides a signed LEB128 offset from the specified register.
3. OP_BREGX
   The 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 names and their meanings for the location operations that manipulate the ‘‘location stack,’’ are as follows. Location operations that index the location stack assume that the top of the stack (most recently added entry) has index 0.

1. OP_DUP
   The OP_DUP operation duplicates the value at the top of the location stack.
2. OP_DROP
   The OP_DROP operation pops the value at the top of the stack.
3. OP_PICK
   The single operand of the 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. OP_OVER
   The OP_OVER operation duplicates the entry currently second in the stack at the top of the stack. This is equivalent to an OP_PICK operation, with index 1.
5. OP_SWAP
   The 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. OP_ROT
   The 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. OP_DEREF
   The 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. OP_XDEREF
   The 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.

2.4.3.4 Arithmetic and Logical Operations

The required names and there meanings for the arithmetic and logical operations are as follows:

1. OP_ABS
   The OP_ABS operation pops the top stack entry and pushes its absolute value.
2. OP_AND
   The OP_AND operation pops the top two stack values, performs a bitwise *and* operation on the two, and pushes the result.
3. OP_DIV
   The 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. OP_MINUS
   The 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. OP_MOD
   The 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. OP_MUL
   The OP_MUL operation pops the top two stack entries, multiplies them together, and pushes the result.

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

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

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

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

11. OP_PLUS_UCONST
    The 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 "*OP_LIT*n* OP_ADD*".*

12. OP_SHL
    The 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. OP_SHR
    The 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. OP_SHRA
    The 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. OP_XOR
    The 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 OP_LE (less than or equal to), OP_GE (greater than or equal to), OP_EQ (equal to), OP_LT (less than), OP_GT (greater than) and OP_NE (not equal to).

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

   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. OP_NOP
   The 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	OP_DUP	0	17
1	29		1	17
2	1000		2	29
			3	1000
0	17	OP_DROP	0	29
1	29		1	1000
2	1000			
0	17	OP_PICK 2	0	1000
1	29		1	17
2	1000		2	29
			3	1000
0	17	OP_OVER	0	29
1	29		1	17
2	1000		2	29
			3	1000
0	17	OP_SWAP	0	29
1	29		1	17
2	1000		2	1000
0	17	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 OP_REG*n, or* OP_REGX *operations are used.*

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

`OP_REG3`
The value is in register 3.

`OP_REGX 54`
The value is in register 54.

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

`OP_BREG11 44`
Add 44 to the value in register 11 to get the address of an automatic variable instance.

`OP_FBREG -50`
Given an `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).*

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

`OP_ADD_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.

`OP_REG3 OP_PIECE 4 OP_REG10 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 an `AT_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 Type Attributes

Program variables and other debugging information entries have attributes that describe the type of the entry (as defined by the source language). There are four type attributes: fundamental types, user-defined types, modified fundamental types and modified user-defined types.

2.5.1 Fundamental Types

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

A fundamental type is represented in the debugging information as an `AT_fund_type` attribute whose value is a constant. The value corresponds to one member of the set of fundamental types whose names are enumerated in Figure 3.

`FT_boolean`	`FT_char`
`FT_complex`	`FT_dbl_prec_complex`
`FT_dbl_prec_float`	`FT_ext_prec_complex`
`FT_ext_prec_float`	`FT_float`
`FT_integer`	`FT_label`
`FT_long`	`FT_pointer`
`FT_short`	`FT_signed_char`
`FT_signed_integer`	`FT_signed_long`
`FT_signed_short`	`FT_unsigned_char`
`FT_unsigned_integer`	`FT_unsigned_long`
`FT_unsigned_short`	`FT_void`

Figure 3. Fundamental types

The type `FT_pointer` *may be used to describe the C type ‘‘void *’’. It is a more compact representation than a modified fundamental type (see below).*

The type `FT_label` *is used to describe Fortran ‘‘alternate return’’ parameters.*

2.5.2 User-Defined Type Attributes

In addition to the types that are built into a language, the user may define new data types. Some of the user-defined data types are described in their own debugging information entries. These types include structures, unions, arrays, classes and enumeration types.

There is an `AT_user_def_type` attribute whose value is a reference to the debugging information entry for a user-defined type.

2.5.3 Modified Types

There are user defined types that can be described by applying a small set of modifiers to other types. One or more modifiers may be applied to either fundamental types or user-defined types in the same fashion. There are modifiers that have the meanings ‘‘pointer to,’’ ‘‘reference to,’’ ‘‘const’’ and ‘‘volatile.’’

The ‘‘pointer to’’ modifier means that the value is the address of the data of the type modified. The ‘‘reference to’’ modifier means that the value is a C++ reference to data of the type modified. The ‘‘const’’ modifier means that the variable has been qualified by the ANSI C or C++ qualifier `const`. *The ‘‘volatile’’ modifier means that the variable has been qualified by the ANSI C qualifier* `volatile`.

The full set of type modifier names are listed in figure 4.

```
MOD_const
MOD_pointer_to
MOD_reference_to
MOD_volatile
```

Figure 4. Type modifiers

Modifiers are applied by prefixing a fundamental type value or a user-defined type reference with one or more modifiers. The modifiers appear as if part of a right-associative expression involving the fundamental or user-defined type. When one or more modifiers are applied to a fundamental type, the modifiers and the fundamental type value are stored in a block of contiguous bytes that are the value of the `AT_mod_fund_type` attribute. When one or more modifiers are applied to a user-defined type, the modifiers and the user-defined type reference are stored in a block of contiguous bytes that are the value of the `AT_mod_u_d_type` attribute.

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:

```
MOD_volatile MOD_pointer_to MOD_const FT_char
```

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

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

```
MOD_const MOD_pointer_to MOD_volatile FT_char
```

2.5.4 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 an `AT_accessibility` attribute, whose value is a constant drawn from the set of codes listed in Figure 5.

```
ACCESS_public
ACCESS_private
ACCESS_protected
```

Figure 5. Accessibility codes

2.5.5 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 an `AT_visibility` attribute, whose value is a constant drawn from the set of codes listed in Figure 6.

```
VIS_local
VIS_exported
VIS_qualified
```

Figure 6. Visibility codes

2.5.6 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 an `AT_virtuality` attribute, whose value is a constant drawn from the set of codes listed in Figure 7.

```
VIRTUALITY_none
VIRTUALITY_virtual
VIRTUALITY_pure_virtual
```

Figure 7. Virtuality codes

2.5.7 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 an `AT_artificial` attribute. The value of this attribute is a flag.

2.5.8 Target-Specific Addressing Information

In many 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 an 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 AT_segment attribute has an AT_low_pc or 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 AT_segment.

If an entry has no 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 AT_segment attributes, then the entry is assumed to exist within a flat address space. Similarly, if the entry has an 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 an AT_address_class attribute, whose value is a constant. The set of permissible values is specific to each target architecture. The value 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
ADDR_none	0	no class specified
ADDR_near16	1	16-bit offset, no segment
ADDR_far16	2	16-bit offset, 16-bit segment
ADDR_huge16	3	16-bit offset, 16-bit segment
ADDR_near32	4	32-bit offset, no segment
ADDR_far32	5	32-bit offset, 16-bit segment

Figure 8. Example address class codes

2.6 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 an AT_declaration attribute, whose value is a flag.

2.7 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 AT_decl_file, AT_decl_line and AT_decl_column attributes, each of whose value is a constant.

The value of the 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 3.13). The value 0 indicates that no source file has been specified.

The value of the 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 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.

2.8 Identifier Names

Any debugging information entry representing a program entity that has been given a name may have an 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 AT_name *attribute, including the keyword* operator*, if present. Sequences of multiple whitespace characters may be compressed.*

3. DEBUGGING INFORMATION ENTRIES

This section describes the debugging information entries currently defined and the attributes they contain.

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 `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. An `AT_low_pc` attribute whose value is the relocated address of the first machine instruction generated for that compilation unit.
2. An `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.
3. An `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. An `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 9.

`LANG_ADA83`	ADA
`LANG_C`	Non-ANSI C, such as K&R
`LANG_C89`	ISO/ANSI C
`LANG_C_PLUS_PLUS`	C++
`LANG_COBOL74`	ANSI COBOL-74
`LANG_COBOL85`	ANSI COBOL-85
`LANG_FORTRAN77`	FORTRAN77
`LANG_FORTRAN90`	Fortran90
`LANG_MODULA2`	Modula2
`LANG_PASCAL83`	ISO/ANSI Pascal

Figure 9. Language names

5. An `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 3.13.
6. An `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 3.14.

7. An `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 `AT_comp_dir` *attribute on UNIX systems is ‘‘hostname*`:`*pathname’’. If no hostname is available, the suggested form is ‘‘*`:`*pathname’’.*

8. An `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. An `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 10.

`ID_case_sensitive`
`ID_up_case`
`ID_down_case`

Figure 10. Identifier case codes

   `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 `AT_name` attributes in debugging information entries for the compilation unit reflect the names as they appeared in the source program.

   `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.

   `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

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

3.2 Modules

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

A module is represented by a debugging information entry with the tag `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 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 low pc attribute whose value is the relocated address of the first machine instruction generated for that initialization code. It also has a 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 an `AT_priority` attribute. The value of this attribute may be a string or any of the constant data or data block forms. as appropriate for the representation of the priority. The value of this attribute 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 an `AT_declaration` *attribute.*

3.3 Subroutine and Entry Point Entries

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

`TAG_global_subroutine`	A global subroutine or function.
`TAG_subroutine`	A file static subroutine or function.
`TAG_inlined_subroutine`	A particular inlined instance of a subroutine or function.
`TAG_entry_point`	A Fortran entry point.

3.3.1 General Subroutine and Entry Point Information

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

The subroutine entry for a member function definition for a member function defined outside of a class, structure or union body has an `AT_member` attribute whose value is a reference to a type definition of the class or structure. *

Additional attributes for member functions are described in section 3.9.4.3.

The presence of the member attribute implies that the subroutine is a member of the type specified by the member attribute. The member attribute makes C++ identifier resolution through member functions possible.

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 an `AT_calling_convention` attribute, whose value is a constant. If this attribute is not present, or its value is the constant `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 `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 `CC_program`.

The `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 one of the four type attributes (fundamental type, modified fundamental type, user-defined type or modified user-defined type) described in section 2.5, 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 an `AT_prototyped` attribute, whose value is a flag.

3.3.3 Subroutine and Entry Point Locations

A subroutine entry has a low pc attribute whose value is the relocated address of the first machine instruction generated for the subroutine. It also has a 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 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 segment and 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 `TAG_unspecified_parameters`.

The entry for a subroutine or entry point that includes a Fortran common block has a child entry with the tag `TAG_common_inclusion`. The common inclusion entry has an

AT_common_reference attribute whose value is a reference to the debugging entry for the common block being included (see section 3.7).

3.3.5 Low-Level Information

A subroutine or entry point entry may have an 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 an 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 AT_frame_base *attribute can be used in several ways:*

1. *In procedures that need location lists to locate local variables, the* 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* 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 AT_static_link *and* AT_frame_base *attributes allow debuggers to support this same kind of referencing.*

If a subroutine or entry point is nested, it may have an 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 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 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 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 Inline Subroutines

A declaration or a definition of an inlinable subroutine is represented by a debugging information entry with either of the tags TAG_global_subroutine or TAG_local_subroutine (depending upon the linkage). 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 an AT_inline attribute whose value is a constant. The set of values for the AT_inline attribute is given in Figure 11.

Name	Meaning
INL_not_inlined	Not declared inline nor inlined by the compiler
INL_inlined	Not declared inline but inlined by the compiler
INL_declared_not_inlined	Declared inline but not inlined by the compiler
INL_declared_inlined	Declared inline and inlined by the compiler

Figure 11. Inline codes

3.3.6.1 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 AT_inline attribute will be referred to as an ‘‘abstract instance entry.’’ Any subroutine entry that contains an 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 an AT_high_pc, AT_low_pc, AT_location, AT_return_addr, AT_start_scope, or 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.

3.3.6.2 Concrete Inlined Instances

Each inline expansion of an inlinable subroutine is represented by a debugging information entry with the tag 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 low pc attribute, representing the address of the first instruction associated with the given inline expansion. Each inlined subroutine entry also contains a 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 TAG_inlined_subroutine will be referred to as a ‘‘concrete inlined instance entry.’’ Any entry that has the tag 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 `AT_low_pc`, `AT_high_pc`, `AT_location`, `AT_return_addr`, `AT_start_scope` and `AT_segment`) that such entries would otherwise normally have. In place of these omitted attributes, each concrete inlined instance entry has an `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 an `AT_abstract_origin` attribute, both members of the pair will have the same tag. So, for example, an entry with the tag `TAG_local_variable` can only be associated with another entry that also has the tag `TAG_local_variable.` The only exception to this rule is that the root of a concrete instance tree (which must always have the tag `TAG_inlined_subroutine`) can only be associated with the root of its associated abstract instance tree (which must have one of the tags `TAG_global_subroutine` or `TAG_local_subroutine`).

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.

3.3.6.3 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

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 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 TAG_lexical_block.

The lexical block entry has a 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 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 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 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 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 name attribute whose value is a null-terminated string containing the name of the label as it appears in the source program.

3.6 Program Variable Entries

Global variables, formal parameters and local variables are represented by debugging information entries with the tags `TAG_global_variable`, `TAG_formal_parameter` and `TAG_local_variable`, respectively.

The definitions of C++ static data members of structures or classes are represented by debugging information entries with the tag `TAG_global_variable`. *File static variables in C and C++ are represented by entries with the tag* `TAG_local_variable`.

The debugging information entry for a program variable may have the following attributes:

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

2. An `AT_location` attribute, whose value describes the location of the variable at run-time.

   A variable entry representing a non-defining declaration of the variable will not have a location attribute, and will have the `AT_declaration` attribute.

   In a variable entry representing the definition of the variable (that is, with no `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 8, below).

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

3. One of the four type attributes (fundamental type, modified fundamental type, user-defined type or modified user defined type).

4. If the variable entry represents the defining declaration for a C++ static data member of a structure, class or union, the entry has an `AT_member` attribute, whose value is a reference to the structure, class or union type of which this object is a member.

5. *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 an `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.

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

   If the variable represents an optional parameter, it has an `AT_is_optional` attribute, whose value is a flag.

7. A formal parameter entry describing a formal parameter that has a default value may have an `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 `AT_default_value` attribute is 0, it means that no default value has been specified.

8. A variable entry describing a variable whose value is constant and not represented by an object in the address space of the program does not have a location attribute. Such an entry has an 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.

9. If the scope of the variable begins sometime after the low pc value for the scope most closely enclosing the variable, the variable may have an AT_start_scope attribute. The value of this attribute is the offset in bytes of the beginning of the scope for the variable 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*.*

3.7 Common Block Entries

A Fortran common block may be described by a debugging information entry with the tag TAG_common_block. The common block entry has a name attribute whose value is a null-terminated string containing the common block name as it appears in the source program. It also has an 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.

3.8 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 TAG_imported_declaration. The entry for the imported declaration has a 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 an AT_import attribute, whose value is a reference to the debugging information entry representing the declaration that is being imported.

3.9 User-Defined Type Entries

There are several debugging information entry types that describe user-defined data types, including typedefs, pointers, references, arrays, structures, unions, classes, enumerations, subroutines, strings, sets and subranges.

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 an `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.

3.9.1 Typedef Entries

Any arbitrary type named via a typedef is represented by a debugging information entry with the tag `TAG_typedef`. The typedef entry has a 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 one of the four type attributes (fundamental type, user defined type, modified fundamental type, or modified user defined type).

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

3.9.2 Pointer and Reference Type Entries

Several languages share the concept of a ‘‘pointer,’’ which is an object whose value is the address of another object. A pointer can also be ‘‘null,’’ which means that it does not refer to any entity. In addition to the pointer, C++ supports the concept of a ‘‘reference.’’ A reference is a fixed pointer to another object. It may be thought of as an alternate name for the object with which it has been initialized or as a pointer that is automatically de-referenced.

A pointer type may be represented by a debugging information entry with the tag `TAG_pointer_type`. A reference type may be represented by a debugging information entry with the tag `TAG_reference_type`.

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

The pointer type entry or reference type entry has a fundamental type attribute, a modified fundamental type attribute, a user-defined type attribute, or a modified user defined type attribute to denote the type pointed to or referenced.

A pointer or reference type entry may have an address class attribute to describe how objects having the given pointer or reference type ought to be dereferenced.

3.9.3 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 `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 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 an AT_ordering attribute whose constant value is interpreted to mean either row-major or column-major ordering of array elements. The required attribute names are listed in Figure 12. If no ordering attribute is present, the default ordering for the source language (which is indicated by the AT_language attribute of the enclosing compilation unit entry) is assumed.

ORD_col_major
ORD_row_major

Figure 12. Array ordering

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

The array subscripts and the data type of the elements of the array are described by a subscript data attribute (AT_subscr_data) whose value is stored in a block of contiguous bytes. The subscript data attribute consists of a list of data items. There is a data item describing each array dimension and an item describing the element type. The data items describing the array dimensions are ordered to reflect the appearance of the dimensions in the source program (i.e. leftmost dimension first, next to leftmost dimension second, and so on). The last data item in the subscript data attribute is the description of the element type.

Each data item describing an array dimension consists of four parts, ordered as follows:

1. A format specifier describing the information that follows.
2. The type of the subscript index. This may be either a fundamental type value or a user-defined type reference. In either case, there is no attribute tag, but the value has the same form as a fundamental type or user-defined type reference when used as an attribute value.
3. Information describing the low bound of the array dimension. This may be either a signed constant or a location description. The location description is contained in a block of contiguous bytes; its generated value is the address of the lowest numbered element of this array dimension calculated during the execution of a program using that array. An unspecified lower bound is represented by a block of contiguous bytes with a length of zero. As with the subscript types, neither the constant nor the location description has a preceding attribute tag, but each follows the form for constant or location description attribute values.
4. Information describing the high bound of the array dimension. Again, this may either be a signed constant or a location description. An unspecified upper bound is represented by a block of contiguous bytes with a length of zero.

The data item for the element type is constructed from a format specifier followed by either a fundamental type attribute, a modified fundamental type attribute, a user-defined type attribute or a modified user defined type attribute.

The format specifier determines how the pieces of the data items should be interpreted and replaces the need for specific attribute tags to describe subscript index types and upper and lower dimension bounds. The format specifier is actually composed of three sub-fields:

1. An index type format descriptor
2. A upper bound format descriptor
3. A lower bound format descriptor

The index type format descriptor can have one of the following values and meanings:

`AR_index_ft`	The index descriptor is a fundamental type.
`AR_index_ut`	The index descriptor is a user defined type.
`AR_et`	A type attribute for the element type of the array. (Note that if the index type format descriptor has this value then the bound descriptors are undefined.)

The upper and lower bound format descriptors can have the following format:

`AR_bound_constant`	The bound format descriptor is a constant.
`AR_bound_locexpr`	The bound format descriptor is a location expression.
`AR_bound_loclist`	The bound format descriptor is a location list.

Note that the order of components of a subscript data item is fixed and independent of the value of its format specifier.

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 an `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 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.

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.

3.9.4 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.

3.9.4.1 General Structure Description

Structure, union, and class types are represented by debugging information entries with the tags `TAG_structure_type`, `TAG_union_type` and `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 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 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 an `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 an `AT_member` attribute referencing the structure, union or class declaration containing the given member. 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 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 an `AT_declaration` attribute.

3.9.4.1.1 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 `TAG_inheritance`.

An inheritance entry has a user-defined 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 location attribute 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 an `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 an `AT_virtuality` attribute.

3.9.4.1.2 Friends

If the declaration of a structure, union or class type specifies ‘‘friends’’ to that structure, union or class, then the debugging entry for the type may have an `AT_friends` attribute. The value of this attribute is a list of references to the debugging information entries for the structures, unions, classes or functions declared to be friends to the structure, union or class containing the friends attribute.

3.9.4.2 Structure Data Member Entries

A data member (as opposed to a member function) is represented by a debugging information entry with the tag `TAG_member`. The member entry for a named member has a name attribute whose value is a null-terminated string containing the member name as it appears in the source program.

The structure data member entry has a fundamental type attribute, a modified fundamental type attribute, a user-defined type attribute, or a modified user defined type attribute to denote the type of that member.

If the member entry is defined in the structure or class body, it has an `AT_data_member_offset` 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. An `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. An `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. An `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:

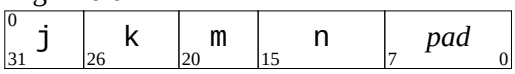
```
struct S {
            int   j:5;
            int   k:6;
            int   m:5;
            int   n:8;
};
```

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
```

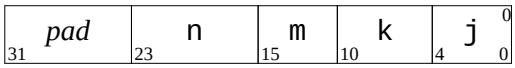
Big-Endian



Bit Offsets:

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

Little-Endian



3.9.4.3 Structure Member Function Entries

A member function is represented in the debugging information by a debugging information entry with the tag `TAG_global_subroutine`. 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 virtuality attribute.

An entry for a virtual function also has an `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.

3.9.4.4 Variant Entries

A variant part of a structure is represented by a debugging information entry with the tag `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 an `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 one of the four type attributes (fundamental type, user defined type, modified fundamental type, or modified

user defined type) to represent the tag type.

Each variant of a particular variant part is represented by a debugging information entry with the tag `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 an `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 an `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 `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.

DSC_label
DSC_range

Figure 13. Discriminant descriptor values

If a variant entry has neither an `AT_discr_value` attribute nor an `AT_discr_list` attribute, or if it has an `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.

3.9.5 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 `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 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 byte size attribute whose constant value is the number of bytes required to hold an instance of the enumeration.

Information about the enumeration literals is stored in an `AT_element_list` attribute whose value is a list of data elements stored in a block of contiguous bytes. Each data item consists of a constant followed by a null-terminated block of contiguous bytes. The constant will contain the internal value of the enumeration element. The null-terminated block of bytes holds the enumeration literal string as it appears in the source program. The data elements in an element list should be generated in the reverse order to the order in which they appear in the source program.

For consistency, DWARF specifies an ordering for enumeration entries. Reverse order was chosen for compatibility with existing implementations.

3.9.6 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 `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 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 fundamental type attribute, a modified fundamental type attribute, a user-defined type attribute, or a modified user-defined 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 an `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 `TAG_formal_parameter`.

   The formal parameter entry has a fundamental type attribute, a modified fundamental type attribute, a user-defined type attribute, or a modified user defined 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 `TAG_unspecified_parameters`.

3.9.7 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 `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 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 an `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 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 the data to be retrieved is the same as the size of the

fundamental type FT_long on the target machine.

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

3.9.8 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 TAG_set_type. If a name has been given to the set type, then the set type entry has a 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 fundamental type attribute, a modified fundamental type attribute, a user-defined type attribute, or a modified user defined 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 byte-size attribute, whose constant value represents the size in bytes of an instance of the set type.

3.9.9 Subrange Types

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.

A subrange type is represented by a debugging information entry with the tag TAG_subrange_type. If a name has been given to the subrange type, then the subrange type entry has a 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 one of the four type attributes (fundamental type, modified fundamental type, user-defined type, modified user-defined type) 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 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 AT_lower_bound and AT_upper_bound to describe, respectively, the lower and upper bound values of the subrange. If a bound value is described by a constant not represented in the program’s address space that can be represented by one of the constant attribute forms, then the value of the lower or upper bound attribute may be one of the constant types. Otherwise, the value of the lower or upper bound attribute is a reference to a debugging information describing an object containing the bound value or itself describing a constant value.

If either the lower or upper bound 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 attribute (if it references an object). If there is no upper bound attribute or it does not reference an object, the type is assumed to be the same type object represented by the fundamental type `FT_long`.

3.9.10 Pointer to Member Types

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 `TAG_ptr_to_member_type`.

If the pointer to member type has a name, the pointer to member entry has a 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 one of the four type attributes (fundamental type, modified fundamental type, user-defined type, modified user-defined type) 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 an `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 an `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.

As with locations for data members of structures, the location expression for an `AT_use_location` *attribute expects the base address of the structure or class object to be on the location expression stack before the* `AT_use_location` *expression is evaluated.*

3.10 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 `TAG_with_stmt`. A with statement entry has a 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 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 user-defined 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 location description attribute, describing how to find the base address of the record object referenced within the body of the with statement.

3.11 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 TAG_namelist. If the namelist itself has a name, the namelist entry has an AT_name attribute, whose value is a null-terminated string containing the namelist's name as it appears in the source program.

The namelist entry also has an AT_namelist_items attribute whose value is a block containing a sequence of references to the debugging information entries representing the declarations of the names belonging to the list. The list of references is ordered as were the list of names they correspond to in the source program.

3.12 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.

3.12.1 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 of 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 AT_name attribute of the referenced debugging entry. Each set of names is terminated by zero.

3.12.2 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 pairs of address range descriptors. Each pair consists 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 a pair 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.

3.13 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 currently exists 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.

3.13.1 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 4.4.

3.13.2 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’’

3.13.3 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 `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.

3.13.4 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`.

3.13.5 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.)

3.13.5.1 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`.

3.13.5.2 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. `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. `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. `LNS_advance_line`
   Takes a single signed LEB128 operand and adds that value to the `line` register of the state machine.
4. `LNS_set_file`
   Takes a single unsigned LEB128 operand and stores it in the `file` register of the state machine.
5. `LNS_set_column`
   Takes a single unsigned LEB128 operand and stores it in the `column` register of the state machine.

6. LNS_negate_stmt
   Takes no arguments. Set the is_stmt register of the state machine to the logical negation of its current value.

7. LNS_set_basic_block
   Takes no arguments. Set the basic_block register of the state machine to ‘‘true.’’

8. 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 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* 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* LNS_advance_pc *and a special opcode, requiring three or more bytes.*

9. 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 LNS_fixed_advance_pc *is this: existing assemblers cannot emit* 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* LNS_advance_pc *instead. However, such assemblers can use* LNS_fixed_advance_pc *instead, sacrificing compression.*

3.13.5.3 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. 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 an 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. 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. 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 compiler can emit these opcodes just before the statement program first needs to refer to the file. 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 `LNE_define_file` | instruction. These numbers are used in the the `file` register of the state machine.

Appendix 3 gives some sample statement programs.

3.14 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.

3.14.1 Macinfo Types

The valid macinfo types are as follows:

`MACINFO_define`	A macro definition.
`MACINFO_undef`	A macro un-definition.
`MACINFO_start_file`	The start of a new source file inclusion.
`MACINFO_end_file`	The end of the current source file inclusion.
`MACINFO_vendor_ext`	Vendor specific macro information directives that do not fit into one of the standard categories.

3.14.1.1 Define and Undefine Entries

All `MACINFO_define` and `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 `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 `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.

3.14.1.2 Start File Entries

Each `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 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.

3.14.1.3 End File Entries

A `MACINFO_end_file` entry has no operands. The presence of the entry marks the end of the current source file inclusion.

3.14.1.4 Vendor Extension Entries

A `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 `MACINFO_vendor_ext` entries that it does not understand.

3.14.2 Base Source Entries

In addition to producing a matched pair of `MACINFO_start_file` and `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 `MACINFO_start_file` and `MACINFO_end_file` entries for the base source file, however, the file name indicated (indirectly) by the `MACINFO_start_file` entry of the pair should reference a statement information file name entry consisting of a null string.

3.14.3 Macinfo Entries for Command Line Options

In addition to producing `MACINFO_define` and `MACINFO_undef` entries for each of the define and undefine directives processed during compilation, the DWARF producer should

generate a `MACINFO_define` or `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 `MACINFO_define` and `MACINFO_undef` entries.

All such `MACINFO_define` and `MACINFO_undef` entries representing compilation options should appear prior to the first `MACINFO_start_file` entry for that compilation unit and should encode the value 0 in their line number operands.

3.14.4 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.

4. 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.

4.1 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, fundamental types, location operations, type modifiers, language names and calling conventions. The labels denoting the beginning and end of the reserved value range for vendor specific extensions consist of the appropriate prefix (`TAG`, `AT`, `FT`, `OP`, `MOD`, `LANG`, or `CC` respectively) followed by `_lo_user` or `_hi_user`. For example, for entry tags, the special labels are `TAG_lo_user` and `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 tag, attribute, fundamental type, location atom, type modifier and language names 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.

4.2 Reserved Error Values

As a convenience for consumers of DWARF information, the value 0 is reserved in the encodings for attribute names, attribute forms, location operations, fundamental types, type modifiers, 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.

4.3 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.

4.3.1 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. An 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 `TAG_compile_unit` *debugging information entry. It is additional information that is represented outside the standard DWARF tag/attributes format.*

4.3.2 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.

4.3.3 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 consists of: 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 tag encodings are given in Figure 14.

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

Tag name	Value
TAG_array_type	0x01
TAG_class_type	0x02
TAG_entry_point	0x03
TAG_enumeration_type	0x04
TAG_formal_parameter	0x05
TAG_global_subroutine	0x06
TAG_global_variable	0x07
TAG_label	0x0a
TAG_lexical_block	0x0b
TAG_local_variable	0x0c
TAG_member	0x0d
TAG_pointer_type	0x0f
TAG_reference_type	0x10
TAG_compile_unit	0x11
TAG_string_type	0x12
TAG_structure_type	0x13
TAG_subroutine	0x14
TAG_subroutine_type	0x15
TAG_typedef	0x16
TAG_union_type	0x17
TAG_unspecified_parameters	0x18
TAG_variant	0x19
TAG_common_block	0x1a
TAG_common_inclusion	0x1b
TAG_inheritance	0x1c
TAG_inlined_subroutine	0x1d
TAG_module	0x1e
TAG_ptr_to_member_type	0x1f
TAG_set_type	0x20
TAG_subrange_type	0x21
TAG_with_stmt	0x22
TAG_imported_declaration	0x23
TAG_namelist	0x24
TAG_variant_part	0x25
TAG_lo_user	0x4080
TAG_hi_user	0xffff

Figure 14. Tag encodings

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 15.

Finally, the child encoding is followed by a series of attribute specifications. The abbreviations for a given compilation unit end with an entry consisting of a 0 byte for the abbreviation code. 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

Child determination name	Value
DW_children_no	0
DW_children_yes	1

Figure 15. Child determination encodings

name and 0 for the form.

The attribute form 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.

See Appendix 2 for a depiction of the organization of the debugging information.

4.3.4 Attribute Encodings

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 (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 (FORM_BLOCK1). The second consists of a 2-byte length followed by 0 to 65,535 contiguous information bytes (FORM_BLOCK2). The third consists of a 4-byte length followed by 0 to 4,294,967,295 contiguous information bytes (FORM_BLOCK4). The fourth consists of an unsigned LEB128 length followed by the number of bytes specified by the length (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, FORM_DATA1, FORM_DATA2, FORM_DATA4, and FORM_DATA8. There are also variable length constant data forms encoded using LEB128 numbers (see below). Both signed (FORM_SDATA) and unsigned (FORM_UDATA) variable length constants are available. *

flag — A flag is represented as a single byte of data (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 — Represented as any of the constant data forms. A reference is an offset relative to the first byte of the compilation unit header for the compilation unit containing the reference entry.

Note that in DWARF Version 1 references were relative to the beginning of the .debug section, rather than to a particular compilation unit. Compilation unit relative references will reduce the number of link-time relocations and so speed up linking.

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 (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 (FORM_STRP).

The form encodings are listed in Figure 16.

Form name	Value	Class
FORM_ADDR	0x01	address
FORM_BLOCK2	0x03	block
FORM_BLOCK4	0x04	block
FORM_DATA2	0x05	constant, reference
FORM_DATA4	0x06	constant, reference
FORM_DATA8	0x07	constant, reference
FORM_STRING	0x08	string
FORM_BLOCK	0x09	block
FORM_BLOCK1	0x0a	block
FORM_DATA1	0x0b	constant, reference
FORM_FLAG	0x0c	flag
FORM_INDIRECT	0x0d	(see section 4.3.3)
FORM_SDATA	0x0e	constant, reference
FORM_STRP	0x0f	string
FORM_UDATA	0x10	constant, reference

Figure 16. Attribute form encodings

The encodings for the attribute names are given in Figures 17 and 18.

4.4 Variable Length Data

The special constant data forms FORM_SDATA and 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).

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 19 gives some examples of 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 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 containing the

Attribute name	Value	Classes
AT_sibling	0x01	reference
AT_location	0x02	block, constant
AT_name	0x03	string
AT_fund_type	0x05	constant
AT_mod_fund_type	0x06	block
AT_user_def_type	0x07	reference
AT_mod_u_d_type	0x08	block
AT_ordering	0x09	constant
AT_subscr_data	0x0a	block
AT_byte_size	0x0b	constant
AT_bit_offset	0x0c	constant
AT_bit_size	0x0d	constant
AT_element_list	0x0f	block
AT_stmt_list	0x10	constant
AT_low_pc	0x11	address
AT_high_pc	0x12	address
AT_language	0x13	constant
AT_member	0x14	reference
AT_discr	0x15	reference
AT_discr_value	0x16	block
AT_visibility	0x17	constant
AT_import	0x18	reference
AT_string_length	0x19	block, constant
AT_common_reference	0x1a	reference
AT_comp_dir	0x1b	string
AT_const_value	0x1c	string, constant, block
AT_containing_type	0x1d	reference
AT_default_value	0x1e	reference
AT_friends	0x1f	block
AT_inline	0x20	constant
AT_is_optional	0x21	flag
AT_lower_bound	0x22	constant, reference
AT_producer	0x25	string
AT_prototyped	0x27	flag
AT_return_addr	0x2a	block, constant
AT_start_scope	0x2c	constant
AT_stride_size	0x2e	constant
AT_upper_bound	0x2f	constant, reference

Figure 17. Attribute encodings, part 1

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 20 gives some examples of FORM_SDATA *numbers.*

Attribute name	Value	Classes
AT_abstract_origin	0x31	reference
AT_accessibility	0x32	constant
AT_address_class	0x33	constant
AT_artificial	0x34	flag
AT_calling_convention	0x35	constant
AT_data_member_offset	0x36	block, reference
AT_decl_column	0x37	constant
AT_decl_file	0x38	constant
AT_decl_line	0x39	constant
AT_declaration	0x3a	flag
AT_discr_list	0x3b	block
AT_frame_base	0x3c	block, constant
AT_identifier_case	0x3d	constant
AT_macro_info	0x3e	constant
AT_namelist_items	0x3f	block
AT_priority	0x40	string, constant, block
AT_segment	0x41	block, constant
AT_static_link	0x42	block, constant
AT_use_location	0x43	block, constant
AT_variable_parameter	0x44	flag
AT_virtuality	0x45	constant
AT_vtable_elem_location	0x46	block, reference
AT_lo_user	0x2000	—
AT_hi_user	0x3fff	—

Figure 18. Attribute encodings, part 2

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 19. Examples of unsigned LEB128 encodings

4.5 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.

4.6 File Constraints

All debugging information entries in a relocatable object file, executable object or shared object are required to be physically contiguous.

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 20. Examples of signed LEB128 encodings

Appendix 4 gives algorithms for encoding and decoding these forms.

4.7 Location Descriptions

4.7.1 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 21 and 22.

Operation	Code	No. of Operands	Notes
OP_ADDR	0x03	1	constant address (size target specific)
OP_DEREF	0x06	0	
OP_CONST1U	0x08	1	1-byte constant
OP_CONST1S	0x09	1	1-byte constant
OP_CONST2U	0x0a	1	2-byte constant
OP_CONST2S	0x0b	1	2-byte constant
OP_CONST4U	0x0c	1	4-byte constant
OP_CONST4S	0x0d	1	4-byte constant
OP_CONST8U	0x0e	1	8-byte constant
OP_CONST8S	0x0f	1	8-byte constant
OP_CONSTU	0x10	1	ULEB128 constant
OP_CONSTS	0x11	1	SLEB128 constant
OP_DUP	0x12	0	
OP_DROP	0x13	0	
OP_OVER	0x14	0	
OP_PICK	0x15	1	1-byte stack index
OP_SWAP	0x16	0	
OP_ROT	0x17	0	
OP_XDEREF	0x18	0	

Figure 21. Location operation encodings, part 1

Operation	Code	No. of Operands	Notes
OP_ABS	0x19	0	
OP_AND	0x1a	0	
OP_DIV	0x1b	0	
OP_MINUS	0x1c	0	
OP_MOD	0x1d	0	
OP_MUL	0x1e	0	
OP_NEG	0x1f	0	
OP_NOT	0x20	0	
OP_OR	0x21	0	
OP_PLUS	0x22	0	
OP_PLUS_UCONST	0x23	1	ULEB128 addend
OP_SHL	0x24	0	
OP_SHR	0x25	0	
OP_SHRA	0x26	0	
OP_XOR	0x27	0	
OP_BRA	0x28	1	signed 2-byte constant
OP_EQ	0x29	0	
OP_GE	0x2a	0	
OP_GT	0x2b	0	
OP_LE	0x2c	0	
OP_LT	0x2d	0	
OP_NE	0x2e	0	
OP_SKIP	0x2f	1	signed 2-byte constant
OP_LIT0	0x30	0	literals 0..31 = (OP_LIT0|literal)
OP_LIT1	0x31	0	
...			
OP_LIT31	0x4f		
OP_REG0	0x50	0	reg 0..31 = (OP_REG0|regnum)
OP_REG1	0x51	0	
...			
OP_REG31	0x6f	0	
OP_BREG0	0x70	1	SLEB128 offset
OP_BREG1	0x71	1	base reg 0..31 = (OP_BREG0|regnum)
...			
OP_BREG31	0x8f	1	
OP_REGX	0x90	1	ULEB128 register
OP_FBREG	0x91	1	SLEB128 offset
OP_BREGX	0x92	2	ULEB128 register followed by SLEB128 offset
OP_PIECE	0x93	1	ULEB128 size of piece addressed
OP_NOP	0x94	0	
OP_lo_user	0xe0		
OP_hi_user	0xff		

Figure 22. Location operation encodings, part 2

4.7.2 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 FORM_ADDR on the target machine.

4.8 Fundamental Types

The fundamental type attribute is represented as a 2-byte constant. The encodings for the required fundamental type values are listed in Figure 23. For values in the range from FT_lo_user through FT_hi_user, inclusive, the low order byte of the fundamental type code contains the size in bytes of objects having the specified type, if the size is constant, otherwise the low order byte contains 0.

Type name	Value
FT_char	0x0001
FT_signed_char	0x0002
FT_unsigned_char	0x0003
FT_short	0x0004
FT_signed_short	0x0005
FT_unsigned_short	0x0006
FT_integer	0x0007
FT_signed_integer	0x0008
FT_unsigned_integer	0x0009
FT_long	0x000a
FT_signed_long	0x000b
FT_unsigned_long	0x000c
FT_pointer	0x000d
FT_float	0x000e
FT_dbl_prec_float	0x000f
FT_ext_prec_float	0x0010
FT_complex	0x0011
FT_dbl_prec_complex	0x0012
FT_void	0x0014
FT_boolean	0x0015
FT_ext_prec_complex	0x0016
FT_label	0x0017
FT_lo_user	0x8000
FT_hi_user	0xffff

Figure 23. Type encodings

4.9 Type Modifiers

Modifier types are represented by a single byte value. The encodings for the required values are given in Figure 24.

4.10 Accessibility Codes

The encodings of the constants used in the AT_accessibility attribute are given in Figure 25.

4.11 Visibility Codes

The encodings of the constants used in the AT_visibility attribute are given in Figure 26.

Modifier name	Value
MOD_pointer_to	0x01
MOD_reference_to	0x02
MOD_const	0x03
MOD_volatile	0x04
MOD_lo_user	0x80
MOD_hi_user	0xff

Figure 24. Type modifier encodings

Accessibility code name	Value
ACCESS_public	1
ACCESS_protected	2
ACCESS_private	3

Figure 25. Accessibility encodings

Visibility code name	Value
VIS_local	1
VIS_exported	2
VIS_qualified	3

Figure 26. Visibility encodings

4.12 Virtuality Codes

The encodings of the constants used in the AT_virtuality attribute are given in Figure 27.

Virtuality code name	Value
VIRTUALITY_none	0
VIRTUALITY_virtual	1
VIRTUALITY_pure_virtual	2

Figure 27. Virtuality encodings

4.13 Source Languages

The encodings for source languages are given in figure 28.

Language name	Value
LANG_C89	0x0001
LANG_C	0x0002
LANG_ADA83	0x0003
LANG_C_PLUS_PLUS	0x0004
LANG_COBOL74	0x0005
LANG_COBOL85	0x0006
LANG_FORTRAN77	0x0007
LANG_FORTRAN90	0x0008
LANG_PASCAL83	0x0009
LANG_MODULA2	0x000a
LANG_lo_user	0x8000
LANG_hi_user	0xffff

Figure 28. Language encodings

4.14 Identifier Case

The encodings of the constants used in the `AT_identifier_case` attribute are given in Figure 29.

Identifier Case Name	Value
`ID_case_sensitive`	`0`
`ID_up_case`	`1`
`ID_down_case`	`2`

Figure 29. Identifier case encodings

4.15 Calling Convention Encodings

The encodings for the values of the `AT_calling_convention` attribute are given in Figure 30.

Calling Convention Name	Value
`CC_normal`	`0x1`
`CC_program`	`0x2`
`CC_nocall`	`0x3`
`CC_lo_user`	`0x40`
`CC_hi_user`	`0xff`

Figure 30. Calling convention encodings

4.16 Inline Codes

The encodings of the constants used in the `AT_inline` attribute are given in Figure 31.

Inline Code Name	Value
`INL_not_inlined`	`0`
`INL_inlined`	`1`
`INL_declared_not_inlined`	`2`
`INL_declared_inlined`	`3`

Figure 31. Inline encodings

4.17 Array Type Entries

4.17.1 Array Ordering

The encodings for the values of the order attributes of arrays is given in Figure 32.

Ordering name	Value
`ORD_row_major`	`0`
`ORD_col_major`	`1`

Figure 32. Ordering encodings

4.17.2 Array Subscripts

The components of a subscript data value are represented as follows:

Format specifier: 1-byte constant. See Figure 33 for the layout.

Format spec index type: 4-bit constant encoded in the high order 4 bits of the specifier byte. The encodings are given in Figure 34.

Format spec lower bound: 2-bit constant encoded in the third and fourth bits (reading from the low order bit) of the specifier byte. The encodings are given in Figure 35.

Format spec upper bound: 2-bit constant encoded in the low order two bits of the specifier byte. The encodings are given in Figure 35.

Fundamental type: 2-byte constant.

User-defined type: 4-byte reference.

Subscript bound index: Signed LEB128 constant.

Subscript bound location expression: Data block with a 2-byte length.

Subscript bound location list: a 4-byte offset into the .debug_loc section.

Element type: Fundamental type, user-defined type, modified fundamental type, or modified user-defined type, preceded by the corresponding unsigned LEB128 attribute name encoding and the unsigned LEB128 attribute form encoding.

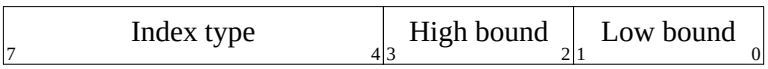


Figure 33. Format Specifier Layout

Format Name	Value
AR_index_ft	0b0001
AR_index_ut	0b0010
AR_et	0b0011

Figure 34. Array Index Type Format Descriptors

Format Name	Value
AR_bound_constant	0b01
AR_bound_locexpr	0b10
AR_bound_loclist	0b11

Figure 35. Array Upper/Lower Bound Format Descriptors

4.18 Friend Lists

The list of friends to a structure, union or class type that is the value of an AT_friends attribute is contained in a block of contiguous bytes. Each entry in the list is a 4-byte reference to another debugging information entry.

4.19 Discriminant Lists

The descriptors used in the AT_dicsr_list attribute are encoded as 1-byte constants. The defined values are presented in Figure 36.

4.20 Enumeration Type Entries

Information about the enumeration literals is stored in an element list attribute whose value is a list of data elements stored in a block of contiguous bytes.

Descriptor Name	Value
DSC_label	0
DSC_range	1

Figure 36. Discriminant descriptor encodings

Each data item in an element list consists of a signed constant whose size is the same as the size of the fundamental type FT_long on the target machine, followed by a null-terminated block of contiguous bytes.

4.21 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 1-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.

4.22 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 (in the size appropriate for the given architecture) and of a 4-byte constant length. Each set of tuples is terminated by a 0 for the address and 0 for the length.

4.23 Line Number Information

The sizes of the integers used in the line number information section 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 37. The encodings for the pre-defined extended opcodes are given in Figure 38.

4.24 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 MACINFO_vend_ext

Opcode Name	Value
LNS_copy	1
LNS_advance_pc	2
LNS_advance_line	3
LNS_set_file	4
LNS_set_column	5
LNS_negate_stmt	6
LNS_set_basic_block	7
LNS_const_add_pc	8
LNS_fixed_advance_pc	9

Figure 37. Standard Opcode Encodings

Opcode Name	Value
LNE_end_sequence	1
LNE_set_address	2
LNE_define_file	3

Figure 38. Extended Opcode Encodings

Macinfo Type Name	Value
MACINFO_define	1
MACINFO_undef	2
MACINFO_start_file	3
MACINFO_end_file	4
MACINFO_vend_ext	255

Figure 39. Macinfo Type Encodings

entry. The macinfo type is encoded as a single byte. The encodings are given in Figure 39.

4.25 Dependencies

The debugging information in this format is intended to exist in the `.debug_abbrev`, `.debug_aranges`, `.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.

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
TAG_array_type	DECL†
	AT_abstract_origin
	AT_accessibility
	AT_byte_size
	AT_declaration
	AT_name
	AT_ordering
	AT_sibling
	AT_start_scope
	AT_stride_size
	AT_subscr_data
	AT_visibility
TAG_class_type	DECL
	AT_abstract_origin
	AT_accessibility
	AT_byte_size
	AT_declaration
	AT_friends
	AT_name
	AT_sibling
	AT_start_scope
	AT_visibility

† AT_decl_column, AT_decl_file, AT_decl_line.

Appendix 1 (cont'd) -- Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
TAG_common_block	DECL AT_declaration AT_location AT_name AT_sibling AT_visibility
TAG_common_inclusion	DECL AT_common_reference AT_declaration AT_sibling AT_visibility
TAG_compile_unit	AT_comp_dir AT_identifier_case AT_high_pc AT_language AT_low_pc AT_macro_info AT_name AT_producer AT_sibling AT_stmt_list
TAG_entry_point	FT/MFT/UDT/MUDT† AT_address_class AT_low_pc AT_name AT_return_addr AT_segment AT_sibling AT_static_link
TAG_enumeration_type	DECL AT_abstract_origin AT_accessibility AT_byte_size AT_declaration AT_element_list AT_name AT_sibling AT_start_scope AT_visibility

† AT_fund_type, AT_mod_fund_type, AT_user_def_type or AT_mod_u_d_type.

Appendix 1 (cont'd) -- Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
TAG_formal_parameter	DECL
	FT/MFT/UDT/MUDT
	AT_abstract_origin
	AT_artificial
	AT_default_value
	AT_is_optional
	AT_location
	AT_name
	AT_segment
	AT_sibling
	AT_variable_parameter
TAG_global_subroutine	DECL
	FT/MFT/UDT/MUDT
	AT_abstract_origin
	AT_accessibility
	AT_address_class
	AT_artificial
	AT_calling_convention
	AT_declaration
	AT_frame_base
	AT_high_pc
	AT_inline
	AT_low_pc
	AT_member
	AT_name
	AT_prototyped
	AT_return_addr
	AT_segment
	AT_sibling
	AT_start_scope
	AT_static_link
	AT_visibility
	AT_virtuality
	AT_vtable_elem_location

Appendix 1 (cont'd) -- Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
TAG_global_variable	DECL
	FT/MFT/UDT/MUDT
	AT_accessibility
	AT_constant_value
	AT_declaration
	AT_location
	AT_member
	AT_name
	AT_segment
	AT_sibling
	AT_start_scope
	AT_visibility
TAG_imported_declaration	DECL
	AT_accessibility
	AT_import
	AT_name
	AT_sibling
	AT_start_scope
TAG_inheritance	DECL
	AT_accessibility
	AT_location
	AT_sibling
	AT_user_def_type
	AT_virtuality
TAG_inlined_subroutine	AT_abstract_origin
	AT_high_pc
	AT_low_pc
	AT_segment
	AT_sibling
	AT_return_addr
	AT_start_scope

Appendix 1 (cont'd) -- Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
TAG_label	AT_abstract_origin AT_low_pc AT_name AT_segment AT_start_scope AT_sibling
TAG_lexical_block	AT_abstract_origin AT_high_pc AT_low_pc AT_name AT_segment AT_sibling
TAG_local_variable	DECL FT/MFT/UDT/MUDT AT_abstract_origin AT_accessibility AT_constant_value AT_declaration AT_location AT_name AT_segment AT_sibling AT_start_scope AT_visibility
TAG_member	DECL FT/MFT/UDT/MUDT AT_accessibility AT_byte_size AT_bit_offset AT_bit_size AT_data_member_offset AT_declaration AT_name AT_sibling AT_visibility

Appendix 1 (cont'd) -- Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
TAG_module	DECL
	AT_accessibility
	AT_declaration
	AT_high_pc
	AT_low_pc
	AT_name
	AT_priority
	AT_segment
	AT_sibling
	AT_visibility
TAG_namelist	DECL
	AT_accessibility
	AT_abstract_origin
	AT_declaration
	AT_namelist_items
	AT_sibling
	AT_visibility
TAG_pointer_type	DECL
	FT/MFT/UDT/MUDT
	AT_abstract_origin
	AT_accessibility
	AT_address_class
	AT_declaration
	AT_name
	AT_start_scope
	AT_sibling
	AT_visibility
TAG_ptr_to_member_type	DECL
	FT/MFT/UDT/MUDT
	AT_abstract_origin
	AT_address_class
	AT_containing_type
	AT_declaration
	AT_name
	AT_sibling
	AT_use_location
	AT_visibility

Appendix 1 (cont'd) -- Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
TAG_reference_type	DECL FT/MFT/UDT/MUDT AT_abstract_origin AT_accessibility AT_address_class AT_declaration AT_name AT_start_scope AT_sibling AT_visibility
TAG_set_type	DECL FT/MFT/UDT/MUDT AT_abstract_origin AT_accessibility AT_byte_size AT_declaration AT_name AT_start_scope AT_sibling AT_visibility
TAG_string_type	DECL AT_accessibility AT_abstract_origin AT_byte_size AT_declaration AT_name AT_segment AT_sibling AT_start_scope AT_string_length AT_visibility
TAG_structure_type	DECL AT_abstract_origin AT_accessibility AT_byte_size AT_declaration AT_friends AT_name AT_sibling AT_start_scope AT_visibility

Appendix 1 (cont'd) -- Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
TAG_subrange_type	DECL
	FT/MFT/UDT/MUDT
	AT_abstract_origin
	AT_accessibility
	AT_byte_size
	AT_declaration
	AT_lower_bound
	AT_name
	AT_sibling
	AT_upper_bound
	AT_visibility
TAG_subroutine	DECL
	FT/MFT/UDT/MUDT
	AT_abstract_origin
	AT_accessibility
	AT_address_class
	AT_artificial
	AT_calling_convention
	AT_declaration
	AT_frame_base
	AT_high_pc
	AT_inline
	AT_low_pc
	AT_name
	AT_prototyped
	AT_return_addr
	AT_segment
	AT_start_scope
	AT_static_link
	AT_sibling
	AT_visibility

Appendix 1 (cont'd) -- Current Attributes by Tag Value

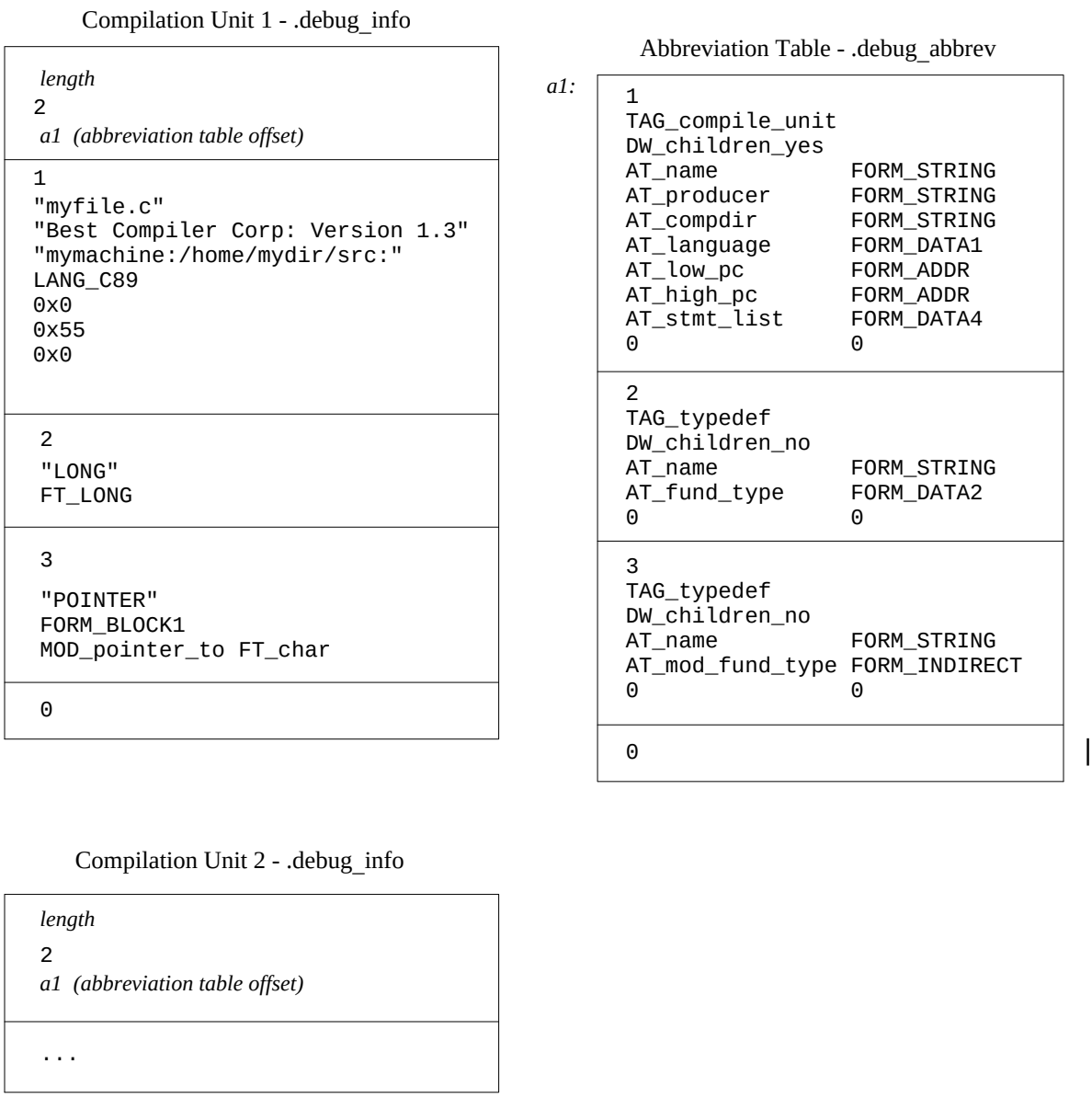
TAG NAME	APPLICABLE ATTRIBUTES
TAG_subroutine_type	DECL FT/MFT/UDT/MUDT AT_abstract_origin AT_accessibility AT_address_class AT_declaration AT_name AT_prototyped AT_sibling AT_start_scope AT_visibility
TAG_typedef	DECL FT/MFT/UDT/MUDT AT_abstract_origin AT_accessibility AT_declaration AT_name AT_sibling AT_start_scope AT_visibility
TAG_union_type	DECL AT_abstract_origin AT_accessibility AT_byte_size AT_declaration AT_friends AT_name AT_sibling AT_start_scope AT_visibility
TAG_unspecified_parameters	DECL AT_abstract_origin AT_artificial AT_sibling

Appendix 1 (cont'd) -- Current Attributes by Tag Value

TAG NAME	APPLICABLE ATTRIBUTES
TAG_variant	DECL AT_accessibility AT_abstract_origin AT_declaration AT_discr_list AT_discr_value AT_sibling
TAG_variant_part	DECL FT/MFT/UDT/MUDT AT_accessibility AT_abstract_origin AT_declaration AT_discr AT_sibling
TAG_with_statement	AT_accessibility AT_address_class AT_declaration AT_high_pc AT_location AT_low_pc AT_segment AT_sibling AT_user_def_type 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.



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
`ADVANCE_PC`	`LEB128(0x239)`	`0x2, 0xb9, 0x04`
`SPECIAL(2, 0)`		`0xb`
`SPECIAL(2, 3)`		`0x38`
`SPECIAL(1, 8)`		`0x82`
`SPECIAL(1, 7)`		`0x73`
`ADVANCE_PC`	`LEB128(2)`	`0x2, 0x2`
`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
FIXED_ADVANCE_PC	0x239	0x9, 0x39, 0x2
SPECIAL(2, 0)		0xb
FIXED_ADVANCE_PC	0x3	0x9, 0x3, 0x0
SPECIAL(2, 0)		0xb
FIXED_ADVANCE_PC	0x8	0x9, 0x8, 0x0
SPECIAL(1, 0)		0xa
FIXED_ADVANCE_PC	0x7	0x9, 0x7, 0x0
SPECIAL(1, 0)		0xa
FIXED_ADVANCE_PC	0x2	0x9, 0x2, 0x0
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);
```

Table of Contents

1. INTRODUCTION 1
   1.1 Purpose and Scope 1
   1.2 Overview 1
   1.3 Vendor Extensibility 2
   1.4 Changes from Version 1 2

2. GENERAL DESCRIPTION 3
   2.1 The Debugging Information Entry 3
   2.2 Attribute Types 3
   2.3 Relationship of Debugging Information Entries 4
   2.4 Location Descriptions 5
   2.5 Type Attributes 13
   2.6 Non-Defining Declarations 16
   2.7 Declaration Coordinates 17
   2.8 Identifier Names 17

3. DEBUGGING INFORMATION ENTRIES 19
   3.1 Compilation Unit Entries 19
   3.2 Modules 20
   3.3 Subroutine and Entry Point Entries 21
   3.4 Lexical Block Entries 26
   3.5 Label Entries 26
   3.6 Program Variable Entries 27
   3.7 Common Block Entries 28
   3.8 Imported Declaration Entries 28
   3.9 User-Defined Type Entries 29
   3.10 With Statement Entries 38
   3.11 Namelist Entries 38
   3.12 Accelerated Access 39
   3.13 Line Number Information 40
   3.14 Macro Information 47

4. DATA REPRESENTATION 51
   4.1 Vendor Extensibility 51
   4.2 Reserved Error Values 51
   4.3 Format of Debugging Information 51
   4.4 Variable Length Data 55
   4.5 Executable Objects and Shared Objects 57
   4.6 File Constraints 57
   4.7 Location Descriptions 58
   4.8 Fundamental Types 60
   4.9 Type Modifiers 60
   4.10 Accessibility Codes 60
   4.11 Visibility Codes 60
   4.12 Virtuality Codes 61

4.13 Source Languages 61
4.14 Identifier Case 62
4.15 Calling Convention Encodings 62
4.16 Inline Codes 62
4.17 Array Type Entries 62
4.18 Friend Lists 63
4.19 Discriminant Lists 63
4.20 Enumeration Type Entries 63
4.21 Name Lookup Table 64
4.22 Address Range Table 64
4.23 Line Number Information 64
4.24 Macro Information 64
4.25 Dependencies 65

Appendix 1 -- Current Attributes by Tag Value 67

Appendix 2 -- Organization of Debugging Information 77

Appendix 3 -- Statement Program Examples 79

Appendix 4 -- Encoding and decoding variable length data 81

List of Figures

Figure 1. Tag names 3
Figure 2. Attribute names 4
Figure 3. Fundamental types 13
Figure 4. Type modifiers 14
Figure 5. Accessibility codes 15
Figure 6. Visibility codes 15
Figure 7. Virtuality codes 15
Figure 8. Example address class codes 16
Figure 9. Language names 19
Figure 10. Identifier case codes 20
Figure 11. Inline codes 24
Figure 12. Array ordering 30
Figure 13. Discriminant descriptor values 35
Figure 14. Tag encodings 53
Figure 15. Child determination encodings 54
Figure 16. Attribute form encodings 55
Figure 17. Attribute encodings, part 1 56
Figure 18. Attribute encodings, part 2 57
Figure 19. Examples of unsigned LEB128 encodings 57
Figure 20. Examples of signed LEB128 encodings 58
Figure 21. Location operation encodings, part 1 58
Figure 22. Location operation encodings, part 2 59
Figure 23. Type encodings 60
Figure 24. Type modifier encodings 61
Figure 25. Accessibility encodings 61
Figure 26. Visibility encodings 61
Figure 27. Virtuality encodings 61
Figure 28. Language encodings 61

Figure 29. Identifier case encodings 62
Figure 30. Calling convention encodings 62
Figure 31. Inline encodings 62
Figure 32. Ordering encodings 62
Figure 33. Format Specifier Layout 63
Figure 34. Array Index Type Format Descriptors 63
Figure 35. Array Upper/Lower Bound Format Descriptors 63
Figure 36. Discriminant descriptor encodings 64
Figure 37. Standard Opcode Encodings 65
Figure 38. Extended Opcode Encodings 65
Figure 39. Macinfo Type Encodings 65