DWARF Debugging Information Format

UNIX International
Programming Languages SIG
Revision: 2.0.0 (July 27, 1993)

Published by:

UNIX International
Waterview Corporate Center
20 Waterview Boulevard
Parsippany, NJ 07054

for further information, contact:
Vice President of Marketing

Phone: +1 201-263-8400
Fax: +1 201-263-8401

NOTICE:

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

FOREWORD

This document specifies the second generation of symbolic debugging information based on the DWARF format that has been developed by the UNIX International Programming Languages Special Interest Group (SIG). It is being circulated for industry review. The first version of the DWARF specification was published by UNIX International in January, 1992. The current version adds significant new functionality, but its main thrust is to achieve a much denser encoding of the DWARF information. Because of the new encoding, DWARF Version 2 is not binary compatible with DWARF Version 1.

At this point, the SIG believes that this document sufficiently supports the debugging needs of C, C++, FORTRAN 77, Fortran90, Modula2 and Pascal, and we have released it for public comment. We will accept comments on this document until September 30, 1994. Comments may be directed via email to the SIG mailing list (plsig@ui.org). If you are unable to send email, paper mail, FAX, or machine readable copy on UNIX, MS-DOS, or Macintosh compatible media can be sent to UNIX International at the address listed below, and will be forwarded to the SIG.

UNIX International
Waterview Corporate Center
20 Waterview Boulevard
Parsippany, NJ 07054
Phone: +1 201-263-8400
Fax: +1 201-263-8401

1. INTRODUCTION

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

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

1.1 Purpose and Scope

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

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

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

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

1.2 Overview

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

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

Section eight describes some future directions for the DWARF specification.

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

1.3 Vendor Extensibility

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

1.4 Changes from Version 1

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

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

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

2. GENERAL DESCRIPTION

2.1 The Debugging Information Entry

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

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

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

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

Figure 1. Tag names

2.2 Attribute Types

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

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

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

Figure 2. Attribute names

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

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

block — An arbitrary number of uninterpreted bytes of data.

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

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

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

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

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

2.3 Relationship of Debugging Information Entries

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

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

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

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

2.4 Location Descriptions

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

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

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

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

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

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

2.4.1 Location Expressions

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

2.4.2 Register Name Operators

The following operations can be used to name a register.

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

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

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

2.4.3 Addressing Operations

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

2.4.3.1 Literal Encodings

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4.3.2 Register Based Addressing

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

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

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

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

2.4.3.3 Stack Operations

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

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

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

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

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

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

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

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

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

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

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

2.4.3.4 Arithmetic and Logical Operations

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

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

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

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

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

5. DW_OP_mod
The DW_OP_mod operation pops the top two stack values and pushes the result of the

calculation: former second stack entry modulo the former top of the stack.

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

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

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

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

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

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

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

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

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

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

2.4.3.5 Control Flow Operations

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

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

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

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

2.4.3.6 Special Operations

There are two special operations currently defined:

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

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

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

2.4.4 Sample Stack Operations

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

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

2.4.5 Example Location Expressions

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

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

`DW_OP_reg3`
The value is in register 3.

`DW_OP_regx 54`
The value is in register 54.

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

`DW_OP_breg11 44`
Add 44 to the value in register 11 to get the address of an automatic variable instance.

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

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

`DW_OP_plus_uconst 4`
A structure member is four bytes from the start of the structure instance. The base address is assumed to be already on the stack.

`DW_OP_reg3 DW_OP_piece 4 DW_OP_reg10 DW_OP_piece 2`
A variable whose first four bytes reside in register 3 and whose next two bytes reside in register 10.

2.4.6 Location Lists

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

Each entry in a location list consists of:

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

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

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

2.5 Types of Declarations

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

2.6 Accessibility of Declarations

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

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

```
DW_ACCESS_public
DW_ACCESS_private
DW_ACCESS_protected
```

Figure 3. Accessibility codes

2.7 Visibility of Declarations

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

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

```
DW_VIS_local
DW_VIS_exported
DW_VIS_qualified
```

Figure 4. Visibility codes

2.8 Virtuality of Declarations

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

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

```
DW_VIRTUALITY_none
DW_VIRTUALITY_virtual
DW_VIRTUALITY_pure_virtual
```

Figure 5. Virtuality codes

2.9 Artificial Entries

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

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

2.10 Target-Specific Addressing Information

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

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

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

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

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

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

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

Figure 6. Example address class codes

2.11 Non-Defining Declarations

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

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

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

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

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

2.12 Declaration Coordinates

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

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

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

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

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

2.13 Identifier Names

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

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

3. PROGRAM SCOPE ENTRIES

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

3.1 Compilation Unit Entries

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

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

The compilation unit entry may have the following attributes:

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

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

   The presence of low and high pc attributes in a compilation unit entry imply that the code generated for that compilation unit is contiguous and exists totally within the boundaries specified by those two attributes. If that is not the case, no low and high pc attributes should be produced.
3. A `DW_AT_name` attribute whose value is a null-terminated string containing the full or relative path name of the primary source file from which the compilation unit was derived.
4. A `DW_AT_language` attribute whose constant value is a code indicating the source language of the compilation unit. The set of language names and their meanings are given in Figure 7.

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

Figure 7. Language names

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

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

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

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

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

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

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

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

   ```
   DW_ID_case_sensitive
   DW_ID_up_case
   DW_ID_down_case
   DW_ID_case_insensitive
   ```

   Figure 8. Identifier case codes

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

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

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

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

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

compilation unit (see section 5.1).

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

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

3.2 Module Entries

Several languages have the concept of a ''module.''

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

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

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

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

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

3.3 Subroutine and Entry Point Entries

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

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

3.3.1 General Subroutine and Entry Point Information

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

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

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

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

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

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

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

3.3.2 Subroutine and Entry Point Return Types

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

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

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

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

3.3.3 Subroutine and Entry Point Locations

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

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

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

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

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

3.3.4 Declarations Owned by Subroutines and Entry Points

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

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

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

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

3.3.5 Low-Level Information

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

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

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

1. *In procedures that need location lists to locate local variables, the* `DW_AT_frame_base` *can hold the needed location list, while all variables' location descriptions can be simpler location expressions involving the frame base.*

2. *It can be used as a key in resolving "up-level" addressing with nested routines. (See* `DW_AT_static_link`*, below)*

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

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

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

1. It should compute a value that does not change during the life of the procedure, and

2. The computed value should be unique among instances of the same subroutine. (For typical `DW_AT_frame_base` use, this means that a recursive subroutine's stack frame must have non-zero size.)

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

3.3.6 Types Thrown by Exceptions

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

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

3.3.7 Function Template Instantiations

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

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

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

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

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

3.3.8 Inline Subroutines

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

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

Figure 9. Inline codes

attribute is given in Figure 9.

3.3.8.1 Abstract Instances

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

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

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

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

3.3.8.2 Concrete Inlined Instances

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3.8.3 Out-of-Line Instances of Inline Subroutines

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

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

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

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

1. The root entry for a concrete out-of-line instance of a given inline subroutine has the same tag as does its associated (abstract) inline subroutine entry (that is, it does not have the tag `DW_TAG_inlined_subroutine`).

2. The root entry for a concrete out-of-line instance tree is always directly owned by the same parent entry that also owns the root entry of the associated abstract instance.

3.4 Lexical Block Entries

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

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

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

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

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

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

3.5 Label Entries

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

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

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

3.6 With Statement Entries

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

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

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

3.7 Try and Catch Block Entries

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

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

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

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

4. DATA OBJECT AND OBJECT LIST ENTRIES

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

4.1 Data Object Entries

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
float x = 99.99;

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

                            return 0;
}
```

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

4.2 Common Block Entries

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

4.3 Imported Declaration Entries

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

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

4.4 Namelist Entries

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

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

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

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

5. TYPE ENTRIES

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

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

5.1 Base Type Entries

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

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

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

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

Figure 10. Encoding attribute values

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

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

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

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

5.2 Type Modifier Entries

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

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

Figure 11. Type modifier tags

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

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

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

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

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

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

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

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

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

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

5.3 Typedef Entries

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

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

5.4 Array Type Entries

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

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

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

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

DW_ORD_col_major
DW_ORD_row_major

Figure 12. Array ordering

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

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

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

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

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

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

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

5.5 Structure, Union, and Class Type Entries

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

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

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

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

5.5.1 General Structure Description

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

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

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

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

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

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

5.5.2 Derived Classes and Structures

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

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

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

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

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

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

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

5.5.3 Friends

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

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

5.5.4 Structure Data Member Entries

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Big-Endian

0 `j`	`k`	`m`	`n`	*pad*
31	26	20	15	7 0

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

Little-Endian

pad	`n`	`m`	`k`	`j` 0
31	23	15	10	4 0

5.5.5 Structure Member Function Entries

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

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

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

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

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

5.5.6 Class Template Instantiations

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

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

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

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

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

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

5.5.7 Variant Entries

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

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

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

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

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

`DW_DSC_label`
`DW_DSC_range`

Figure 13. Discriminant descriptor values

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

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

5.6 Enumeration Type Entries

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

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

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

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

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

5.7 Subroutine Type Entries

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

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

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

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

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

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

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

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

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

5.8 String Type Entries

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

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

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

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

5.9 Set Entries

Pascal provides the concept of a ‘‘set,’’ which represents a group of values of ordinal type.

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

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

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

5.10 Subrange Type Entries

Several languages support the concept of a ‘‘subrange’’ type object. These objects can represent a subset of the values that an object of the basis type for the subrange can represent. Subrange type entries may also be used to represent the bounds of array dimensions.

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

The subrange entry may have a `DW_AT_type` attribute to describe the type of object of whose values this subrange is a subset.

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

The subrange entry may have the attributes `DW_AT_lower_bound` and `DW_AT_upper_bound` to describe, respectively, the lower and upper bound values of the subrange. The `DW_AT_upper_bound` attribute may be replaced by a `DW_AT_count` attribute, whose value describes the number of elements in the subrange rather than the value of the last element. If a bound or count value is described by a constant not represented in the program's address space and can be represented by one of the constant attribute forms, then the value of the lower or upper bound or count attribute may be one of the constant types. Otherwise, the value of the lower or upper bound or count attribute is a reference to a debugging information entry describing an object containing the bound value or itself describing a constant value.

If either the lower or upper bound or count values are missing, the bound value is assumed to be a language-dependent default constant.

The default lower bound value for C or C++ is 0. For Fortran, it is 1. No other default values are currently defined by DWARF.

If the subrange entry has no type attribute describing the basis type, the basis type is assumed to be the same as the object described by the lower bound attribute (if it references an object). If there is no lower bound attribute, or it does not reference an object, the basis type is the type of the upper bound or count attribute (if it references an object). If there is no upper bound or count attribute or it does not reference an object, the type is assumed to be the same type, in the source language of the compilation unit containing the subrange entry, as a signed integer with the same size as an address on the target machine.

5.11 Pointer to Member Type Entries

In C++, a pointer to a data or function member of a class or structure is a unique type.

A debugging information entry representing the type of an object that is a pointer to a structure or class member has the tag `DW_TAG_ptr_to_member_type`.

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

The pointer to member entry has a `DW_AT_type` attribute to describe the type of the class or structure member to which objects of this type may point.

The pointer to member entry also has a `DW_AT_containing_type` attribute, whose value is a reference to a debugging information entry for the class or structure to whose members objects of this type may point.

Finally, the pointer to member entry has a `DW_AT_use_location` attribute whose value is a location description that computes the address of the member of the class or structure to which the pointer to member type entry can point.

The method used to find the address of a given member of a class or structure is common to any instance of that class or structure and to any instance of the pointer or member type. The method is thus associated with the type entry, rather than with each instance of the type.

The DW_AT_use_location *expression, however, cannot be used on its own, but must be used in conjunction with the location expressions for a particular object of the given pointer to member type and for a particular structure or class instance. The* DW_AT_use_location *attribute expects two values to be pushed onto the location expression stack before the* DW_AT_use_location *expression is evaluated. The first value pushed should be the value of the pointer to member object itself. The second value pushed should be the base address of the entire structure or union instance containing the member whose address is being calculated.*

So, for an expression like

```
object.*mbr_ptr
```

where mbr_ptr *has some pointer to member type, a debugger should:*

1. *Push the value of* mbr_ptr *onto the location expression stack.*
2. *Push the base address of* object *onto the location expression stack.*
3. *Evaluate the* DW_AT_use_location *expression for the type of* mbr_ptr.

5.12 File Type Entries

Some languages, such as Pascal, provide a first class data type to represent files.

A file type is represented by a debugging information entry with the tag DW_TAG_file_type. If the file type has a name, the file type entry has a DW_AT_name attribute, whose value is a null-terminated string containing the type name as it appears in the source program.

The file type entry has a DW_AT_type attribute describing the type of the objects contained in the file.

The file type entry also has a DW_AT_byte_size attribute, whose value is a constant representing the size in bytes of an instance of this file type.

6. OTHER DEBUGGING INFORMATION

This section describes debugging information that is not represented in the form of debugging information entries and is not contained within the `.debug_info` section.

6.1 Accelerated Access

A debugger frequently needs to find the debugging information for a program object defined outside of the compilation unit where the debugged program is currently stopped. Sometimes it will know only the name of the object; sometimes only the address. To find the debugging information associated with a global object by name, using the DWARF debugging information entries alone, a debugger would need to run through all entries at the highest scope within each compilation unit. For lookup by address, for a subroutine, a debugger can use the low and high pc attributes of the compilation unit entries to quickly narrow down the search, but these attributes only cover the range of addresses for the text associated with a compilation unit entry. To find the debugging information associated with a data object, an exhaustive search would be needed. Furthermore, any search through debugging information entries for different compilation units within a large program would potentially require the access of many memory pages, probably hurting debugger performance.

To make lookups of program objects by name or by address faster, a producer of DWARF information may provide two different types of tables containing information about the debugging information entries owned by a particular compilation unit entry in a more condensed format.

6.1.1 Lookup by Name

For lookup by name, a table is maintained in a separate object file section called `.debug_pubnames`. The table consists of sets of variable length entries, each set describing the names of global objects whose definitions or declarations are represented by debugging information entries owned by a single compilation unit. Each set begins with a header containing four values: the total length of the entries for that set, not including the length field itself, a version number, the offset from the beginning of the `.debug_info` section of the compilation unit entry referenced by the set and the size in bytes of the contents of the `.debug_info` section generated to represent that compilation unit. This header is followed by a variable number of offset/name pairs. Each pair consists of the offset from the beginning of the compilation unit entry corresponding to the current set to the debugging information entry for the given object, followed by a null-terminated character string representing the name of the object as given by the `DW_AT_name` attribute of the referenced debugging entry. Each set of names is terminated by zero.

In the case of the name of a static data member or function member of a C++ structure, class or union, the name presented in the `.debug_pubnames` section is not the simple name given by the `DW_AT_name` attribute of the referenced debugging entry, but rather the fully class qualified name of the data or function member.

6.1.2 Lookup by Address

For lookup by address, a table is maintained in a separate object file section called `.debug_aranges`. The table consists of sets of variable length entries, each set describing the portion of the program’s address space that is covered by a single compilation unit. Each set begins with a header containing five values:

1. The total length of the entries for that set, not including the length field itself.
2. A version number.
3. The offset from the beginning of the `.debug_info` section of the compilation unit entry referenced by the set.
4. The size in bytes of an address on the target architecture. For segmented addressing, this is the size of the offset portion of the address.
5. The size in bytes of a segment descriptor on the target architecture. If the target system uses a flat address space, this value is 0.

This header is followed by a variable number of address range descriptors. Each descriptor is a pair consisting of the beginning address of a range of text or data covered by some entry owned by the corresponding compilation unit entry, followed by the length of that range. A particular set is terminated by an entry consisting of two zeroes. By scanning the table, a debugger can quickly decide which compilation unit to look in to find the debugging information for an object that has a given address.

6.2 Line Number Information

A source-level debugger will need to know how to associate statements in the source files with the corresponding machine instruction addresses in the executable object or the shared objects used by that executable object. Such an association would make it possible for the debugger user to specify machine instruction addresses in terms of source statements. This would be done by specifying the line number and the source file containing the statement. The debugger can also use this information to display locations in terms of the source files and to single step from statement to statement.

As mentioned in section 3.1, above, the line number information generated for a compilation unit is represented in the `.debug_line` section of an object file and is referenced by a corresponding compilation unit debugging information entry in the `.debug_info` section.

If space were not a consideration, the information provided in the `.debug_line` *section could be represented as a large matrix, with one row for each instruction in the emitted object code. The matrix would have columns for:*

— *the source file name*

— *the source line number*

— *the source column number*

— *whether this instruction is the beginning of a source statement*

— *whether this instruction is the beginning of a basic block.*

Such a matrix, however, would be impractically large. We shrink it with two techniques. First, we delete from the matrix each row whose file, line and source column information is identical with that of its predecessors. Second, we design a byte-coded language for a state machine and store a stream of bytes in the object file instead of the matrix. This language can be much more compact than the matrix. When a consumer of the statement information executes, it must ‘‘run’’ the state machine to generate the matrix for each compilation unit it is interested in. The concept of an encoded matrix also leaves room for expansion. In the future, columns can be added to the matrix to encode other things that are related to individual instruction addresses.

6.2.1 Definitions

The following terms are used in the description of the line number information format:

state machine	The hypothetical machine used by a consumer of the line number information to expand the byte-coded instruction stream into a matrix of line number information.
statement program	A series of byte-coded line number information instructions representing one compilation unit.
basic block	A sequence of instructions that is entered only at the first instruction and exited only at the last instruction. We define a procedure invocation to be an exit from a basic block.
sequence	A series of contiguous target machine instructions. One compilation unit may emit multiple sequences (that is, not all instructions within a compilation unit are assumed to be contiguous).
sbyte	Small signed integer.
ubyte	Small unsigned integer.
uhalf	Medium unsigned integer.
sword	Large signed integer.
uword	Large unsigned integer.
LEB128	Variable length signed and unsigned data. See section 7.6.

6.2.2 State Machine Registers

The statement information state machine has the following registers:

`address`	The program-counter value corresponding to a machine instruction generated by the compiler.
`file`	An unsigned integer indicating the identity of the source file corresponding to a machine instruction.
`line`	An unsigned integer indicating a source line number. Lines are numbered beginning at 1. The compiler may emit the value 0 in cases where an instruction cannot be attributed to any source line.
`column`	An unsigned integer indicating a column number within a source line. Columns are numbered beginning at 1. The value 0 is reserved to indicate that a statement begins at the ‘‘left edge’’ of the line.
`is_stmt`	A boolean indicating that the current instruction is the beginning of a statement.
`basic_block`	A boolean indicating that the current instruction is the beginning of a basic block.
`end_sequence`	A boolean indicating that the current address is that of the first byte after the end of a sequence of target machine instructions.

At the beginning of each sequence within a statement program, the state of the registers is:

`address`	0
`file`	1
`line`	1
`column`	0
`is_stmt`	determined by `default_is_stmt` in the statement program prologue
`basic_block`	‘‘false’’
`end_sequence`	‘‘false’’

6.2.3 Statement Program Instructions

The state machine instructions in a statement program belong to one of three categories:

special opcodes
: These have a ubyte opcode field and no arguments. Most of the instructions in a statement program are special opcodes.

standard opcodes
: These have a ubyte opcode field which may be followed by zero or more LEB128 arguments (except for `DW_LNS_fixed_advance_pc`, see below). The opcode implies the number of arguments and their meanings, but the statement program prologue also specifies the number of arguments for each standard opcode.

extended opcodes
: These have a multiple byte format. The first byte is zero; the next bytes are an unsigned LEB128 integer giving the number of bytes in the instruction itself (does not include the first zero byte or the size). The remaining bytes are the instruction itself.

6.2.4 The Statement Program Prologue

The optimal encoding of line number information depends to a certain degree upon the architecture of the target machine. The statement program prologue provides information used by consumers in decoding the statement program instructions for a particular compilation unit and also provides information used throughout the rest of the statement program. The statement program for each compilation unit begins with a prologue containing the following fields in order:

1. `total_length` (uword)
   The size in bytes of the statement information for this compilation unit (not including the `total_length` field itself).

2. `version` (uhalf)
   Version identifier for the statement information format.

3. `prologue_length` (uword)
   The number of bytes following the `prologue_length` field to the beginning of the first byte of the statement program itself.

4. `minimum_instruction_length` (ubyte)
   The size in bytes of the smallest target machine instruction. Statement program opcodes that alter the `address` register first multiply their operands by this value.

5. `default_is_stmt` (ubyte)
   The initial value of the `is_stmt` register.

A simple code generator that emits machine instructions in the order implied by the source program would set this to ''true,'' and every entry in the matrix would represent a statement boundary. A pipeline scheduling code generator would set this to ''false'' and emit a specific statement program opcode for each instruction that represented a statement boundary.

6. `line_base` (sbyte)
   This parameter affects the meaning of the special opcodes. See below.

7. `line_range` (ubyte)
   This parameter affects the meaning of the special opcodes. See below.

8. `opcode_base` (ubyte)
   The number assigned to the first special opcode.

9. `standard_opcode_lengths` (array of ubyte)
   This array specifies the number of LEB128 operands for each of the standard opcodes. The first element of the array corresponds to the opcode whose value is 1, and the last element corresponds to the opcode whose value is `opcode_base - 1`. By increasing `opcode_base`, and adding elements to this array, new standard opcodes can be added, while allowing consumers who do not know about these new opcodes to be able to skip them.

10. `include_directories` (sequence of path names)
    The sequence contains an entry for each path that was searched for included source files in this compilation. (The paths include those directories specified explicitly by the user for the compiler to search and those the compiler searches without explicit direction). Each path entry is either a full path name or is relative to the current directory of the compilation. The current directory of the compilation is understood to be the first entry and is not explicitly represented. Each entry is a null-terminated string containing a full path name. The last entry is followed by a single null byte.

11. `file_names` (sequence of file entries)
    The sequence contains an entry for each source file that contributed to the statement information for this compilation unit or is used in other contexts, such as in a declaration coordinate or a macro file inclusion. Each entry has a null-terminated string containing the file name, an unsigned LEB128 number representing the directory index of the directory in which the file was found, an unsigned LEB128 number representing the time of last modification for the file and an unsigned LEB128 number representing the length in bytes of the file. A compiler may choose to emit LEB128(0) for the time and length fields to indicate that this information is not available. The last entry is followed by a single null byte.

    The directory index represents an entry in the `include_directories` section. The index is LEB128(0) if the file was found in the current directory of the compilation, LEB128(1) if it was found in the first directory in the `include_directories` section, and so on. The directory index is ignored for file names that represent full path names.

    The statement program assigns numbers to each of the file entries in order, beginning with 1, and uses those numbers instead of file names in the `file` register.

    A compiler may generate a single null byte for the file names field and define file names using the extended opcode `DEFINE_FILE`.

6.2.5 The Statement Program

As stated before, the goal of a statement program is to build a matrix representing one compilation unit, which may have produced multiple sequences of target-machine instructions. Within a sequence, addresses may only increase. (Line numbers may decrease in cases of pipeline scheduling.)

6.2.5.1 Special Opcodes

Each 1-byte special opcode has the following effect on the state machine:

1. Add a signed integer to the `line` register.
2. Multiply an unsigned integer by the `minimum_instruction_length` field of the statement program prologue and add the result to the `address` register.
3. Append a row to the matrix using the current values of the state machine registers.
4. Set the `basic_block` register to ‘‘false.’’

All of the special opcodes do those same four things; they differ from one another only in what values they add to the `line` and `address` registers.

Instead of assigning a fixed meaning to each special opcode, the statement program uses several parameters in the prologue to configure the instruction set. There are two reasons for this. First, although the opcode space available for special opcodes now ranges from 10 through 255, the lower bound may increase if one adds new standard opcodes. Thus, the `opcode_base` *field of the statement program prologue gives the value of the first special opcode. Second, the best choice of special-opcode meanings depends on the target architecture. For example, for a RISC machine where the compiler-generated code interleaves instructions from different lines to schedule the pipeline, it is important to be able to add a negative value to the* `line` *register to express the fact that a later instruction may have been emitted for an earlier source line. For a machine where pipeline scheduling never occurs, it is advantageous to trade away the ability to decrease the* `line` *register (a standard opcode provides an alternate way to decrease the line number) in return for the ability to add larger positive values to the* `address` *register. To permit this variety of strategies, the statement program prologue defines a* `line_base` *field that specifies the minimum value which a special opcode can add to the* `line` *register and a* `line_range` *field that defines the range of values it can add to the* `line` *register.*

A special opcode value is chosen based on the amount that needs to be added to the `line` and `address` registers. The maximum line increment for a special opcode is the value of the `line_base` field in the prologue, plus the value of the `line_range` field, minus 1 (`line base + line range - 1`). If the desired line increment is greater than the maximum line increment, a standard opcode must be used instead of a special opcode. The ‘‘address advance’’ is calculated by dividing the desired address increment by the `minimum_instruction_length` field from the prologue. The special opcode is then calculated using the following formula:

```
        opcode = (desired line increment - line_base) +
                  (line_range * address advance) + opcode_base
```

If the resulting opcode is greater than 255, a standard opcode must be used instead.

To decode a special opcode, subtract the `opcode_base` from the opcode itself. The amount to increment the `address` register is the adjusted opcode divided by the `line_range`. The amount to increment the `line` register is the `line_base` plus the result of the adjusted opcode modulo the `line_range`. That is,

```
line increment = line_base + (adjusted opcode % line_range)
```

As an example, suppose that the `opcode_base` *is 16,* `line_base` *is -1 and* `line_range` *is 4. This means that we can use a special opcode whenever two successive rows in the matrix have source line numbers differing by any value within the range [-1, 2] (and, because of the limited number of opcodes available, when the difference between addresses is within the range [0, 59]).*

The opcode mapping would be:

Opcode	Line advance	Address advance
16	-1	0
17	0	0
18	1	0
19	2	0
20	-1	1
21	0	1
22	1	1
23	2	1
253	0	59
254	1	59
255	2	59

There is no requirement that the expression `255 - line_base + 1` be an integral multiple of `line_range`.

6.2.5.2 Standard Opcodes

There are currently 9 standard ubyte opcodes. In the future additional ubyte opcodes may be defined by setting the `opcode_base` field in the statement program prologue to a value greater than 10.

1. `DW_LNS_copy`
   Takes no arguments. Append a row to the matrix using the current values of the state-machine registers. Then set the `basic_block` register to ‘‘false.’’

2. `DW_LNS_advance_pc`
   Takes a single unsigned LEB128 operand, multiplies it by the `minimum_instruction_length` field of the prologue, and adds the result to the `address` register of the state machine.

3. `DW_LNS_advance_line`
   Takes a single signed LEB128 operand and adds that value to the `line` register of the state machine.

4. `DW_LNS_set_file`
   Takes a single unsigned LEB128 operand and stores it in the `file` register of the state machine.

5. `DW_LNS_set_column`
   Takes a single unsigned LEB128 operand and stores it in the `column` register of the state machine.

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

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

8. `DW_LNS_const_add_pc`
   Takes no arguments. Add to the `address` register of the state machine the address increment value corresponding to special opcode 255.

   The motivation for `DW_LNS_const_add_pc` *is this: when the statement program needs to advance the address by a small amount, it can use a single special opcode, which occupies a single byte. When it needs to advance the address by up to twice the range of the last special opcode, it can use* `DW_LNS_const_add_pc` *followed by a special opcode, for a total of two bytes. Only if it needs to advance the address by more than twice that range will it need to use both* `DW_LNS_advance_pc` *and a special opcode, requiring three or more bytes.*

9. `DW_LNS_fixed_advance_pc`
   Takes a single uhalf operand. Add to the `address` register of the state machine the value of the (unencoded) operand. This is the only extended opcode that takes an argument that is not a variable length number.

   The motivation for `DW_LNS_fixed_advance_pc` *is this: existing assemblers cannot emit* `DW_LNS_advance_pc` *or special opcodes because they cannot encode LEB128 numbers or judge when the computation of a special opcode overflows and requires the use of* `DW_LNS_advance_pc`. *Such assemblers, however, can use* `DW_LNS_fixed_advance_pc` *instead, sacrificing compression.*

6.2.5.3 Extended Opcodes

There are three extended opcodes currently defined. The first byte following the length field of the encoding for each contains a sub-opcode.

1. `DW_LNE_end_sequence`
   Set the `end_sequence` register of the state machine to ‘‘true’’ and append a row to the matrix using the current values of the state-machine registers. Then reset the registers to the initial values specified above.

   Every statement program sequence must end with a `DW_LNE_end_sequence` instruction which creates a row whose address is that of the byte after the last target machine instruction of the sequence.

2. `DW_LNE_set_address`
   Takes a single relocatable address as an operand. The size of the operand is the size appropriate to hold an address on the target machine. Set the `address` register to the value given by the relocatable address.

   All of the other statement program opcodes that affect the `address` *register add a delta to it. This instruction stores a relocatable value into it instead.*

3. `DW_LNE_define_file`
   Takes 4 arguments. The first is a null terminated string containing a source file name. The second is an unsigned LEB128 number representing the directory index of the directory in

which the file was found. The third is an unsigned LEB128 number representing the time of last modification of the file. The fourth is an unsigned LEB128 number representing the length in bytes of the file. The time and length fields may contain LEB128(0) if the information is not available.

The directory index represents an entry in the `include_directories` section of the statement program prologue. The index is LEB128(0) if the file was found in the current directory of the compilation, LEB128(1) if it was found in the first directory in the `include_directories` section, and so on. The directory index is ignored for file names that represent full path names.

The files are numbered, starting at 1, in the order in which they appear; the names in the prologue come before names defined by the `DW_LNE_define_file` instruction. These numbers are used in the the `file` register of the state machine.

Appendix 3 gives some sample statement programs.

6.3 Macro Information

Some languages, such as C and C++, provide a way to replace text in the source program with macros defined either in the source file itself, or in another file included by the source file. Because these macros are not themselves defined in the target language, it is difficult to represent their definitions using the standard language constructs of DWARF. The debugging information therefore reflects the state of the source after the macro definition has been expanded, rather than as the programmer wrote it. The macro information table provides a way of preserving the original source in the debugging information.

As described in section 3.1, the macro information for a given compilation unit is represented in the `.debug_macinfo` section of an object file. The macro information for each compilation unit is represented as a series of ‘‘macinfo’’ entries. Each macinfo entry consists of a ‘‘type code’’ and up to two additional operands. The series of entries for a given compilation unit ends with an entry containing a type code of 0.

6.3.1 Macinfo Types

The valid macinfo types are as follows:

`DW_MACINFO_define`	A macro definition.
`DW_MACINFO_undef`	A macro un-definition.
`DW_MACINFO_start_file`	The start of a new source file inclusion.
`DW_MACINFO_end_file`	The end of the current source file inclusion.
`DW_MACINFO_vendor_ext`	Vendor specific macro information directives that do not fit into one of the standard categories.

6.3.1.1 Define and Undefine Entries

All `DW_MACINFO_define` and `DW_MACINFO_undef` entries have two operands. The first operand encodes the line number of the source line on which the relevant defining or undefining pre-processor directives appeared.

The second operand consists of a null-terminated character string. In the case of a `DW_MACINFO_undef` entry, the value of this string will be simply the name of the pre-processor symbol which was undefined at the indicated source line.

In the case of a `DW_MACINFO_define` entry, the value of this string will be the name of the pre-processor symbol that was defined at the indicated source line, followed immediately by the macro formal parameter list including the surrounding parentheses (in the case of a function-like macro) followed by the definition string for the macro. If there is no formal parameter list, then the name of the defined macro is followed directly by its definition string.

In the case of a function-like macro definition, no whitespace characters should appear between the name of the defined macro and the following left parenthesis. Also, no whitespace characters should appear between successive formal parameters in the formal parameter list. (Successive formal parameters should, however, be separated by commas.) Also, exactly one space character should separate the right parenthesis which terminates the formal parameter list and the following definition string.

In the case of a ‘‘normal’’ (i.e. non-function-like) macro definition, exactly one space character should separate the name of the defined macro from the following definition text.

6.3.1.2 Start File Entries

Each `DW_MACINFO_start_file` entry also has two operands. The first operand encodes the line number of the source line on which the inclusion pre-processor directive occurred.

The second operand encodes a source file name index. This index corresponds to a file number in the statement information table for the relevant compilation unit. This index indicates (indirectly) the name of the file which is being included by the inclusion directive on the indicated source line.

6.3.1.3 End File Entries

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

6.3.1.4 Vendor Extension Entries

A `DW_MACINFO_vendor_ext` entry has two operands. The first is a constant. The second is a null-terminated character string. The meaning and/or significance of these operands is intentionally left undefined by this specification.

A consumer must be able to totally ignore all `DW_MACINFO_vendor_ext` entries that it does not understand.

6.3.2 Base Source Entries

In addition to producing a matched pair of `DW_MACINFO_start_file` and `DW_MACINFO_end_file` entries for each inclusion directive actually processed during compilation, a producer should generate such a matched pair also for the ‘‘base’’ source file submitted to the compiler for compilation. If the base source file for a compilation is submitted to the compiler via some means other than via a named disk file (e.g. via the standard input *stream* on a UNIX system) then the compiler should still produce this matched pair of `DW_MACINFO_start_file` and `DW_MACINFO_end_file` entries for the base source file, however, the file name indicated (indirectly) by the `DW_MACINFO_start_file` entry of the pair should reference a statement information file name entry consisting of a null string.

6.3.3 Macinfo Entries for Command Line Options

In addition to producing `DW_MACINFO_define` and `DW_MACINFO_undef` entries for each of the define and undefine directives processed during compilation, the DWARF producer should generate a `DW_MACINFO_define` or `DW_MACINFO_undef` entry for each pre-processor symbol which is defined or undefined by some means other than via a define or undefine directive within the compiled source text. In particular, pre-processor symbol definitions and un-definitions which occur as a result of command line options (when invoking the compiler) should be represented by their own `DW_MACINFO_define` and `DW_MACINFO_undef` entries.

All such `DW_MACINFO_define` and `DW_MACINFO_undef` entries representing compilation options should appear before the first `DW_MACINFO_start_file` entry for that compilation unit and should encode the value 0 in their line number operands.

6.3.4 General Rules and Restrictions

All macinfo entries within a `.debug_macinfo` section for a given compilation unit should appear in the same order in which the directives were processed by the compiler.

All macinfo entries representing command line options should appear in the same order as the relevant command line options were given to the compiler. In the case where the compiler itself implicitly supplies one or more macro definitions or un-definitions in addition to those which may be specified on the command line, macinfo entries should also be produced for these implicit definitions and un-definitions, and these entries should also appear in the proper order relative to each other and to any definitions or undefinitions given explicitly by the user on the command line.

6.4 Call Frame Information

Debuggers often need to be able to view and modify the state of any subroutine activation that is on the call stack. An activation consists of:

- *A code location that is within the subroutine. This location is either the place where the program stopped when the debugger got control (e.g. a breakpoint), or is a place where a subroutine made a call or was interrupted by an asynchronous event (e.g. a signal).*
- *An area of memory that is allocated on a stack called a ‘‘call frame.’’ The call frame is identified by an address on the stack. We refer to this address as the Canonical Frame Address or CFA.*
- *A set of registers that are in use by the subroutine at the code location.*

Typically, a set of registers are designated to be preserved across a call. If a callee wishes to use such a register, it saves the value that the register had at entry time in its call frame and restores it on exit. The code that allocates space on the call frame stack and performs the save operation is called the subroutine’s prologue, and the code that performs the restore operation and deallocates the frame is called its epilogue. Typically, the prologue code is physically at the beginning of a subroutine and the epilogue code is at the end.

To be able to view or modify an activation that is not on the top of the call frame stack, the debugger must ‘‘virtually unwind’’ the stack of activations until it finds the activation of interest. A debugger unwinds a stack in steps. Starting with the current activation it restores any registers that were preserved by the current activation and computes the predecessor’s CFA and code location. This has the logical effect of returning from the current subroutine to its predecessor. We say that the debugger virtually unwinds the stack because it preserves enough information to be able to ‘‘rewind’’ the stack back to the state it was in before it attempted to unwind it.

The unwinding operation needs to know where registers are saved and how to compute the predecessor’s CFA and code location. When considering an architecture-independent way of *encoding this information one has to consider a number of special things.*

- *Prologue and epilogue code is not always in distinct blocks at the beginning and end of a subroutine. It is common to duplicate the epilogue code at the site of each return from the code. Sometimes a compiler breaks up the register save/unsave operations and moves them into the body of the subroutine to just where they are needed.*
- *Compilers use different ways to manage the call frame. Sometimes they use a frame pointer register, sometimes not.*
- *The algorithm to compute the CFA changes as you progress through the prologue and epilogue code. (By definition, the CFA value does not change.)*
- *Some subroutines have no call frame.*
- *Sometimes a register is saved in another register that by convention does not need to be saved.*
- *Some architectures have special instructions that perform some or all of the register management in one instruction, leaving special information on the stack that indicates how registers are saved.*
- *Some architectures treat return address values specially. For example, in one architecture, the call instruction guarantees that the low order two bits will be zero and the return instruction ignores those bits. This leaves two bits of storage that are available to other uses that must be treated specially.*

6.4.1 Structure of Call Frame Information

DWARF supports virtual unwinding by defining an architecture independent basis for recording how procedures save and restore registers throughout their lifetimes. This basis must be augmented on some machines with specific information that is defined by either an architecture specific ABI authoring committee, a hardware vendor, or a compiler producer. The body defining a specific augmentation is referred to below as the ‘‘augmenter.’’

Abstractly, this mechanism describes a very large table that has the following structure:

```
LOC  CFA   R0   R1   ...  RN
L0
L1
...
LN
```

The first column indicates an address for every location that contains code in a program. (In shared objects, this is an object-relative offset.) The remaining columns contain virtual unwinding rules that are associated with the indicated location. The first column of the rules defines the CFA rule which is a register and a signed offset that are added together to compute the CFA value.

The remaining columns are labeled by register number. This includes some registers that have special designation on some architectures such as the PC and the stack pointer register. (The actual mapping of registers for a particular architecture is performed by the augmenter.) The register columns contain rules that describe whether a given register has been saved and the rule to find the value for the register in the previous frame.

The register rules are:

undefined	A register that has this rule has no value in the previous frame. (By convention, it is not preserved by a callee.)
same value	This register has not been modified from the previous frame. (By convention, it is preserved by the callee, but the callee has not modified it.)
offset(N)	The previous value of this register is saved at the address CFA+N where CFA is the current CFA value and N is a signed offset.
register(R)	The previous value of this register is stored in another register numbered R.
architectural	The rule is defined externally to this specification by the augmenter.

This table would be extremely large if actually constructed as described. Most of the entries at any point in the table are identical to the ones above them. The whole table can be represented quite compactly by recording just the differences starting at the beginning address of each subroutine in the program.

The virtual unwind information is encoded in a self-contained section called `.debug_frame`. Entries in a `.debug_frame` section are aligned on an addressing unit boundary and come in two forms: A Common Information Entry (CIE) and a Frame Description Entry (FDE). Sizes of data objects used in the encoding of the `.debug_frame` section are described in terms of the same data definitions used for the line number information (see section 6.2.1).

A Common Information Entry holds information that is shared among many Frame Descriptors. There is at least one CIE in every non-empty `.debug_frame` section. A CIE contains the following fields, in order:

1. `length`
   A uword constant that gives the number of bytes of the CIE structure, not including the length field, itself (length mod <addressing unit size> == 0).
2. `CIE_id`
   A uword constant that is used to distinguish CIEs from FDEs.
3. `version`
   A ubyte version number. This number is specific to the call frame information and is independent of the DWARF version number.
4. `augmentation`
   A null terminated string that identifies the augmentation to this CIE or to the FDEs that use it. If a reader encounters an augmentation string that is unexpected, then only the following fields can be read: CIE: `length`, `CIE_id`, `version`, `augmentation`; FDE: `length`, `CIE_pointer`, `initial_location`, `address_range`. If there is no augmentation, this value is a zero byte.
5. `code_alignment_factor`
   An unsigned LEB128 constant that is factored out of all advance location instructions (see below).
6. `data_alignment_factor`
   A signed LEB128 constant that is factored out of all offset instructions (see below.)

7. `return_address_register`
   A ubyte constant that indicates which column in the rule table represents the return address of the function. Note that this column might not correspond to an actual machine register.

8. `initial_instructions`
   A sequence of rules that are interpreted to create the initial setting of each column in the table.

9. `padding`
   Enough `DW_CFA_nop` instructions to make the size of this entry match the `length` value above.

An FDE contains the following fields, in order:

1. `length`
   A uword constant that gives the number of bytes of the header and instruction stream for this function (not including the length field itself) (length mod <addressing unit size> == 0).

2. `CIE_pointer`
   A uword constant offset into the `.debug_frame` section that denotes the CIE that is associated with this FDE.

3. `initial_location` An addressing-unit sized constant indicating the address of the first location associated with this table entry.

4. `address_range`
   An addressing unit sized constant indicating the number of bytes of program instructions described by this entry.

5. `instructions`
   A sequence of table defining instructions that are described below.

6.4.2 Call Frame Instructions

Each call frame instruction is defined to take 0 or more operands. Some of the operands may be encoded as part of the opcode (see section 7.23). The instructions are as follows:

1. `DW_CFA_advance_loc` takes a single argument that represents a constant delta. The required action is to create a new table row with a location value that is computed by taking the current entry's location value and adding (delta * `code_alignment_factor`). All other values in the new row are initially identical to the current row.

2. `DW_CFA_offset` takes two arguments: an unsigned LEB128 constant representing a factored offset and a register number. The required action is to change the rule for the register indicated by the register number to be an offset(N) rule with a value of (N = factored offset * `data_alignment_factor`).

3. `DW_CFA_restore` takes a single argument that represents a register number. The required action is to change the rule for the indicated register to the rule assigned it by the `initial_instructions` in the CIE.

4. `DW_CFA_set_loc` takes a single argument that represents an address. The required action is to create a new table row using the specified address as the location. All other values in the new row are initially identical to the current row. The new location value should always be greater than the current one.

5. `DW_CFA_advance_loc1` takes a single ubyte argument that represents a constant delta. This instruction is identical to `DW_CFA_advance_loc` except for the encoding and size of the delta argument.

6. `DW_CFA_advance_loc2` takes a single uhalf argument that represents a constant delta. This instruction is identical to `DW_CFA_advance_loc` except for the encoding and size of the delta argument.

7. `DW_CFA_advance_loc4` takes a single uword argument that represents a constant delta. This instruction is identical to `DW_CFA_advance_loc` except for the encoding and size of the delta argument.

8. `DW_CFA_offset_extended` takes two unsigned LEB128 arguments representing a register number and a factored offset. This instruction is identical to `DW_CFA_offset` except for the encoding and size of the register argument.

9. `DW_CFA_restore_extended` takes a single unsigned LEB128 argument that represents a register number. This instruction is identical to `DW_CFA_restore` except for the encoding and size of the register argument.

10. `DW_CFA_undefined` takes a single unsigned LEB128 argument that represents a register number. The required action is to set the rule for the specified register to ‘‘undefined.’’

11. `DW_CFA_same_value` takes a single unsigned LEB128 argument that represents a register number. The required action is to set the rule for the specified register to ‘‘same value.’’

12. `DW_CFA_register` takes two unsigned LEB128 arguments representing register numbers. The required action is to set the rule for the first register to be the second register.

13. `DW_CFA_remember_state`

14. `DW_CFA_restore_state`
    These instructions define a stack of information. Encountering the `DW_CFA_remember_state` instruction means to save the rules for every register on the current row on the stack. Encountering the `DW_CFA_restore_state` instruction means to pop the set of rules off the stack and place them in the current row. *(This operation is useful for compilers that move epilogue code into the body of a function.)*

15. `DW_CFA_def_cfa` takes two unsigned LEB128 arguments representing a register number and an offset. The required action is to define the current CFA rule to use the provided register and offset.

16. `DW_CFA_def_cfa_register` takes a single unsigned LEB128 argument representing a register number. The required action is to define the current CFA rule to use the provided register (but to keep the old offset).

17. `DW_CFA_def_cfa_offset` takes a single unsigned LEB128 argument representing an offset. The required action is to define the current CFA rule to use the provided offset (but to keep the old register).

18. `DW_CFA_nop` has no arguments and no required actions. It is used as padding to make the FDE an appropriate size.

6.4.3 Call Frame Instruction Usage

To determine the virtual unwind rule set for a given location (L1), one searches through the FDE headers looking at the `initial_location` *and* `address_range` *values to see if L1 is contained in the FDE. If so, then:*

1. *Initialize a register set by reading the* `initial_instructions` *field of the associated CIE.*
2. *Read and process the FDE's instruction sequence until a* `DW_CFA_advance_loc`*,* `DW_CFA_set_loc`*, or the end of the instruction stream is encountered.*
3. *If a* `DW_CFA_advance_loc` *or* `DW_CFA_set_loc` *instruction was encountered, then compute a new location value (L2). If L1 >= L2 then process the instruction and go back to step 2.*
4. *The end of the instruction stream can be thought of as a*
   `DW_CFA_set_loc( initial_location + address_range )`
   instruction. Unless the FDE is ill-formed, L1 should be less than L2 at this point.

The rules in the register set now apply to location L1.

For an example, see Appendix 5.

7. DATA REPRESENTATION

This section describes the binary representation of the debugging information entry itself, of the attribute types and of other fundamental elements described above.

7.1 Vendor Extensibility

To reserve a portion of the DWARF name space and ranges of enumeration values for use for vendor specific extensions, special labels are reserved for tag names, attribute names, base type encodings, location operations, language names, calling conventions and call frame instructions. The labels denoting the beginning and end of the reserved value range for vendor specific extensions consist of the appropriate prefix (`DW_TAG`, `DW_AT`, `DW_ATE`, `DW_OP`, `DW_LANG`, or `DW_CFA` respectively) followed by `_lo_user` or `_hi_user`. For example, for entry tags, the special labels are `DW_TAG_lo_user` and `DW_TAG_hi_user`. Values in the range between *prefix*`_lo_user` and *prefix*`_hi_user` inclusive, are reserved for vendor specific extensions. Vendors may use values in this range without conflicting with current or future system-defined values. All other values are reserved for use by the system.

Vendor defined tags, attributes, base type encodings, location atoms, language names, calling conventions and call frame instructions, conventionally use the form *prefix_vendor_id_name*, where *vendor_id* is some identifying character sequence chosen so as to avoid conflicts with other vendors.

To ensure that extensions added by one vendor may be safely ignored by consumers that do not understand those extensions, the following rules should be followed:

1. New attributes should be added in such a way that a debugger may recognize the format of a new attribute value without knowing the content of that attribute value.

2. The semantics of any new attributes should not alter the semantics of previously existing attributes.

3. The semantics of any new tags should not conflict with the semantics of previously existing tags.

7.2 Reserved Error Values

As a convenience for consumers of DWARF information, the value 0 is reserved in the encodings for attribute names, attribute forms, base type encodings, location operations, languages, statement program opcodes, macro information entries and tag names to represent an error condition or unknown value. DWARF does not specify names for these reserved values, since they do not represent valid encodings for the given type and should not appear in DWARF debugging information.

7.3 Executable Objects and Shared Objects

The relocated addresses in the debugging information for an executable object are virtual addresses and the relocated addresses in the debugging information for a shared object are offsets relative to the start of the lowest segment used by that shared object.

This requirement makes the debugging information for shared objects position independent. Virtual addresses in a shared object may be calculated by adding the offset to the base address at which the object was attached. This offset is available in the run-time linker's data structures.

7.4 File Constraints

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

7.5 Format of Debugging Information

For each compilation unit compiled with a DWARF Version 2 producer, a contribution is made to the `.debug_info` section of the object file. Each such contribution consists of a compilation unit header followed by a series of debugging information entries. Unlike the information encoding for DWARF Version 1, Version 2 debugging information entries do not themselves contain the debugging information entry tag or the attribute name and form encodings for each attribute. Instead, each debugging information entry begins with a code that represents an entry in a separate abbreviations table. This code is followed directly by a series of attribute values. The appropriate entry in the abbreviations table guides the interpretation of the information contained directly in the `.debug_info` section. Each compilation unit is associated with a particular abbreviation table, but multiple compilation units may share the same table.

This encoding was based on the observation that typical DWARF producers produce a very limited number of different types of debugging information entries. By extracting the common information from those entries into a separate table, we are able to compress the generated information.

7.5.1 Compilation Unit Header

The header for the series of debugging information entries contributed by a single compilation unit consists of the following information:

1. A 4-byte unsigned integer representing the length of the `.debug_info` contribution for that compilation unit, not including the length field itself.
2. A 2-byte unsigned integer representing the version of the DWARF information for that compilation unit. For DWARF Version 2, the value in this field is 2.
3. A 4-byte unsigned offset into the `.debug_abbrev` section. This offset associates the compilation unit with a particular set of debugging information entry abbreviations.
4. A 1-byte unsigned integer representing the size in bytes of an address on the target architecture. If the system uses segmented addressing, this value represents the size of the offset portion of an address.

The compilation unit header does not replace the `DW_TAG_compile_unit` *debugging information entry. It is additional information that is represented outside the standard DWARF tag/attributes format.*

7.5.2 Debugging Information Entry

Each debugging information entry begins with an unsigned LEB128 number containing the abbreviation code for the entry. This code represents an entry within the abbreviation table associated with the compilation unit containing this entry. The abbreviation code is followed by a series of attribute values.

On some architectures, there are alignment constraints on section boundaries. To make it easier to pad debugging information sections to satisfy such constraints, the abbreviation code 0 is reserved. Debugging information entries consisting of only the 0 abbreviation code are considered null entries.

7.5.3 Abbreviation Tables

The abbreviation tables for all compilation units are contained in a separate object file section called `.debug_abbrev`. As mentioned before, multiple compilation units may share the same abbreviation table.

The abbreviation table for a single compilation unit consists of a series of abbreviation declarations. Each declaration specifies the tag and attributes for a particular form of debugging information entry. Each declaration begins with an unsigned LEB128 number representing the abbreviation code itself. It is this code that appears at the beginning of a debugging information entry in the `.debug_info` section. As described above, the abbreviation code 0 is reserved for null debugging information entries. The abbreviation code is followed by another unsigned LEB128 number that encodes the entry's tag. The encodings for the tag names are given in Figures 14 and 15.

Following the tag encoding is a 1-byte value that determines whether a debugging information entry using this abbreviation has child entries or not. If the value is `DW_CHILDREN_yes`, the next physically succeeding entry of any debugging information entry using this abbreviation is the first child of the prior entry. If the 1-byte value following the abbreviation's tag encoding is `DW_CHILDREN_no`, the next physically succeeding entry of any debugging information entry using this abbreviation is a sibling of the prior entry. (Either the first child or sibling entries may be null entries). The encodings for the child determination byte are given in Figure 16. (As mentioned in section 2.3, each chain of sibling entries is terminated by a null entry).

Finally, the child encoding is followed by a series of attribute specifications. Each attribute specification consists of two parts. The first part is an unsigned LEB128 number representing the attribute's name. The second part is an unsigned LEB128 number representing the attribute's form. The series of attribute specifications ends with an entry containing 0 for the name and 0 for the form.

The attribute form `DW_FORM_indirect` is a special case. For attributes with this form, the attribute value itself in the `.debug_info` section begins with an unsigned LEB128 number that represents its form. This allows producers to choose forms for particular attributes dynamically, without having to add a new entry to the abbreviation table.

The abbreviations for a given compilation unit end with an entry consisting of a 0 byte for the abbreviation code.

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

7.5.4 Attribute Encodings

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

The attribute form governs how the value of the attribute is encoded. The possible forms may belong to one of the following form classes:

address — Represented as an object of appropriate size to hold an address on the target machine (`DW_FORM_addr`). This address is relocatable in a relocatable object file and is relocated in an executable file or shared object.

block — Blocks come in four forms. The first consists of a 1-byte length followed by 0 to 255 contiguous information bytes (`DW_FORM_block1`). The second consists of a 2-byte length followed by 0 to 65,535 contiguous

Tag name	Value
DW_TAG_array_type	0x01
DW_TAG_class_type	0x02
DW_TAG_entry_point	0x03
DW_TAG_enumeration_type	0x04
DW_TAG_formal_parameter	0x05
DW_TAG_imported_declaration	0x08
DW_TAG_label	0x0a
DW_TAG_lexical_block	0x0b
DW_TAG_member	0x0d
DW_TAG_pointer_type	0x0f
DW_TAG_reference_type	0x10
DW_TAG_compile_unit	0x11
DW_TAG_string_type	0x12
DW_TAG_structure_type	0x13
DW_TAG_subroutine_type	0x15
DW_TAG_typedef	0x16
DW_TAG_union_type	0x17
DW_TAG_unspecified_parameters	0x18
DW_TAG_variant	0x19
DW_TAG_common_block	0x1a
DW_TAG_common_inclusion	0x1b
DW_TAG_inheritance	0x1c
DW_TAG_inlined_subroutine	0x1d
DW_TAG_module	0x1e
DW_TAG_ptr_to_member_type	0x1f
DW_TAG_set_type	0x20
DW_TAG_subrange_type	0x21
DW_TAG_with_stmt	0x22
DW_TAG_access_declaration	0x23
DW_TAG_base_type	0x24
DW_TAG_catch_block	0x25
DW_TAG_const_type	0x26
DW_TAG_constant	0x27
DW_TAG_enumerator	0x28
DW_TAG_file_type	0x29

Figure 14. Tag encodings (part 1)

information bytes (DW_FORM_block2). The third consists of a 4-byte length followed by 0 to 4,294,967,295 contiguous information bytes (DW_FORM_block4). The fourth consists of an unsigned LEB128 length followed by the number of bytes specified by the length (DW_FORM_block). In all forms, the length is the number of information bytes that follow. The information bytes may contain any mixture of relocated (or relocatable) addresses, references to other debugging information entries or data bytes.

constant — There are six forms of constants: one, two, four and eight byte values (respectively, DW_FORM_data1, DW_FORM_data2,

Tag name	Value
DW_TAG_friend	0x2a
DW_TAG_namelist	0x2b
DW_TAG_namelist_item	0x2c
DW_TAG_packed_type	0x2d
DW_TAG_subprogram	0x2e
DW_TAG_template_type_param	0x2f
DW_TAG_template_value_param	0x30
DW_TAG_thrown_type	0x31
DW_TAG_try_block	0x32
DW_TAG_variant_part	0x33
DW_TAG_variable	0x34
DW_TAG_volatile_type	0x35
DW_TAG_lo_user	0x4080
DW_TAG_hi_user	0xffff

Figure 15. Tag encodings (part 2)

Child determination name	Value
DW_CHILDREN_no	0
DW_CHILDREN_yes	1

Figure 16. Child determination encodings

DW_FORM_data4, and DW_FORM_data8). There are also variable length constant data forms encoded using LEB128 numbers (see below). Both signed (DW_FORM_sdata) and unsigned (DW_FORM_udata) variable length constants are available.

flag — A flag is represented as a single byte of data (DW_FORM_flag). If the flag has value zero, it indicates the absence of the attribute. If the flag has a non-zero value, it indicates the presence of the attribute.

reference — There are two types of reference. The first is an offset relative to the first byte of the compilation unit header for the compilation unit containing the reference. The offset must refer to an entry within that same compilation unit. There are five forms for this type of reference: one, two, four and eight byte offsets (respectively, DW_FORM_ref1, DW_FORM_ref2, DW_FORM_ref4, and DW_FORM_ref8). There are is also an unsigned variable length offset encoded using LEB128 numbers (DW_FORM_ref_udata).

The second type of reference is the address of any debugging information entry within the same executable or shared object; it may refer to an entry in a different compilation unit from the unit containing the reference. This type of reference (DW_FORM_ref_addr) is the size of an address on the target architecture; it is relocatable in a relocatable object file and relocated in an executable file or shared object.

The use of compilation unit relative references will reduce the number of link-time relocations and so speed up linking.

Attribute name	Value	Classes
DW_AT_sibling	0x01	reference
DW_AT_location	0x02	block, constant
DW_AT_name	0x03	string
DW_AT_ordering	0x09	constant
DW_AT_byte_size	0x0b	constant
DW_AT_bit_offset	0x0c	constant
DW_AT_bit_size	0x0d	constant
DW_AT_stmt_list	0x10	constant
DW_AT_low_pc	0x11	address
DW_AT_high_pc	0x12	address
DW_AT_language	0x13	constant
DW_AT_discr	0x15	reference
DW_AT_discr_value	0x16	constant
DW_AT_visibility	0x17	constant
DW_AT_import	0x18	reference
DW_AT_string_length	0x19	block, constant
DW_AT_common_reference	0x1a	reference
DW_AT_comp_dir	0x1b	string
DW_AT_const_value	0x1c	string, constant, block
DW_AT_containing_type	0x1d	reference
DW_AT_default_value	0x1e	reference
DW_AT_inline	0x20	constant
DW_AT_is_optional	0x21	flag
DW_AT_lower_bound	0x22	constant, reference
DW_AT_producer	0x25	string
DW_AT_prototyped	0x27	flag
DW_AT_return_addr	0x2a	block, constant
DW_AT_start_scope	0x2c	constant
DW_AT_stride_size	0x2e	constant
DW_AT_upper_bound	0x2f	constant, reference

Figure 17. Attribute encodings, part 1

The use of address-type references allows for the commonization of information, such as types, across compilation units.

string — A string is a sequence of contiguous non-null bytes followed by one null byte. A string may be represented immediately in the debugging information entry itself (`DW_FORM_string`), or may be represented as a 4-byte offset into a string table contained in the `.debug_str` section of the object file (`DW_FORM_strp`).

The form encodings are listed in Figure 19.

7.6 Variable Length Data

The special constant data forms `DW_FORM_sdata` and `DW_FORM_udata` are encoded using ‘‘Little Endian Base 128’’ (LEB128) numbers. LEB128 is a scheme for encoding integers densely that exploits the assumption that most integers are small in magnitude. (This encoding is equally suitable whether the target machine architecture represents data in big-endian or little-endian order. It is ‘‘little endian’’ only in the sense that it avoids using space to represent the

Attribute name	Value	Classes
DW_AT_abstract_origin	0x31	reference
DW_AT_accessibility	0x32	constant
DW_AT_address_class	0x33	constant
DW_AT_artificial	0x34	flag
DW_AT_base_types	0x35	reference
DW_AT_calling_convention	0x36	constant
DW_AT_count	0x37	constant, reference
DW_AT_data_member_location	0x38	block, reference
DW_AT_decl_column	0x39	constant
DW_AT_decl_file	0x3a	constant
DW_AT_decl_line	0x3b	constant
DW_AT_declaration	0x3c	flag
DW_AT_discr_list	0x3d	block
DW_AT_encoding	0x3e	constant
DW_AT_external	0x3f	flag
DW_AT_frame_base	0x40	block, constant
DW_AT_friend	0x41	reference
DW_AT_identifier_case	0x42	constant
DW_AT_macro_info	0x43	constant
DW_AT_namelist_item	0x44	block
DW_AT_priority	0x45	reference
DW_AT_segment	0x46	block, constant
DW_AT_specification	0x47	reference
DW_AT_static_link	0x48	block, constant
DW_AT_type	0x49	reference
DW_AT_use_location	0x4a	block, constant
DW_AT_variable_parameter	0x4b	flag
DW_AT_virtuality	0x4c	constant
DW_AT_vtable_elem_location	0x4d	block, reference
DW_AT_lo_user	0x2000	—
DW_AT_hi_user	0x3fff	—

Figure 18. Attribute encodings, part 2

‘‘big’’ end of an unsigned integer, when the big end is all zeroes or sign extension bits).

DW_FORM_udata (unsigned LEB128) numbers are encoded as follows: start at the low order end of an unsigned integer and chop it into 7-bit chunks. Place each chunk into the low order 7 bits of a byte. Typically, several of the high order bytes will be zero; discard them. Emit the remaining bytes in a stream, starting with the low order byte; set the high order bit on each byte except the last emitted byte. The high bit of zero on the last byte indicates to the decoder that it has encountered the last byte.

The integer zero is a special case, consisting of a single zero byte.

Figure 20 gives some examples of DW_FORM_udata *numbers. The* 0x80 *in each case is the high order bit of the byte, indicating that an additional byte follows:*

The encoding for DW_FORM_sdata (signed, 2s complement LEB128) numbers is similar, except that the criterion for discarding high order bytes is not whether they are zero, but whether they consist entirely of sign extension bits. Consider the 32-bit integer -2. The three high level

Form name	Value	Class
DW_FORM_addr	0x01	address
DW_FORM_block2	0x03	block
DW_FORM_block4	0x04	block
DW_FORM_data2	0x05	constant
DW_FORM_data4	0x06	constant
DW_FORM_data8	0x07	constant
DW_FORM_string	0x08	string
DW_FORM_block	0x09	block
DW_FORM_block1	0x0a	block
DW_FORM_data1	0x0b	constant
DW_FORM_flag	0x0c	flag
DW_FORM_sdata	0x0d	constant
DW_FORM_strp	0x0e	string
DW_FORM_udata	0x0f	constant
DW_FORM_ref_addr	0x10	reference
DW_FORM_ref1	0x11	reference
DW_FORM_ref2	0x12	reference
DW_FORM_ref4	0x13	reference
DW_FORM_ref8	0x14	reference
DW_FORM_ref_udata	0x15	reference
DW_FORM_indirect	0x16	(see section 7.5.3)

Figure 19. Attribute form encodings

Number	First byte	Second byte
2	2	—
127	127	—
128	0+0x80	1
129	1+0x80	1
130	2+0x80	1
12857	57+0x80	100

Figure 20. Examples of unsigned LEB128 encodings

bytes of the number are sign extension, thus LEB128 would represent it as a single byte containing the low order 7 bits, with the high order bit cleared to indicate the end of the byte stream. Note that there is nothing within the LEB128 representation that indicates whether an encoded number is signed or unsigned. The decoder must know what type of number to expect.

Figure 21 gives some examples of `DW_FORM_sdata` *numbers.*

Appendix 4 gives algorithms for encoding and decoding these forms.

7.7 Location Descriptions

7.7.1 Location Expressions

A location expression is stored in a block of contiguous bytes. The bytes form a set of operations. Each location operation has a 1-byte code that identifies that operation. Operations can be followed by one or more bytes of additional data. All operations in a location expression are concatenated from left to right. The encodings for the operations in a location expression are described in Figures 22 and 23.

Number	First byte	Second byte
2	2	—
-2	0x7e	—
127	127+0x80	0
-127	1+0x80	0x7f
128	0+0x80	1
-128	0+0x80	0x7f
129	1+0x80	1
-129	0x7f+0x80	0x7e

Figure 21. Examples of signed LEB128 encodings

Operation	Code	No. of Operands	Notes
DW_OP_addr	0x03	1	constant address (size target specific)
DW_OP_deref	0x06	0	
DW_OP_const1u	0x08	1	1-byte constant
DW_OP_const1s	0x09	1	1-byte constant
DW_OP_const2u	0x0a	1	2-byte constant
DW_OP_const2s	0x0b	1	2-byte constant
DW_OP_const4u	0x0c	1	4-byte constant
DW_OP_const4s	0x0d	1	4-byte constant
DW_OP_const8u	0x0e	1	8-byte constant
DW_OP_const8s	0x0f	1	8-byte constant
DW_OP_constu	0x10	1	ULEB128 constant
DW_OP_consts	0x11	1	SLEB128 constant
DW_OP_dup	0x12	0	
DW_OP_drop	0x13	0	
DW_OP_over	0x14	0	
DW_OP_pick	0x15	1	1-byte stack index
DW_OP_swap	0x16	0	
DW_OP_rot	0x17	0	
DW_OP_xderef	0x18	0	
DW_OP_abs	0x19	0	
DW_OP_and	0x1a	0	
DW_OP_div	0x1b	0	
DW_OP_minus	0x1c	0	
DW_OP_mod	0x1d	0	
DW_OP_mul	0x1e	0	
DW_OP_neg	0x1f	0	
DW_OP_not	0x20	0	
DW_OP_or	0x21	0	
DW_OP_plus	0x22	0	
DW_OP_plus_uconst	0x23	1	ULEB128 addend
DW_OP_shl	0x24	0	
DW_OP_shr	0x25	0	
DW_OP_shra	0x26	0	

Figure 22. Location operation encodings, part 1

Operation	Code	No. of Operands	Notes
DW_OP_xor	0x27	0	
DW_OP_skip	0x2f	1	signed 2-byte constant
DW_OP_bra	0x28	1	signed 2-byte constant
DW_OP_eq	0x29	0	
DW_OP_ge	0x2a	0	
DW_OP_gt	0x2b	0	
DW_OP_le	0x2c	0	
DW_OP_lt	0x2d	0	
DW_OP_ne	0x2e	0	
DW_OP_lit0	0x30	0	literals 0..31 = (DW_OP_LIT0|literal)
DW_OP_lit1	0x31	0	
...			
DW_OP_lit31	0x4f	0	
DW_OP_reg0	0x50	0	reg 0..31 = (DW_OP_REG0|regnum)
DW_OP_reg1	0x51	0	
...			
DW_OP_reg31	0x6f	0	
DW_OP_breg0	0x70	1	SLEB128 offset
DW_OP_breg1	0x71	1	base reg 0..31 = (DW_OP_BREG0|regnum)
...			
DW_OP_breg31	0x8f	1	
DW_OP_regx	0x90	1	ULEB128 register
DW_OP_fbreg	0x91	1	SLEB128 offset
DW_OP_bregx	0x92	2	ULEB128 register followed by SLEB128 offset
DW_OP_piece	0x93	1	ULEB128 size of piece addressed
DW_OP_deref_size	0x94	1	1-byte size of data retrieved
DW_OP_xderef_size	0x95	1	1-byte size of data retrieved
DW_OP_nop	0x96	0	
DW_OP_lo_user	0xe0		
DW_OP_hi_user	0xff		

Figure 23. Location operation encodings, part 2

7.7.2 Location Lists

Each entry in a location list consists of two relative addresses followed by a 2-byte length, followed by a block of contiguous bytes. The length specifies the number of bytes in the block that follows. The two addresses are the same size as used by `DW_FORM_addr` on the target machine.

7.8 Base Type Encodings

The values of the constants used in the `DW_AT_encoding` attribute are given in Figure 24.

7.9 Accessibility Codes

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

Base type encoding name	Value
DW_ATE_address	0x1
DW_ATE_boolean	0x2
DW_ATE_complex_float	0x3
DW_ATE_float	0x4
DW_ATE_signed	0x5
DW_ATE_signed_char	0x6
DW_ATE_unsigned	0x7
DW_ATE_unsigned_char	0x8
DW_ATE_lo_user	0x80
DW_ATE_hi_user	0xff

Figure 24. Base type encoding values

Accessibility code name	Value
DW_ACCESS_public	1
DW_ACCESS_protected	2
DW_ACCESS_private	3

Figure 25. Accessibility encodings

7.10 Visibility Codes

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

Visibility code name	Value
DW_VIS_local	1
DW_VIS_exported	2
DW_VIS_qualified	3

Figure 26. Visibility encodings

7.11 Virtuality Codes

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

Virtuality code name	Value
DW_VIRTUALITY_none	0
DW_VIRTUALITY_virtual	1
DW_VIRTUALITY_pure_virtual	2

Figure 27. Virtuality encodings

7.12 Source Languages

The encodings for source languages are given in Figure 28. Names marked with † and their associated values are reserved, but the languages they represent are not supported in DWARF Version 2.

7.13 Address Class Encodings

The value of the common address class encoding `DW_ADDR_none` is 0.

Language name	Value
DW_LANG_C89	0x0001
DW_LANG_C	0x0002
DW_LANG_Ada83†	0x0003
DW_LANG_C_plus_plus	0x0004
DW_LANG_Cobol74†	0x0005
DW_LANG_Cobol85†	0x0006
DW_LANG_Fortran77	0x0007
DW_LANG_Fortran90	0x0008
DW_LANG_Pascal83	0x0009
DW_LANG_Modula2	0x000a
DW_LANG_lo_user	0x8000
DW_LANG_hi_user	0xffff

Figure 28. Language encodings

7.14 Identifier Case

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

Identifier Case Name	Value
DW_ID_case_sensitive	0
DW_ID_up_case	1
DW_ID_down_case	2
DW_ID_case_insensitive	3

Figure 29. Identifier case encodings

7.15 Calling Convention Encodings

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

Calling Convention Name	Value
DW_CC_normal	0x1
DW_CC_program	0x2
DW_CC_nocall	0x3
DW_CC_lo_user	0x40
DW_CC_hi_user	0xff

Figure 30. Calling convention encodings

7.16 Inline Codes

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

Inline Code Name	Value
DW_INL_not_inlined	0
DW_INL_inlined	1
DW_INL_declared_not_inlined	2
DW_INL_declared_inlined	3

Figure 31. Inline encodings

7.17 Array Ordering

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

Ordering name	Value
DW_ORD_row_major	0
DW_ORD_col_major	1

Figure 32. Ordering encodings

7.18 Discriminant Lists

The descriptors used in the DW_AT_dicsr_list attribute are encoded as 1-byte constants. The defined values are presented in Figure 33.

Descriptor Name	Value
DW_DSC_label	0
DW_DSC_range	1

Figure 33. Discriminant descriptor encodings

7.19 Name Lookup Table

Each set of entries in the table of global names contained in the .debug_pubnames section begins with a header consisting of: a 4-byte length containing the length of the set of entries for this compilation unit, not including the length field itself; a 2-byte version identifier containing the value 2 for DWARF Version 2; a 4-byte offset into the .debug_info section; and a 4-byte length containing the size in bytes of the contents of the .debug_info section generated to represent this compilation unit. This header is followed by a series of tuples. Each tuple consists of a 4-byte offset followed by a string of non-null bytes terminated by one null byte. Each set is terminated by a 4-byte word containing the value 0.

7.20 Address Range Table

Each set of entries in the table of address ranges contained in the .debug_aranges section begins with a header consisting of: a 4-byte length containing the length of the set of entries for this compilation unit, not including the length field itself; a 2-byte version identifier containing the value 2 for DWARF Version 2; a 4-byte offset into the .debug_info section; a 1-byte unsigned integer containing the size in bytes of an address (or the offset portion of an address for segmented addressing) on the target system; and a 1-byte unsigned integer containing the size in bytes of a segment descriptor on the target system. This header is followed by a series of tuples. Each tuple consists of an address and a length, each in the size appropriate for an address on the target architecture. The first tuple following the header in each set begins at an offset that is a multiple of the size of a single tuple (that is, twice the size of an address). The header is padded, if necessary, to the appropriate boundary. Each set of tuples is terminated by a 0 for the address and 0 for the length.

7.21 Line Number Information

The sizes of the integers used in the line number and call frame information sections are as follows:

sbyte Signed 1-byte value.

ubyte Unsigned 1-byte value.

uhalf	Unsigned 2-byte value.
sword	Signed 4-byte value.
uword	Unsigned 4-byte value.

The version number in the statement program prologue is 2 for DWARF Version 2. The boolean values ‘‘true’’ and ‘‘false’’ used by the statement information program are encoded as a single byte containing the value 0 for ‘‘false,’’ and a non-zero value for ‘‘true.’’ The encodings for the pre-defined standard opcodes are given in Figure 34.

Opcode Name	Value
`DW_LNS_copy`	1
`DW_LNS_advance_pc`	2
`DW_LNS_advance_line`	3
`DW_LNS_set_file`	4
`DW_LNS_set_column`	5
`DW_LNS_negate_stmt`	6
`DW_LNS_set_basic_block`	7
`DW_LNS_const_add_pc`	8
`DW_LNS_fixed_advance_pc`	9

Figure 34. Standard Opcode Encodings

The encodings for the pre-defined extended opcodes are given in Figure 35.

Opcode Name	Value
`DW_LNE_end_sequence`	1
`DW_LNE_set_address`	2
`DW_LNE_define_file`	3

Figure 35. Extended Opcode Encodings

7.22 Macro Information

The source line numbers and source file indices encoded in the macro information section are represented as unsigned LEB128 numbers as are the constants in an `DW_MACINFO_vendor_ext` entry. The macinfo type is encoded as a single byte. The encodings are given in Figure 36.

Macinfo Type Name	Value
`DW_MACINFO_define`	1
`DW_MACINFO_undef`	2
`DW_MACINFO_start_file`	3
`DW_MACINFO_end_file`	4
`DW_MACINFO_vendor_ext`	255

Figure 36. Macinfo Type Encodings

7.23 Call Frame Information

The value of the CIE id in the CIE header is `0xffffffff`. The initial value of the CIE version number is 1.

Call frame instructions are encoded in one or more bytes. The primary opcode is encoded in the high order two bits of the first byte (that is, opcode = byte >> 6). An operand or extended opcode may be encoded in the low order 6 bits. Additional operands are encoded in subsequent bytes. The instructions and their encodings are presented in Figure 37.

Instruction	High 2 Bits	Low 6 Bits	Operand 1	Operand 2
DW_CFA_advance_loc	0x1	delta		
DW_CFA_offset	0x2	register	ULEB128 offset	
DW_CFA_restore	0x3	register		
DW_CFA_set_loc	0	0x01	address	
DW_CFA_advance_loc1	0	0x02	1-byte delta	
DW_CFA_advance_loc2	0	0x03	2-byte delta	
DW_CFA_advance_loc4	0	0x04	4-byte delta	
DW_CFA_offset_extended	0	0x05	ULEB128 register	ULEB128 offset
DW_CFA_restore_extended	0	0x06	ULEB128 register	
DW_CFA_undefined	0	0x07	ULEB128 register	
DW_CFA_same_value	0	0x08	ULEB128 register	
DW_CFA_register	0	0x09	ULEB128 register	ULEB128 register
DW_CFA_remember_state	0	0x0a		
DW_CFA_restore_state	0	0x0b		
DW_CFA_def_cfa	0	0x0c	ULEB128 register	ULEB128 offset
DW_CFA_def_cfa_register	0	0x0d	ULEB128 register	
DW_CFA_def_cfa_offset	0	0x0e	ULEB128 offset	
DW_CFA_nop	0	0		
DW_CFA_lo_user	0	0x1c		
DW_CFA_hi_user	0	0x3f		

Figure 37. Call frame instruction encodings

7.24 Dependencies

The debugging information in this format is intended to exist in the .debug_abbrev, .debug_aranges, .debug_frame, .debug_info, .debug_line, .debug_loc, .debug_macinfo, .debug_pubnames and .debug_str sections of an object file. The information is not word-aligned, so the assembler must provide a way for the compiler to produce 2-byte and 4-byte quantities without alignment restrictions, and the linker must be able to relocate a 4-byte reference at an arbitrary alignment. In target architectures with 64-bit addresses, the assembler and linker must similarly handle 8-byte references at arbitrary alignments.

8. FUTURE DIRECTIONS

The UNIX International Programming Languages SIG is working on a specification for a set of interfaces for reading DWARF information, that will hide changes in the representation of that information from its consumers. It is hoped that using these interfaces will make the transition from DWARF Version 1 to Version 2 much simpler and will make it easier for a single consumer to support objects using either Version 1 or Version 2 DWARF.

A draft of this specification is available for review from UNIX International. The Programming Languages SIG wishes to stress, however, that the specification is still in flux.

Appendix 1 -- Current Attributes by Tag Value

The list below enumerates the attributes that are most applicable to each type of debugging information entry. DWARF does not in general require that a given debugging information entry contain a particular attribute or set of attributes. Instead, a DWARF producer is free to generate any, all, or none of the attributes described in the text as being applicable to a given entry. Other attributes (both those defined within this document but not explicitly associated with the entry in question, and new, vendor-defined ones) may also appear in a given debugging entry. Therefore, the list may be taken as instructive, but cannot be considered definitive.

TAG NAME	APPLICABLE ATTRIBUTES
DW_TAG_access_declaration	DECL† DW_AT_accessibility DW_AT_name DW_AT_sibling
DW_TAG_array_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_byte_size DW_AT_declaration DW_AT_name DW_AT_ordering DW_AT_sibling DW_AT_start_scope DW_AT_stride_size DW_AT_type DW_AT_visibility
DW_TAG_base_type	DW_AT_bit_offset DW_AT_bit_size DW_AT_byte_size DW_AT_encoding DW_AT_name DW_AT_sibling
DW_TAG_catch_block	DW_AT_abstract_origin DW_AT_high_pc DW_AT_low_pc DW_AT_segment DW_AT_sibling

† DW_AT_decl_column, DW_AT_decl_file, DW_AT_decl_line.

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

TAG NAME	APPLICABLE ATTRIBUTES
DW_TAG_class_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_byte_size DW_AT_declaration DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_visibility
DW_TAG_common_block	DECL DW_AT_declaration DW_AT_location DW_AT_name DW_AT_sibling DW_AT_visibility
DW_TAG_common_inclusion	DECL DW_AT_common_reference DW_AT_declaration DW_AT_sibling DW_AT_visibility
DW_TAG_compile_unit	DW_AT_base_types DW_AT_comp_dir DW_AT_identifier_case DW_AT_high_pc DW_AT_language DW_AT_low_pc DW_AT_macro_info DW_AT_name DW_AT_producer DW_AT_sibling DW_AT_stmt_list
DW_TAG_const_type	DW_AT_sibling DW_AT_type

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

TAG NAME	APPLICABLE ATTRIBUTES
DW_TAG_constant	DECL DW_AT_accessibility DW_AT_constant_value DW_AT_declaration DW_AT_external DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_type DW_AT_visibility
DW_TAG_entry_point	DW_AT_address_class DW_AT_low_pc DW_AT_name DW_AT_return_addr DW_AT_segment DW_AT_sibling DW_AT_static_link DW_AT_type
DW_TAG_enumeration_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_byte_size DW_AT_declaration DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_visibility
DW_TAG_enumerator	DECL DW_AT_const_value DW_AT_name DW_AT_sibling
DW_TAG_file_type	DECL DW_AT_abstract_origin DW_AT_byte_size DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_type DW_AT_visibility

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

TAG NAME	APPLICABLE ATTRIBUTES
DW_TAG_formal_parameter	DECL DW_AT_abstract_origin DW_AT_artificial DW_AT_default_value DW_AT_is_optional DW_AT_location DW_AT_name DW_AT_segment DW_AT_sibling DW_AT_type DW_AT_variable_parameter
DW_TAG_friend	DECL DW_AT_abstract_origin DW_AT_friend DW_AT_sibling
DW_TAG_imported_declaration	DECL DW_AT_accessibility DW_AT_import DW_AT_name DW_AT_sibling DW_AT_start_scope
DW_TAG_inheritance	DECL DW_AT_accessibility DW_AT_data_member_location DW_AT_sibling DW_AT_type DW_AT_virtuality
DW_TAG_inlined_subroutine	DECL DW_AT_abstract_origin DW_AT_high_pc DW_AT_low_pc DW_AT_segment DW_AT_sibling DW_AT_return_addr DW_AT_start_scope
DW_TAG_label	DW_AT_abstract_origin DW_AT_low_pc DW_AT_name DW_AT_segment DW_AT_start_scope DW_AT_sibling

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

TAG NAME	APPLICABLE ATTRIBUTES
DW_TAG_lexical_block	DW_AT_abstract_origin DW_AT_high_pc DW_AT_low_pc DW_AT_name DW_AT_segment DW_AT_sibling
DW_TAG_member	DECL DW_AT_accessibility DW_AT_byte_size DW_AT_bit_offset DW_AT_bit_size DW_AT_data_member_location DW_AT_declaration DW_AT_name DW_AT_sibling DW_AT_type DW_AT_visibility
DW_TAG_module	DECL DW_AT_accessibility DW_AT_declaration DW_AT_high_pc DW_AT_low_pc DW_AT_name DW_AT_priority DW_AT_segment DW_AT_sibling DW_AT_visibility
DW_TAG_namelist	DECL DW_AT_accessibility DW_AT_abstract_origin DW_AT_declaration DW_AT_sibling DW_AT_visibility
DW_TAG_namelist_item	DECL DW_AT_namelist_item DW_AT_sibling
DW_TAG_packed_type	DW_AT_sibling DW_AT_type

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

TAG NAME	APPLICABLE ATTRIBUTES
DW_TAG_pointer_type	DW_AT_address_class DW_AT_sibling DW_AT_type
DW_TAG_ptr_to_member_type	DECL DW_AT_abstract_origin DW_AT_address_class DW_AT_containing_type DW_AT_declaration DW_AT_name DW_AT_sibling DW_AT_type DW_AT_use_location DW_AT_visibility
DW_TAG_reference_type	DW_AT_address_class DW_AT_sibling DW_AT_type
DW_TAG_set_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_byte_size DW_AT_declaration DW_AT_name DW_AT_start_scope DW_AT_sibling DW_AT_type DW_AT_visibility
DW_TAG_string_type	DECL DW_AT_accessibility DW_AT_abstract_origin DW_AT_byte_size DW_AT_declaration DW_AT_name DW_AT_segment DW_AT_sibling DW_AT_start_scope DW_AT_string_length DW_AT_visibility

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

TAG NAME	APPLICABLE ATTRIBUTES
DW_TAG_structure_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_byte_size DW_AT_declaration DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_visibility
DW_TAG_subprogram	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_address_class DW_AT_artificial DW_AT_calling_convention DW_AT_declaration DW_AT_external DW_AT_frame_base DW_AT_high_pc DW_AT_inline DW_AT_low_pc DW_AT_name DW_AT_prototyped DW_AT_return_addr DW_AT_segment DW_AT_sibling DW_AT_specification DW_AT_start_scope DW_AT_static_link DW_AT_type DW_AT_visibility DW_AT_virtuality DW_AT_vtable_elem_location

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

TAG NAME	APPLICABLE ATTRIBUTES
DW_TAG_subrange_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_byte_size DW_AT_count DW_AT_declaration DW_AT_lower_bound DW_AT_name DW_AT_sibling DW_AT_type DW_AT_upper_bound DW_AT_visibility
DW_TAG_subroutine_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_address_class DW_AT_declaration DW_AT_name DW_AT_prototyped DW_AT_sibling DW_AT_start_scope DW_AT_type DW_AT_visibility
DW_TAG_template_type_param	DECL DW_AT_name DW_AT_sibling DW_AT_type
DW_TAG_template_value_param	DECL DW_AT_name DW_AT_const_value DW_AT_sibling DW_AT_type
DW_TAG_thrown_type	DECL DW_AT_sibling DW_AT_type
DW_TAG_try_block	DW_AT_abstract_origin DW_AT_high_pc DW_AT_low_pc DW_AT_segment DW_AT_sibling

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

TAG NAME	APPLICABLE ATTRIBUTES
DW_TAG_typedef	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_declaration DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_type DW_AT_visibility
DW_TAG_union_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_byte_size DW_AT_declaration DW_AT_friends DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_visibility
DW_TAG_unspecified_parameters	DECL DW_AT_abstract_origin DW_AT_artificial DW_AT_sibling
DW_TAG_variable	DECL DW_AT_accessibility DW_AT_constant_value DW_AT_declaration DW_AT_external DW_AT_location DW_AT_name DW_AT_segment DW_AT_sibling DW_AT_specification DW_AT_start_scope DW_AT_type DW_AT_visibility

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

TAG NAME	APPLICABLE ATTRIBUTES
DW_TAG_variant	DECL DW_AT_accessibility DW_AT_abstract_origin DW_AT_declaration DW_AT_discr_list DW_AT_discr_value DW_AT_sibling
DW_TAG_variant_part	DECL DW_AT_accessibility DW_AT_abstract_origin DW_AT_declaration DW_AT_discr DW_AT_sibling DW_AT_type
DW_TAG_volatile_type	DW_AT_sibling DW_AT_type
DW_TAG_with_statement	DW_AT_accessibility DW_AT_address_class DW_AT_declaration DW_AT_high_pc DW_AT_location DW_AT_low_pc DW_AT_segment DW_AT_sibling DW_AT_type DW_AT_visibility

Appendix 2 -- Organization of Debugging Information

The following diagram depicts the relationship of the abbreviation tables contained in the `.debug_abbrev` section to the information contained in the `.debug_info` section. Values are given in symbolic form, where possible.

Compilation Unit 1 - .debug_info

	length 2 *a1 (abbreviation table offset)* 4
	1 "myfile.c" "Best Compiler Corp: Version 1.3" "mymachine:/home/mydir/src:" DW_LANG_C89 0x0 0x55 DW_FORM_data4 0x0
e1:	2 "char" DW_ATE_unsigned_char 1
e2:	3 *e1*
	4 "POINTER" *e2*
	0

Compilation Unit 2 - .debug_info

length 2 *a1 (abbreviation table offset)* 4
...
4 "strp" *e2*
...

Abbreviation Table - .debug_abbrev

a1:

```
1
DW_TAG_compile_unit
DW_CHILDREN_yes
DW_AT_name          DW_FORM_string
DW_AT_producer      DW_FORM_string
DW_AT_compdir       DW_FORM_string
DW_AT_language      DW_FORM_data1
DW_AT_low_poc       DW_FORM_addr
DW_AT_high_pc       DW_FORM_addr
DW_AT_stmt_list     DW_FORM_indirect
0                   0
```

```
2
DW_TAG_base_type
DW_CHILDREN_no
DW_AT_name          DW_FORM_string
DW_AT_encoding      DW_FORM_data1
DW_AT_byte_size     DW_FORM_data1
0                   0
```

```
3
DW_TAG_pointer_type
DW_CHILDREN_no
DW_AT_type          DW_FORM_ref4
0                   0
```

```
4
DW_TAG_typedef
DW_CHILDREN_no
DW_AT_name          DW_FORM_string
DW_AT_type          DW_FORM_ref4
0                   0
```

```
0
```

Appendix 3 -- Statement Program Examples

Consider this simple source file and the resulting machine code for the Intel 8086 processor:

```
1:   int
2:   main()
     0x239:              push pb
     0x23a:              mov bp,sp
3:   {
4:   printf("Omit needless words\n");
     0x23c:              mov ax,0xaa
     0x23f:              push ax
     0x240:              call _printf
     0x243:              pop cx
5:   exit(0);
     0x244:              xor ax,ax
     0x246:              push ax
     0x247:              call _exit
     0x24a:              pop cx
6:   }
     0x24b:              pop bp
     0x24c:              ret
7:
     0x24d:
```

If the statement program prologue specifies the following:

```
minimum_instruction_length   1
opcode_base                  10
line_base                    1
line_range                   15
```

Then one encoding of the statement program would occupy 12 bytes (the opcode `SPECIAL(`*m*`,` *n*`)` indicates the special opcode generated for a line increment of *m* and an address increment of *n*):

Opcode	Operand	Byte Stream
DW_LNS_advance_pc	LEB128(0x239)	0x2, 0xb9, 0x04
SPECIAL(2, 0)		0xb
SPECIAL(2, 3)		0x38
SPECIAL(1, 8)		0x82
SPECIAL(1, 7)		0x73
DW_LNS_advance_pc	LEB128(2)	0x2, 0x2
DW_LNE_end_sequence		0x0, 0x1, 0x1

An alternate encoding of the same program using standard opcodes to advance the program counter would occupy 22 bytes:

Opcode	Operand	Byte Stream
`DW_LNS_fixed_advance_pc`	`0x239`	`0x9, 0x39, 0x2`
`SPECIAL(2, 0)`		`0xb`
`DW_LNS_fixed_advance_pc`	`0x3`	`0x9, 0x3, 0x0`
`SPECIAL(2, 0)`		`0xb`
`DW_LNS_fixed_advance_pc`	`0x8`	`0x9, 0x8, 0x0`
`SPECIAL(1, 0)`		`0xa`
`DW_LNS_fixed_advance_pc`	`0x7`	`0x9, 0x7, 0x0`
`SPECIAL(1, 0)`		`0xa`
`DW_LNS_fixed_advance_pc`	`0x2`	`0x9, 0x2, 0x0`
`DW_LNE_end_sequence`		`0x0, 0x1, 0x1`

Appendix 4 -- Encoding and decoding variable length data

Here are algorithms expressed in a C-like pseudo-code to encode and decode signed and unsigned numbers in LEB128:

Encode an unsigned integer:

```
do
{
        byte = low order 7 bits of value;
        value >>= 7;
        if (value != 0)         /* more bytes to come */
                set high order bit of byte;
        emit byte;
} while (value != 0);
```

Encode a signed integer:

```
more = 1;
negative = (value < 0);
size = no. of bits in signed integer;
while(more)
{
        byte = low order 7 bits of value;
        value >>= 7;
        /* the following is unnecessary if the implementation of >>=
         * uses an arithmetic rather than logical shift for a signed
         * left operand
         */
        if (negative)
                /* sign extend */
                value |= - (1 << (size - 7));
        /* sign bit of byte is 2nd high order bit (0x40) */
        if ((value == 0 && sign bit of byte is clear) ||
                (value == -1 && sign bit of byte is set))
                more = 0;
        else
                set high order bit of byte;
        emit byte;
}
```

Decode unsigned LEB128 number:

```
result = 0;
shift = 0;
while(true)
{
                        byte = next byte in input;
                        result |= (low order 7 bits of byte << shift);
                        if (high order bit of byte == 0)
                                break;
                        shift += 7;
}
```

Decode signed LEB128 number:

```
result = 0;
shift = 0;
size = no. of bits in signed integer;
while(true)
{
                        byte = next byte in input;
                        result |= (low order 7 bits of byte << shift);
                        shift += 7;
                        /* sign bit of byte is 2nd high order bit (0x40) */
                        if (high order bit of byte == 0)
                                break;
}
if ((shift < size) && (sign bit of byte is set))
                        /* sign extend */
                        result |= - (1 << shift);
```

Appendix 5 -- Call Frame Information Examples

The following example uses a hypothetical RISC machine in the style of the Motorola 88000.

- Memory is byte addressed.
- Instructions are all 4-bytes each and word aligned.
- Instruction operands are typically of the form:

  <destination reg> <source reg> <constant>

- The address for the load and store instructions is computed by adding the contents of the source register with the constant.
- There are 8 4-byte registers:

  R0 always 0
  R1 holds return address on call
  R2-R3 temp registers (not preserved on call)
  R4-R6 preserved on call
  R7 stack pointer.

- The stack grows in the negative direction.

The following are two code fragments from a subroutine called `foo` that uses a frame pointer (in addition to the stack pointer.) The first column values are byte addresses.

```
        ;; start prologue
foo     sub    R7, R7, <fsize>        ; Allocate frame
foo+4   store  R1, R7, (<fsize>-4)    ; Save the return address
foo+8   store  R6, R7, (<fsize>-8)    ; Save R6
foo+12  add    R6, R7, 0              ; R6 is now the Frame ptr
foo+16  store  R4, R6, (<fsize>-12)   ; Save a preserve reg.
        ;; This subroutine does not change R5
        ...
        ;; Start epilogue (R7 has been returned to entry value)
foo+64  load   R4, R6, (<fsize>-12)   ; Restore R4
foo+68  load   R6, R7, (<fsize>-8)    ; Restore R6
foo+72  load   R1, R7, (<fsize>-4)    ; Restore return address
foo+76  add    R7, R7, <fsize>        ; Deallocate frame
foo+80  jump   R                                                   ; Return
foo+84
```

The table for the `foo` subroutine is as follows. It is followed by the corresponding fragments from the `.debug_frame` section.

```
Loc     CFA          R0   R1   R2   R3   R4    R5   R6   R7   R8
foo     [R7]+0       s    u    u    u    s     s    s    s    r1
foo+4   [R7]+fsize   s    u    u    u    s     s    s    s    r1
foo+8   [R7]+fsize   s    u    u    u    s     s    s    s    c4
foo+12  [R7]+fsize   s    u    u    u    s     s    c8   s    c4
foo+16  [R6]+fsize   s    u    u    u    s     s    c8   s    c4
foo+20  [R6]+fsize   s    u    u    u    c12   s    c8   s    c4
foo+64  [R6]+fsize   s    u    u    u    c12   s    c8   s    c4
foo+68  [R6]+fsize   s    u    u    u    s     s    c8   s    c4
foo+72  [R7]+fsize   s    u    u    u    s     s    s    s    c4
foo+76  [R7]+fsize   s    u    u    u    s     s    s    s    r1
foo+80  [R7]+0       s    u    u    u    s     s    s    s    r1
```

notes:

1. R8 is the return address
2. s = same_value rule
3. u = undefined rule
4. rN = register(N) rule
5. cN = offset(N) rule

Common Information Entry (CIE):

```
cie     32                       ; length
cie+4   0xffffffff               ; CIE_id
cie+8   1                        ; version
cie+9   0                        ; augmentation
cie+10  4                        ; code_alignment_factor
cie+11  4                        ; data_alignment_factor
cie+12  8                        ; R8 is the return addr.
cie+13  DW_CFA_def_cfa (7, 0)    ; CFA = [R7]+0
cie+16  DW_CFA_same_value (0)    ; R0 not modified (=0)
cie+18  DW_CFA_undefined (1)     ; R1 scratch
cie+20  DW_CFA_undefined (2)     ; R2 scratch
cie+22  DW_CFA_undefined (3)     ; R3 scratch
cie+24  DW_CFA_same_value (4)    ; R4 preserve
cie+26  DW_CFA_same_value (5)    ; R5 preserve
cie+28  DW_CFA_same_value (6)    ; R6 preserve
cie+30  DW_CFA_same_value (7)    ; R7 preserve
cie+32  DW_CFA_register (8, 1)   ; R8 is in R1
cie+35  DW_CFA_nop               ; padding
cie+36
```

Frame Description Entry (FDE):

```
fde      40                             ; length
fde+4    cie                            ; CIE_ptr
fde+8    foo                            ; initial_location
fde+12   84                             ; address_range
fde+16   DW_CFA_advance_loc(1)          ; instructions
fde+17   DW_CFA_def_cfa_offset(<fsize>/4)   ; assuming <fsize> < 512
fde+19   DW_CFA_advance_loc(1)
fde+20   DW_CFA_offset(8,1)
fde+22   DW_CFA_advance_loc(1)
fde+23   DW_CFA_offset(6,2)
fde+25   DW_CFA_advance_loc(1)
fde+26   DW_CFA_def_cfa_register(6)
fde+28   DW_CFA_advance_loc(1)
fde+29   DW_CFA_offset(4,3)
fde+31   DW_CFA_advance_loc(11)
fde+32   DW_CFA_restore(4)
fde+33   DW_CFA_advance_loc(1)
fde+34   DW_CFA_restore(6)
fde+35   DW_CFA_def_cfa_register(7)
fde+37   DW_CFA_advance_loc(1)
fde+38   DW_CFA_restore(8)
fde+39   DW_CFA_advance_loc(1)
fde+40   DW_CFA_def_cfa_offset(0)
fde+42   DW_CFA_nop                     ; padding
fde+43   DW_CFA_nop                     ; padding
fde+44
```

Table of Contents

List of Figures