DWARF Debugging Information Format

Draft Draft Draft

DWARF Working Group

Michael Eager, Chairperson

Ron Brender, Editor

Based on the

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

document that was

Table of Contents

List of Figures

FOREWORD

This document specifies a third generation of symbolic debugging information based on the DWARF format that was developed by the UNIX International Programming Languages Special Interest Group (SIG). The specifications presented are informally named "DWARF Version 3"; they are upward compatible with the UNIX International DWARF Debugging Information Format Revision: 2.0.0 (July 27, 1993) with very minor exceptions that are believed to be of no practical consequence.

This document is intended to be usable in online as well as traditional paper forms. In the online form, blue text is used to indicate what are commonly called "hyperlinks" or "hot links"; these facilitate moving around in the document in a manner like that typically found in web browsers. The Table of Contents also provides hyperlinks to the respective sections.

In the traditional paper form, the appearance of the hyperlinks on a page of paper does not distract the eye because the blue hyperlinks are typically imaged by black and white printers in a manner indistinguishable from other text. (Hyperlinks are not underlined for this same reason.) Page numbers, a Table of Contents and an Index appropriate to a paper document are available in both forms.

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 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. This document is believed to cover most debugging information needs of C, C++, Fortran, Modula2 and Pascal; it also covers the basic needs of various other languages.

This document describes DWARF Version 3, the third generation of debugging information based on the DWARF format. DWARF Version 3 extends DWARF Version 2 in an upwardly compatible manner to add improved language support for several languages. DWARF Version 3 retains the DWARF Version 2 style of representation. It adds an additional format that is able to accommodate DWARF descriptions that exceed 4 GBytes in size. This new format may be attractive for use with large applications on computer systems that support 64-bit addresses. (DWARF Version 3 is not needed to support 64-bit addresses as such; DWARF Version 2 is already sufficient for that.)

The intended audience for this document is 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 is 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. The encoding of the DWARF information is presented in Section seven.

This organization closely follows that used in the DWARF Version 2 document. Except where needed to incorporate new material or to correct errors, text from the DWARF Version 2 text is generally reused in this document with little or no modification.

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. The several Appendices all consist only of explanatory or supplementary material, and are not part of the formal definition.

1.3   Vendor Extensibility

This document does not attempt to cover all interesting languages or even to cover all of the interesting debugging information needs for its primary target languages (C, C++, Fortran, Modula2, and 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. Vendors may also use debugging information entries and attributes defined here in new situations. 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.

1.4   Changes from Version 1 to Version 2

DWARF Version 2 describes 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. The definers believe that the reduction in I/O and in memory paging should more than make up for any increase in processing time.

The representation of information changed from Version 1 to Version 2, so that 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`.

A summary of the major changes made in DWARF Version 2 (published July 27, 1993) compared to the DWARF Version 1 (published January 20, 1992) may be found in the DWARF Version 2 document.

1.5   Changes from Version 2 to Version 3

The following is a list of the major changes made to the DWARF Debugging Information Format since Version 2 of the format was published (July 27, 1993). The list is not meant to be exhaustive.

- Provision is made for DWARF information that requires more than 4 GBytes to represent.

- Attributes can refer to debugging information entries in other shared libraries.

- More complete support is added for Fortran 90 modules as well as allocatable array and pointer types.

- Additional base types are added for C (as revised for 1999).

- Some support is added for Java.

- Namespace support is added for C++.

- An optional section is added for global type names (similar to the global section for objects and functions)

- UTF-8 is adopted as the preferred representation of program name strings.

- Improved support is added for optimized code (discontiguous scopes, end of prologue determination, multiple section code generation).

- Ability to eliminate duplicate DWARF information during linking.

1.5.1 Upward Compatibility

DWARF Version 3 is structurally upward compatible with DWARF Version 2 except as follows:

- Certain very large values of the initial length fields that begin DWARF sections as well as certain structures are reserved to act as escape codes for future extension; one such extension is defined to increase the possible size of DWARF descriptions (see Section 7.4).

- References that use the attribute form DW_FORM_ref_addr are specified to be four bytes in the DWARF 32-bit format and eight bytes in the DWARF 64-bit format, while DWARF Version 2 specifies that such references have the same size as an address on the target system (see Sections 7.4 and 7.5.4).

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 entry descriptions in Sections three, four and five generally include mention of most, but not necessarily all, of the attributes that are normally or possibly used with the entry. Some attributes, whose applicability tends to be pervasive and invariant across kinds of debugging information entries, are described in this Section and not necessarily mentioned in all contexts where they may be appropriate. Examples include DW_AT_artificial, the declaration coordinates, and DW_AT_description, among others.

The debugging information entries in DWARF Version 3 are intended to exist in the `.debug_info` section of an object file.

DW_TAG_access_declaration	DW_TAG_enumerator
DW_TAG_array_type	DW_TAG_file_type
DW_TAG_base_type	DW_TAG_formal_parameter
DW_TAG_catch_block	DW_TAG_friend
DW_TAG_class_type	DW_TAG_imported_declaration
DW_TAG_common_block	DW_TAG_imported_module
DW_TAG_common_inclusion	DW_TAG_imported_unit
DW_TAG_compile_unit	DW_TAG_inheritance
DW_TAG_const_type	DW_TAG_inlined_subroutine
DW_TAG_constant	DW_TAG_interface_type
DW_TAG_dwarf_procedure	DW_TAG_label
DW_TAG_entry_point	DW_TAG_lexical_block
DW_TAG_enumeration_type	DW_TAG_member

DW_TAG_module

DW_TAG_subrange_type

DW_TAG_mutable_type

DW_TAG_subroutine_type

DW_TAG_namelist

DW_TAG_template_type_parameter

DW_TAG_namelist_item

DW_TAG_template_value_parameter

DW_TAG_namespace

DW_TAG_thrown_type

DW_TAG_packed_type

DW_TAG_try_block

DW_TAG_partial_unit

DW_TAG_typedef

DW_TAG_pointer_type

DW_TAG_union_type

DW_TAG_ptr_to_member_type

DW_TAG_unspecified_parameters

DW_TAG_reference_type

DW_TAG_unspecified_type

DW_TAG_restrict_type

DW_TAG_variable

DW_TAG_set_type

DW_TAG_variant

DW_TAG_string_type

DW_TAG_variant_part

DW_TAG_structure_type

DW_TAG_volatile_type

DW_TAG_subprogram

DW_TAG_with_stmt

Figure 1. Tag names

2.2 Attribute Types

Each attribute value is characterized by an attribute name. No more than one attribute with a given name may appear in any debugging information entry. There are no limitations on the ordering of attributes within a debugging information entry.

The defined attribute names and their related uses are listed in Figure 2.

Attribute	Identifies or Specifies
DW_AT_abstract_origin	Inline instances of inline subprograms Out-of-line instances of inline subprograms

Attribute	Identifies or Specifies
DW_AT_accessibility	C++ and Ada declarations C++ base classes C++ inherited members
DW_AT_address_class	Pointer or reference types Subroutine or subroutine type
DW_AT_allocated	Allocation status of types
DW_AT_artificial	Objects or types that are not actually declared in the source
DW_AT_associated	Association status of types
DW_AT_base_types	Primitive data types of compilation unit
DW_AT_bit_offset	Base type bit location Data member bit location
DW_AT_bit_size	Base type bit size Data member bit size
DW_AT_byte_size	Data object or data type size
DW_AT_call_column	Column position of inlined subroutine call
DW_AT_call_file	File containing inlined subroutine call
DW_AT_call_line	Line number of inlined subroutine call
DW_AT_calling_convention	Subprogram calling convention
DW_AT_common_reference	Common block usage
DW_AT_comp_dir	Compilation directory

Attribute	Identifies or Specifies
DW_AT_const_value	Constant object Enumeration literal value Template value parameter
DW_AT_containing_type	Containing type of pointer to member type
DW_AT_count	Elements of subrange type
DW_AT_data_location	Indirection to actual data
DW_AT_data_member_location	Data member location Inherited member location
DW_AT_decl_column	Column position of source declaration
DW_AT_decl_file	File containing source declaration
DW_AT_decl_line	Line number of source declaration
DW_AT_declaration	Incomplete, non-defining, or separate entity declaration
DW_AT_default_value	Default value of parameter
DW_AT_description	Artificial name or description
DW_AT_discr	Discriminant of variant part
DW_AT_discr_list	List of discriminant values
DW_AT_discr_value	Discriminant value
DW_AT_encoding	Encoding of base type
DW_AT_entry_pc	Entry address of module initialization Entry address of subprogram Entry address of inlined subprogram

Attribute	Identifies or Specifies
DW_AT_extension	Previous namespace extension or original namespace
DW_AT_external	External subroutine External variable
DW_AT_frame_base	Subroutine frame base address
DW_AT_friend	Friend relationship
DW_AT_high_pc	Contiguous range of code addresses
DW_AT_identifier_case	Identifier case rule
DW_AT_import	Imported declaration Imported unit Namespace alias Namespace using declaration Namespace using directive
DW_AT_inline	Abstract instance Inlined subroutine
DW_AT_is_optional	Optional parameter
DW_AT_language	Programming language
DW_AT_location	Data object location
DW_AT_low_pc	Code address or range of addresses
DW_AT_lower_bound	Lower bound of subrange
DW_AT_macro_info	Macro information (#define, #undef)
DW_AT_name	Name of declaration Path name of compilation source

Attribute	Identifies or Specifies
DW_AT_namelist_item	Namelist item
DW_AT_ordering	Array row/column ordering
DW_AT_priority	Module priority
DW_AT_producer	Compiler identification
DW_AT_prototyped	Subroutine prototype
DW_AT_ranges	Non-contiguous range of code addresses
DW_AT_return_addr	Subroutine return address save location
DW_AT_segment	Addressing information
DW_AT_sibling	Debugging information entry relationship
DW_AT_specification	Incomplete, non-defining, or separate declaration corresponding to a declaration
DW_AT_start_scope	Inlined subprogram Object declaration Type declaration
DW_AT_static_link	Location of uplevel frame
DW_AT_stmt_list	Line number information for unit
DW_AT_stride	Array dimension stride (from enumeration) Array dimension stride (from subrange)
DW_AT_stride_size	Array element stride (from array type)
DW_AT_string_length	String length of string type
DW_AT_trampoline	Target subroutine

Attribute	Identifies or Specifies
DW_AT_type	Type of declaration Type of subroutine return
DW_AT_upper_bound	Upper bound of subrange
DW_AT_use_location	Member location for pointer to member type
DW_AT_use_UTF8	Compilation unit uses UTF-8 strings
DW_AT_variable_parameter	Non-constant parameter flag
DW_AT_virtuality	Virtuality indication Virtuality of base class Virtuality of function
DW_AT_visibility	Visibility of declaration
DW_AT_vtable_elem_location	Virtual function vtable slot

Figure 2. Attribute names

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

Attribute value forms may belong to one of the classes shown in Figure 3.

Attribute Class	General Use and Encoding
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.). *Most constant values are integers of one kind or another (codes, offsets, counts, and so on); these are sometimes called "integer constants" for emphasis.*
flag	A small constant that indicates the presence or absence of an attribute.
lineptr	Refers to a location in the DWARF section that holds line number information.
loclistptr	Refers to a location in the DWARF section that holds location lists, which describe objects whose location can change during their lifetime.
macptr	Refers to a location in the DWARF section that holds macro definition information.
rangelistptr	Refers to a location in the DWARF section that holds non-contiguous address ranges.
reference	Refers to one of the 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 offset of a debugging information entry in any compilation unit, including one different 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.

Figure 3. Classes of Attribute Value

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.

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.

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 a sibling. If an entry is defined to have children, the next physically succeeding entry is its first child. Additional children 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, while ignoring the children of individual siblings, that producer may attach a DW_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   DWARF Expressions

DWARF expressions describe how to compute a value or name a location during debugging of a program. They are expressed in terms of DWARF operations that operate on a stack of values.

All DWARF 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.

In addition to the general operations that are defined here, additional register name operations (which are specific to location expressions) are defined in Section 2.5.1.

In DWARF Version 2, all DWARF expressions were called "location expressions", whether they computed a location (address, register) or not. This revision defines DWARF expressions as the primary concept, and then defines location expressions as DWARF expressions that are used to compute or name a location.

2.4.1 General Operations

Each general 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 DWARF expression is taken to be the result (the address of the object, the value of the array bound, the length of a dynamic string, and so on).

2.4.1.1 Literal Encodings

The following operations all push a value onto the DWARF stack**.**

1. **DW_OP_lit0**, **DW_OP_lit1**, **...,** **DW_OP_lit31**
   The DW_OP_litn 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.

2.4.1.2 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 description 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 register which is defined with an unsigned LEB128 number, followed by a signed LEB128 offset.

2.4.1.3 Stack Operations

The following operations manipulate the DWARF stack. Operations that index the 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 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 a 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.

11. **DW_OP_push_object_address**

    The DW_OP_push_object_address operation pushes the address of the object currently being evaluated as part of evaluation of a user presented expression. This object may correspond to an independent variable described by its own DIE or it may be a component of an array, structure, or class whose address has been dynamically determined by an earlier step during user expression evaluation.

    This operator provides functionality explicitly and generally (especially for arrays involving descriptors) that is analogous to the implicit push of the base address of a structure prior to evaluation of a DW_AT_data_member_location to access a data member of a structure. For an example, see Appendix D.2.

2.4.1.4 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, interprets it as a signed value and pushes its absolute value. If the absolute value can not be represented, the result is undefined.

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_litn DW_OP_plus".

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 (filling with zero bits) 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 (divide the magnitude by 2, keep the same sign for the result) 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 bitwise *exclusive-or* operation on the two, and pushes the result.

2.4.1.5 Control Flow Operations

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

1. **DW_OP_le**, **DW_OP_ge**, **DW_OP_eq**, **DW_OP_lt**, **DW_OP_gt, DW_OP_ne**
   The six relational operators each:

   - pop the top two stack values,

   - compare the former top of the stack with the former second entry, and

   - push 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 DWARF 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 DWARF expression to skip from the current operation, beginning after the 2-byte constant.

4. **DW_OP_call2**, **DW_OP_call4**, **DW_OP_call_ref**
   DW_OP_call2, DW_OP_call4, and DW_OP_call_ref perform subroutine calls during evaluation of a DWARF expression. For DW_OP_call2 and DW_OP_call4, the operand is the 2- or 4-byte unsigned offset, respectively, of a debugging information entry in the current compilation unit. The DW_OP_call_ref operator has a single operand. In the 32-bit DWARF format, the operand is a 4-byte unsigned value; in the 64-bit DWARF format, it is an 8-byte unsigned value (see Section 7.4). The operand is used as the offset of a debugging information entry in a .debug_info section which may be contained in a shared object for executable other than that containing the operator. For references from one shared object or executable to another, the relocation must be performed by the consumer.

   Operand interpretation of DW_OP_call2, DW_OP_call4 and DW_OP_call_ref is exactly like that for DW_FORM_ref2, DW_FORM_ref4 and DW_FORM_ref_addr, respectively.

   These operations transfer control of DWARF expression evaluation to the DW_AT_location attribute of the referenced DIE. If there is no such attribute, then there is no effect. Execution of the DWARF expression of a DW_AT_location attribute may add to and/or remove from values on the stack. Execution returns to the point following the call when the end of the attribute is reached. Values on the stack at the time of the call may be used as parameters by the called expression and values left on the stack by the called expression may be used as return values by prior agreement between the calling and called expressions.

2.4.1.6 Special Operations

There are two special operations currently defined:

1. **DW_OP_piece**
   DW_OP_piece takes a single operand, which is an unsigned LEB128 number. The number describes the size in bytes of the piece of the object referenced by the DWARF expression whose result is at the top of the stack.

   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 DWARF expression refers to.

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.

2.4.2 Example Stack Operations

The stack operations defined in Section 2.4.1.3 are fairly conventional, but the following examples illustrate their behavior graphically.

```
Before              Operation          After

0     17         DW_OP_dup          0      17
1     29                            1      17
2   1000                            2      29
                                    3    1000

0     17         DW_OP_drop         0      29
1     29                            1    1000
2   1000

0     17         DW_OP_pick 2       0    1000
1     29                            1      17
2   1000                            2      29
                                    3    1000

0     17         DW_OP_over         0      29
1     29                            1      17
2   1000                            2      29
                                    3    1000

0     17         DW_OP_swap         0      29
1     29                            1      17
2   1000                            2    1000

0     17         DW_OP_rot          0      29
1     29                            1    1000
2   1000                            2      17
```

2.5 Location Descriptions

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, debugging information must be able to describe the location of an object whose location changes over the object's lifetime.

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

1. *Location expressions,* which are a language independent representation of addressing rules of arbitrary complexity built from DWARF expressions. 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 using class block and a location list is encoded using class loclistptr (which serves as an offset into a separate location list table).

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

2.5.1 Register Name Operators

The following DWARF operations can be used to name a register. They can be used only in location expressions. Each register name operator must be used alone (as a DWARF expression consisting of just that one operation).

Note that the register number represents a DWARF specific mapping of numbers onto the actual registers of a given architecture. The mapping should be chosen to gain optimal density and should be shared by all users of a given architecture. It is recommended that this mapping be defined by the ABI 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.

2.5.2 Location Expressions

A location expression consists of zero or more DWARF 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.

In the case of locations used for structure members, the computation implicitly pushes the base address of the immediately containing structure on the stack before evaluation of the addressing operation.

2.5.3 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_regn, or DW_OP_regx operations are used.

Here are some examples of how DWARF 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.*

2.5.4 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 an 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 is either a location list entry, a base address selection entry, or an end
of list entry.

A location list entry consists of:

1. A beginning address. This address is relative to the applicable 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 applicable 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. The ending address must be greater than the beginning address.

3. A location expression describing the location of the object over the range specified by the
    beginning and end addresses.

The applicable base address of a location list entry is determined by the closest preceding base
address selection entry (see below) in the same location list. If there is no such selection entry,
then the applicable base address defaults to the base address of the compilation unit (see
Section 3.1).

In the case of a compilation unit where all of the machine code is contained in a single contiguous section, no base address selection entry is needed.

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.

A base address selection entry consists of:

1. The value of the largest representable address.

   This value is `0xffffffff` *in the 32-bit DWARF format and* `0xffffffffffffffff` *in the 64-bit DWARF format (see Section 7.4).*

2. An address, which defines the appropriate base address for use in interpreting the beginning and ending relative addresses of subsequent entries of the location list.

A base address selection entry affects only the list in which it is contained.

The end of any given location list is marked by an end of list entry, which consists of a 0 for the beginning address and a 0 for the ending address. A location list containing only an end of list entry describes an object that exists in the source code but not in the executable program.

Neither a base address selection entry nor an end of list entry includes a location expression.

A base address selection entry and an end of list entry for a location list are identical to a base address selection entry and end of list entry, respectively, for a range list (see Section 2.16.3) in interpretation and representation.

2.6   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.

2.7   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 4.

| DW_ACCESS_public |
| DW_ACCESS_private |
| DW_ACCESS_protected |

Figure 4. Accessibility codes

2.8   Visibility of Declarations

Several languages have the concept of the visibility of a declaration. The visibility specifies which declarations are to be visible outside of the entity 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 5.

| DW_VIS_local |
| DW_VIS_exported |
| DW_VIS_qualified |

Figure 5. Visibility codes

2.9 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 6.

DW_VIRTUALITY_none
DW_VIRTUALITY_virtual
DW_VIRTUALITY_pure_virtual

Figure 6. Virtuality codes

2.10 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.

2.11 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, DW_AT_high_pc, DW_AT_ranges or DW_AT_entry_pc attribute, or a location description that evaluates to an address, then those address 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 an integer 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 7. Example address class codes

2.12 Non-Defining Declarations and Completions

A debugging information entry representing a program entity typically represents the defining declaration of that entity. In certain contexts, however, a debugger might need information about a declaration of an entity that is not also a definition, or is otherwise incomplete, 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.

2.12.1 Non-Defining Declarations

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

2.12.2 Declarations Completing Non-Defining Declarations

Debugging information entries that represent a declaration that completes another (earlier) non-defining declaration, may have a DW_AT_specification attribute whose value is a reference to the debugging information entry representing the non-defining declaration. Debugging information entries with a DW_AT_specification attribute do not need to duplicate information provided by the debugging information entry referenced by that specification attribute.

2.13 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 an integer constant.

The value of the DW_AT_decl_file attribute corresponds to a file number from the line number information table for the compilation unit containing the 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.

2.14 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.

Because 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` *(in names such as* `"operator +"`*), if present. Sequences of multiple whitespace characters may be compressed.*

2.15 Data Locations

Any debugging information entry describing a data object, which includes variables, parameters, common blocks and the like, may have a DW_AT_location attribute, whose value is a location description (see Section 2.5).

2.16 Code Addresses and Ranges

Any debugging information entry describing an entity that has a machine code address or range of machine code addresses, which includes compilation units, module initialization, subroutines, ordinary blocks, try/catch blocks, labels and the like, may have

- A DW_AT_low_pc attribute for a single address,

- A DW_AT_low_pc and DW_AT_high_pc pair of attributes for a single contiguous range of addresses, or

- A DW_AT_ranges attribute for a non-contiguous range of addresses.

If an entity has no associated machine code, none of these attributes are specified.

2.16.1 Single Address

When there is a single address associated with an entity, such as a label or alternate entry point of a subprogram, the entry has a DW_AT_low_pc attribute whose value is the relocated address for the entity.

While the DW_AT_entry_pc attribute might also seem appropriate for this purpose, historically the DW_AT_low_pc attribute was used before the DW_AT_entry_pc was introduced (in DWARF V3). There is insufficient reason to change this.

2.16.2 Contiguous Address Range

When the set of addresses of a debugging information entry can be described as a single contiguous range, the entry may have a DW_AT_low_pc and DW_AT_high_pc pair of attributes. The value of the DW_AT_low_pc attribute is the relocated address of the first instruction associated with the entity, and the value of the DW_AT_high_pc is the relocated address of the first location past the last instruction associated with the entity.

The high PC value may be beyond the last valid instruction in the executable.

The presence of low and high PC attributes for an entity implies that the code generated for the entity 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.

2.16.3 Non-Contiguous Address Ranges

When the set of addresses of a debugging information entry cannot be described as a single contiguous range, the entry has a DW_AT_ranges attribute whose value is of class rangelistptr and indicates the beginning of a range list.

Range lists are contained in a separate object file section called `.debug_ranges`. A range list is indicated by a DW_AT_ranges attribute whose value is represented as an offset from the beginning of the `.debug_ranges` section to the beginning of the range list.

Each entry in a range list is either a range list entry, a base address selection entry, or an end of list entry.

A range list entry consists of:

1. A beginning address. This address is relative to the applicable base address of the compilation unit referencing this range list.

2. An ending address, again relative to the applicable base address of the compilation unit referencing this range list. The ending address must be greater than the beginning address.

The applicable base address of a range list entry is determined by the closest preceding base address selection entry (see below) in the same range list. If there is no such selection entry, then the applicable base address defaults to the base address of the compilation unit (see Section 3.1).

In the case of a compilation unit where all of the machine code is contained in a single contiguous section, no base address selection entry is needed.

Address range entries in a range list may not overlap. There is no requirement that the entries be ordered in any particular way.

A base address selection entry consists of:

1. The value of the largest representable address.

    This value is `0xffffffff` in the 32-bit DWARF format and `0xffffffffffffffff` in the 64-bit DWARF format (see Section 7.4).

2. An address, which defines the appropriate base address for use in interpreting the beginning and ending relative addresses of subsequent entries of the location list.

A base address selection entry affects only the list in which it is contained.

The end of any given range list is marked by an end of list entry, which consists of a 0 for the beginning address and a 0 for the ending address. A range list containing only an end of list entry describes an empty scope (which contains no instructions).

A base address selection entry and an end of list entry for a range list are identical to a base address selection entry and end of list entry, respectively, for a location list (see Section 2.5.4) in interpretation and representation.

2.17 Entry Address

The entry or first executable instruction generated for an entity, if applicable, is often the lowest addressed instruction of a contiguous range of instructions. In other cases, the entry address needs to be specified explicitly.

Any debugging information entry describing an entity that has a range of code addresses, which includes compilation units, module initialization, subroutines, ordinary blocks, try/catch blocks, and the like, may have a DW_entry_pc attribute to indicate the first executable instruction within that range of addresses. The value of the DW_AT_entry_pc attribute is a relocated address. If no DW_AT_entry_pc attribute is present, then the entry address is assumed to be the same as the value of the DW_AT_low_pc attribute, if present; otherwise, the entry address is unknown.

2.18 Static and Dynamic Properties of Types

Some attributes that apply to types specify a property (such as the lower bound of an array) that is an integer value, where the value may be known during compilation or may be computed dynamically during execution. The value of these attributes is determined based on the class as follows:

- For a constant, the value of the constant is the value of the attribute.

- For a reference, the value is a reference to another entity whose value is the value of the attribute.

- For a block, the value is interpreted as a DWARF expression; evaluation of the expression yields the value of the attribute.

Whether an attribute value can be dynamic depends on the rules of the applicable programming language.

The applicable attributes include: DW_AT_allocated, DW_AT_associated, DW_AT_bit_offset, DW_AT_bit_size, DW_AT_byte_size, DW_AT_count, DW_AT_lower_bound, DW_AT_stride, DW_AT_stride_size, DW_AT_upper_bound (and possibly others).

2.19 Entity Descriptions

Some debug information entries may describe entities in the program that are artificial, or which otherwise are "named" in ways which are not valid identifiers in the programming language. For example, several languages may capture or freeze the value of a variable at a particular point in the program. Ada 95 has package elaboration routines, type descriptions of the form `typename`*'Class, and "access typename" parameters.*

Generally, any debug information entry that has, or may have, a DW_AT_name attribute, may also have a DW_AT_description attribute whose value is a null-terminated string representing the description of the entity.

It is expected that a debugger will only display these descriptions as part of the description of other entities. It should not accept them in expressions, nor allow them to be assigned, or the like.

3   PROGRAM SCOPE ENTRIES

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

3.1   Compilation and Importing Entries

An object file may contain one or more compilation units, of which there are two kinds: normal compilation units and partial compilation units. A partial compilation unit is related to one or more other compilation units that import it.

3.1.1   Normal and Partial Compilation Unit Entries

A normal compilation unit is represented by a debugging information entry with the tag DW_TAG_compile_unit. A partial compilation unit is represented by a debugging information entry with the tag DW_TAG_partial_unit.

In a simple normal compilation, a single compilation unit with the tag DW_TAG_compile_unit represents a complete object file and the tag DW_TAG_partial_unit is not used. In a compilation employing DWARF space compression and duplicate elimination techniques (see Appendix E), multiple compilation units using the tags DW_TAG_compile_unit and/or DW_TAG_partial_unit are used to represent portions of an object file.

A normal 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." A partial compilation unit typically represents a part of the text and data of a relocatable object file, in a manner that can potentially be shared with the results of other compilations to save space. It may be derived from an "include file", template instantiation, or other implementation-dependent portion of a compilation. A normal compilation unit can also function in a manner similar to a partial compilation unit in some cases.

A compilation unit entry owns debugging information entries that represent all or part of the declarations made in the corresponding compilation. In the case of a partial compilation unit, the containing scope of its owned declarations is indicated by imported unit entries in one or more other compilation unit entries that refer to that partial compilation unit (see Section 3.1.2).

Compilation unit entries may have the following attributes:

1. Either a DW_AT_low_pc and DW_AT_high_pc pair of attributes or a DW_AT_ranges attribute whose values encode the contiguous or non-contiguous address ranges, respectively, of the machine instructions generated for the compilation unit (see Section 2.16).

    A DW_AT_low_pc attribute may also be specified in combination with DW_AT_ranges to specify the default base address for use in location lists (see Section 2.5.4) and range lists (see Section 2.16.3).

2. 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.

3. A DW_AT_language attribute whose constant value is an integer code indicating the source language of the compilation unit. The set of language names and their meanings are given in Figure 8.

DW_LANG_Ada83†	ISO/ANSI Ada:1983
DW_LANG_Ada95†	ISO/ANSI Ada:1995
DW_LANG_C	Non-ANSI C, such as K&R
DW_LANG_C89	ISO/ANSI C:1989
DW_LANG_C99	ISO/ANSI C:1999
DW_LANG_C_plus_plus	ISO/ANSI C++:1998
DW_LANG_Cobol74†	ANSI Cobol:1974
DW_LANG_Cobol85†	ANSI Cobol:1985
DW_LANG_Fortran77	ISO/ANSI FORTRAN 77
DW_LANG_Fortran90	ISO/ANSI Fortran 90
DW_LANG_Fortran95	ISO/ANSI Fortran 95
DW_LANG_Java†	Java

DW_LANG_Modula2	ISO Modula-2:1996
DW_LANG_Pascal83	ISO/ANSI Pascal
DW_LANG_PLI†	ANSI PL/I:1976

†Support for these languages is limited.

Figure 8. Language names

4. A DW_AT_stmt_list attribute whose value is a section offset to the 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).

5. A DW_AT_macro_info attribute whose value is a section offset 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).

6. 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".

7. 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.

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

DW_ID_case_sensitive
DW_ID_up_case
DW_ID_down_case
DW_ID_case_insensitive

Figure 9. 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.

9. 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.

10. A DW_AT_use_UTF8 attribute, which is a flag whose presence indicates that all strings (such as the names of declared entities in the source program) are represented using the UTF-8 representation (see Section 7.5.4).

The base address of a compilation unit is defined as the value of the DW_AT_low_pc attribute, if present; otherwise, it is undefined. If the base address is undefined, then any DWARF entry or structure defined in terms of the base address of that compilation unit is not valid.

3.1.2 Imported Unit Entries

An imported unit entry is represented by a debugging information entry with the tag DW_TAG_imported_unit. An imported unit entry contains a DW_AT_import attribute whose value is a reference to the normal or partial compilation unit unit whose declarations logically belong at the place of the imported unit entry.

An imported unit entry does not necessarily correspond to any entity or construct in the source program. It is merely "glue" used to relate a partial unit, or a compilation unit used as a partial unit, to a place in some other compilation unit.

3.2 Module, Namespace and Importing Entries

Modules and namespaces provide a means to collect related entities into a single entity and to manage the names of those entities.

3.2.1 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.

The module entry may have either a DW_AT_low_pc and DW_AT_high_pc pair of attributes or a DW_AT_ranges attribute whose values encode the contiguous or non-contiguous address ranges, respectively, of the machine instructions generated for the module initialization code (see

Section 2.16). It may also have a DW_AT_entry_pc attribute whose value is the address of the first executable instruction of that initialization code (see Section 2.17).

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 declaration attribute (DW_AT_declaration). A Fortran 90 module may also be represented by a module entry (but no declaration attribute is warranted because Fortran has no concept of a corresponding module body).

3.2.2 Namespace Entries

C++ has the notion of a namespace, which provides a way to implement name hiding, so that names of unrelated things do not accidentally clash in the global namespace when an application is linked together.

A namespace is represented by a debugging information entry with the tag DW_TAG_namespace. A namespace extension is represented by a DW_TAG_namespace entry with a DW_AT_extension attribute referring to the previous extension, or if there is no previous extension, to the original DW_TAG_namespace entry. A namespace extension entry does not need to duplicate information in a previous extension entry of the namespace nor need it duplicate information in the original namespace entry. (Thus, for a namespace with a name, a DW_AT_name attribute need only be attached directly to the original DW_TAG_namespace entry.)

Namespace and namespace extension entries may own other debugging information entries describing program entities whose declaration scopes end at the end of the namespace itself.

For C++, such owned program entities may be declarations, including certain declarations that are also object or function definitions.

If a type, variable, or function declared in a namespace is defined outside of the body of the namespace declaration, that type, variable, or function definition entry has a DW_AT_specification attribute whose value is a reference to the debugging information entry representing the declaration of the type, variable or function. Type, variable, or function entries with a DW_AT_specification attribute do not need to duplicate information provided by the declaration entry referenced by the specification attribute.

The C++ global namespace (the namespace referred to by "::f", for example) is not explicitly represented in DWARF with a namespace entry (thus mirroring the situation in C++ source). Global items may be simply declared with no reference to a namespace.

The C++ compilation unit specific "unnamed namespace" may be represented by a namespace entry with no name attribute in the original namespace declaration entry (and therefore no name attribute in any namespace extension entry of this namespace).

A compiler emitting namespace information may choose to explicitly represent namespace extensions, or to represent the final namespace declaration of a compilation unit; this is a quality-of-implementation issue and no specific requirements are given here. If only the final namespace is represented, it is impossible for a debugger to interpret using declaration references in exactly the manner defined by the C++ language.

A simple emission of all namespace declaration information in all compilation units could result in a significant increase in the size of the debug information and significant duplication of information across compilation units. The C++ namespace std, *for example, is large and will probably be referenced in every C++ compilation unit.*

For a C++ namespace example, see Appendix D.3.

3.2.3 Imported (or Renamed) Declaration Entries

Some languages support the concept of importing into or making accessible in a given unit declarations made in a different module. An imported declaration may sometimes be given another name.

An imported declaration is represented by one or more debugging information entries with the tag DW_TAG_imported_declaration. When an overloaded entity is imported, there is one imported declaration entry for each overloading. Each imported declaration entry has a DW_AT_import attribute, whose value is a reference to the debugging information entry representing the declaration that is being imported.

An imported declaration may also have a DW_AT_name attribute whose value is a null-terminated string containing the name, as it appears in the source program, by which the imported entity is to be known in the context of the imported declaration entry (which may be different than the name of the entity being imported). If no name is present, then the name by which the entity is to be known is the same as the name of the entity being imported.

An imported declaration entry with a name attribute may be used as a general means to rename or provide an alias for an entity, regardless of the context in which the importing declaration or the imported entity occur.

A C++ namespace alias may be represented by an imported declaration entry with a name attribute whose value is a null-terminated string containing the alias name as it appears in the source program and an import attribute whose value is a reference to the applicable original namespace or namespace extension entry.

A C++ using declaration may be represented by one or more imported declaration entries. When the using declaration refers to an overloaded function, there is one imported declaration entry corresponding to each overloading. Each imported declaration entry has no name attribute but it does have an import attribute that refers to the entry for the entity being imported. (C++ provides no means to "rename" an imported entity, other than a namespace).

A Fortran use statement with an "only list" may be represented by a series of imported declaration entries, one (or more) for each entity that is imported. An entity that is renamed in the importing context may be represented by an imported declaration entry with a name attribute that specifies the new local name.

3.2.4   Imported Module Entries

Some languages support the concept of importing into or making accessible in a given unit all of the declarations contained within a separate module or namespace.

An imported module declaration is represented by a debugging information entry with the tag DW_TAG_imported_module. An imported module entry contains a DW_AT_import attribute whose value is a reference to the module or namespace entry containing the definition and/or declaration entries for the entities that are to be imported into the context of the imported declaration entry.

An imported module declaration may own a set of imported declaration entries, each of which refers to an entry in the module whose corresponding entity is to be known in the context of the imported module declaration by a name other than its name in that module. Any entity in the module that is not referenced in this way is known in the context of the imported module entry by the same name as it is declared in the module.

A C++ using directive may be represented by an imported module entry, with an import attribute referring to the namespace entry of the appropriate extension of the namespace (which might be the original namespace entry), no name attribute and no owned entries.

A Fortran use statement with a "rename list" may be represented by an imported module entry with an import attribute referring to the module, no name attribute and owned entries corresponding to those entities that are renamed as part of being imported.

A Fortran use statement with neither a "rename list" nor an "only list" may be represented by an imported module entry with an import attribute referring to the module, no name attribute and no owned child entries.

A Fortran use statement that for an entity in a module that is itself imported by a use statement without an explicit mention may be represented by an imported declaration entry that refers to the original debugging information entry. For example, given

```
module A
integer X, Y, Z
end module

module B
use A
end module

module C
use B, only Q => X
end module
```

the imported declaration entry for Q within module C refers directly to the variable declaration entry for A in module A because there is no explicit representation for X in module B.

A similar situation arises for a C++ using declaration that imports an entity in terms of a namespace alias. See Appendix D.3 for an example.

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 alternate 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 5.6.7.

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 an integer constant. The set of calling convention codes is given in Figure 10.

| DW_CC_normal |
| DW_CC_program |
| DW_CC_nocall |

Figure 10. Calling convention codes

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 according to 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 may have either a DW_AT_low_pc and DW_AT_high_pc pair of attributes or a DW_AT_ranges attribute whose values encode the contiguous or non-contiguous address ranges, respectively, of the machine instructions generated for the subroutine (see Section 2.16).

A subroutine entry may also have a DW_AT_entry_pc attribute whose value is the address of the first executable instruction of the subroutine (see Section 2.17).

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.

While the DW_AT_entry_pc attribute might also seem appropriate for this purpose, historically the DW_AT_low_pc attribute was used before the DW_AT_entry_pc was introduced (in DWARF V3). There is insufficient reason to change this.

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 code address or range 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 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 will 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 an integer constant. The set of values for the DW_AT_inline attribute is given in Figure 11.

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 11. Inline codes

3.3.8.1   Abstract Instances

Any debugging information entry that is owned (either directly or indirectly) by a debugging information entry that contains the DW_AT_inline attribute is referred to as an "abstract instance entry." Any subroutine entry that contains a DW_AT_inline attribute is 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, is known as an "abstract instance tree." However, in the case where an abstract instance tree is nested within another abstract instance tree, the entries in the nested abstract instance tree are not considered to be entries in the outer abstract instance tree.

Abstract instance trees are defined so that no entry is part of more than one abstract instance tree. This simplifies the following descriptions.

A debugging information entry that is a member of an abstract instance tree should not contain any attributes which describe aspects of the subroutine which vary between distinct inline expansions or distinct out-of-line expansions. For example, the DW_AT_low_pc, DW_AT_high_pc, DW_AT_ranges, DW_AT_entry_pc, DW_AT_location, DW_AT_return_addr, DW_AT_start_scope, and DW_AT_segment attributes typically should be omitted; however, this list is not exhaustive.

It would not make sense normally 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. However, see Appendix D.6.3 for a contrary example.

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 may have either a DW_AT_low_pc and DW_AT_high_pc pair of attributes or a DW_AT_ranges attribute whose values encode the contiguous or non-contiguous address ranges, respectively, of the machine instructions generated for the inlined subroutine (see Section 2.16). An inlined subroutine entry may also contain a DW_AT_entry_pc attribute, representing the first executable instruction of the inline expansion (see Section 2.17).

An inlined subroutine entry may also have DW_AT_call_file, DW_AT_call_line and DW_AT_call_column attributes, each of whose value is an integer constant. These attributes represent the source file, source line number, and source column number, respectively, of the first character of the statement or expression that caused the inline expansion. The call file, call line, and call column attributes are interpreted in the same way as the declaration file, declaration line, and declaration column attributes, respectively (see Section 2.13).

The call file, call line and call column coordinates do not describe the coordinates of the subroutine declaration that was inlined, rather they describe the coordinates of the call.

Any debugging information entry that is owned (either directly or indirectly) by a debugging information entry with the tag DW_TAG_inlined_subroutine is referred to as a "concrete inlined instance entry." Any entry that has the tag DW_TAG_inlined_subroutine is 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, is known as a "concrete inlined instance tree." However, in the case where a concrete instance tree is nested within another concrete instance tree, the entries in the nested concrete instance tree are not considered to be entries in the outer concrete instance tree.

Concrete inline instance trees are defined so that no entry is part of more than one concrete inline instance tree. This simplifies later descriptions.

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.

Concrete inlined instance entries may omit attributes that are not specific to the concrete instance (but present in the abstract instance) and need include only attributes that are specific to the concrete instance (but omitted in the abstract instance). 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.

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. As a consequence, they may usually be omitted from any entry that has an abstract origin attribute.

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_variable can only be associated with another entry that also has the tag DW_TAG_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 inline instance tree will be closely analogous to the structure and content of its associated abstract instance tree. There are a few exceptions:

1. An entry in the concrete instance tree may be omitted if it contains only a DW_AT_abstract_origin attribute and either has no children, or its children are omitted. Such entries would provide no useful information. In C-like languages, such entries frequently include types, including structure, union, class, and interface types; and members of types. If any entry within a concrete inlined instance tree needs to refer to an entity declared within the scope of the relevant inline subroutine and for which no concrete instance entry exists, the reference should refer to the abstract instance entry.

2. Entries in the concrete instance tree which are associated with entries in the abstract instance tree such that neither has a DW_AT_name attribute, and neither is referenced by any other debugging information entry may be omitted. This may happen for debugging information entries in the abstract instance trees that became unnecessary in the concrete instance tree because of additional information available there. For example, an anonymous variable might have been created and described in the abstract instance tree, but because of the actual parameters for a particular inline expansion, it could be described as a constant value without the need for that separate debugging information entry.

3. A concrete instance tree may contain entries which do not correspond to entries in the abstract instance tree to describe new entities that are specific to a particular inline expansion. In that case, they will not have associated entries in the abstract instance tree, should not contain DW_AT_abstract_origin attributes, and must contain all their own attributes directly. This allows an abstract instance tree to omit debugging information entries for anonymous entities that are unlikely to be needed in most inline expansions. In any expansion which deviates from that expectation, the entries can be described in its concrete instance tree.

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. Such concrete instances of inline subroutines are 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).

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 normally owned by the same parent entry that also owns the root entry of the associated abstract instance. However, it is not required that the abstract and out-of-line instance trees are owned by the same parent entry.

3.3.8.4 Nested Inline Subroutines

Some languages and compilers may permit the logical nesting of a subroutine within another subroutine, and may permit either the outer or the nested subroutine, or both, to be inlined.

For a non-inline subroutine nested within an inline subroutine, the nested subroutine is described normally in both the abstract and concrete inlined instance trees for the outer subroutine. All rules pertaining to the abstract and concrete instance trees for the outer subroutine apply also to the abstract and concrete instance entries for the nested subroutine.

For an inline subroutine nested within another inlined subroutine, the following rules apply to their abstract and concrete instance trees:

1. The abstract instance tree for the nested subroutine is described within the abstract instance tree for the outer subroutine according to the rules in Section 3.3.8.1, and without regard to the fact that it is within an outer abstract instance tree.

2. Any abstract instance tree for a nested subroutine is always omitted within the concrete instance tree for an outer subroutine.

3. A concrete instance tree for a nested subroutine is always omitted within the abstract instance tree for an outer subroutine.

4. The concrete instance tree for any inline or out-of-line expansion of the nested subroutine is described within a concrete instance tree for the outer subroutine according to the rules in Sections 3.3.8.2 or 3.3.8.3, respectively, and without regard to the fact that it is within an outer concrete instance tree.

3.3.9 Trampolines

A trampoline is a compiler generated subroutine that acts as a surrogate for another subroutine and serves as an intermediary in making a call to that other subroutine. In the course of doing so, it may adjust parameters and/or the result (if any) as appropriate to the combined calling and called execution contexts.

A trampoline is represented by a debugging information entry with the tag DW_TAG_subprogram or DW_TAG_inlined_subroutine that has a DW_AT_trampoline attribute. The value of that attribute indicates the target subroutine of the trampoline, that is, the subroutine to which the trampoline passes control. (A trampoline entry may but need not also have a DW_AT_artificial attribute.)

The value of the trampoline attribute may be represented using any of the following forms, which are listed in order of preference:

- If the value is of class reference, then the value specifies the debugging information entry of the target subprogram.

- If the value is of class address, then the value is the relocated address of the target subprogram.

- If the value is of class string, then the value is the (possibly mangled) name of the target subprogram.

- If the value is of class flag, then the value *true* indicates that the containing subroutine is a trampoline but that the target subroutine is not known.

The target subprogram may itself be a trampoline. (A sequence of trampolines necessarily ends with a non-trampoline subprogram.)

In C++, trampolines may be used to implement derived virtual member functions; such trampolines typically adjust the implicit this pointer parameter in the course of passing control. Other languages and environments may use trampolines in a manner sometimes known as transfer functions or transfer vectors.

Trampolines may sometimes pass control to the target subprogram using a branch or jump instruction instead of a call instruction, thereby leaving no trace of their existance in the subsequent execution context.

This attribute helps make it feasible for a debugger to arrange that stepping into a trampoline or setting a breakpoint in a trampoline will result in stepping into or setting the breakpoint in the target subroutine instead. This helps to hide the compiler generated subprogram from the user.

If the target subroutine is not known, a debugger may choose to repeatedly step until control arrives in a new subroutine which can be assumed to be the target subroutine.

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 (including 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 may have either a DW_AT_low_pc and DW_AT_high_pc pair of attributes or a DW_AT_ranges attribute whose values encode the contiguous or non-contiguous address ranges, respectively, of the machine instructions generated for the lexical block (see Section 2.16).

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 Modula2 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 may have either a DW_AT_low_pc and DW_AT_high_pc pair of attributes or a DW_AT_ranges attribute whose values encode the contiguous or non-contiguous address ranges, respectively, of the machine instructions generated for the with statement (see Section 2.16).

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 may have either a DW_AT_low_pc and DW_AT_high_pc pair of attributes or a DW_AT_ranges attribute whose values encode the contiguous or non-contiguous address ranges, respectively, of the machine instructions generated for the block (see Section 2.16).

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_declaration attribute, whose value is a flag, which indicates that this entry represents a non-defining declaration of an object.

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

    In a variable entry representing the definition of a 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 2.5), the variable is assumed to exist in the source code but not in the executable program (but see number 10, below)

In a variable entry representing a non-defining declaration of a variable, the location specified modifies the location specified by the defining declaration and only applies for the scope of the variable entry; if no location is specified, then the location specified in the defining declaration applies.

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

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

6. 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 has the tag DW_TAG_member and will be a child of some class, structure or union type entry.

If the variable entry represents a non-defining declaration, *DW_AT_specification* may be used to reference the defining declaration of the variable. If no DW_AT_specification attribute is present, the defining declaration may be found as a global definition either in the current compilation unit or in another compilation unit with the *DW_AT_external* attribute.

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.

7. 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.

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

9. 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.

10. An entry describing a variable or formal parameter 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.

One way in which a formal parameter with a constant value and no location can arise is for a formal parameter of an inlined subprogram that corresponds to a constant actual parameter of a call that is inlined.

11. 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 Namelist Entries

At least one language, Fortran 90, 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 an integer constant. The set of values and their meanings for the DW_AT_encoding attribute is given in Figure 12.

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 (see Section 2.18) is 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 (see Section 2.18) are integers. 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.

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_imaginary_float	imaginary 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 12. Encoding attribute values

5.2   Unspecified Type Entries

Some languages have constructs in which a type may be left unspecified or the absence of a type may be explicitly indicated.

An unspecified (implicit, unknown, ambiguous or nonexistent) type is represented by a debugging information entry with the tag DW_TAG_unspecified_type. If a name has been given to the type, then the corresponding unspecified type entry has a DW_AT_name attribute whose value is a null-terminated string containing the name as it appears in the source program.

The interpretation of this debugging information entry is intentionally left flexible to allow it to be interpreted appropriately in different languages. For example, in C and C++ the language implementation can provide an unspecified type entry with the name "void" which can be referenced by the type attribute of pointer types and typedef declarations for 'void' (see Sections 5.3 and 5.4, respectively). As another example, in Ada such an unspecified type entry can be referred to by the type attribute of an access type where the denoted type is incomplete (the name is declared as a type but the definition is deferred to a separate compilation unit).

5.3 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 13.

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

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.

Tag	Meaning
DW_TAG_const_type	C or C++ const qualified type
DW_TAG_mutable_type	C++ mutable qualified type
DW_TAG_packed_type	Pascal or Ada packed type
DW_TAG_pointer_type	Pointer to an object of the type being modified.
DW_TAG_reference_type	C++ reference to an object of the type being modified
DW_TAG_restrict_type	C restrict qualified type
DW_TAG_volatile_type	C or C++ volatile qualified type

Figure 13. Type modifier tags

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

```
const unsigned char * volatile p;
    which represents a volatile pointer to a constant
    character. This is encoded in DWARF as:
        DW_TAG_variable(p) →
            DW_TAG_volatile_type →
                DW_TAG_pointer_type →
                    DW_TAG_const_type →
                        DW_TAG_base_type(unsigned char)

volatile unsigned char * const restrict p;
    on the other hand, represents a restricted constant
    pointer to a volatile character. This is encoded as:
        DW_TAG_variable(p) →
            DW_TAG_restrict_type →
                DW_TAG_const_type →
                    DW_TAG_pointer_type →
                        DW_TAG_volatile_type →
                            DW_TAG_base_type(unsigned char)
```

5.4   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.

5.5   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 integer 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 14. 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 14. 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 value (see Section 2.18) is the size in bits of each element of the array.

The array type entry may have a DW_AT_byte_size attribute, whose value (see Section 2.18) is the total size in bytes of an instance of the array type.

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.

Other attributes especially applicable to arrays are DW_AT_allocated, DW_AT_associated and DW_AT_data_location, which are described in Section 5.14. For relevant examples, see also Appendix D.2.1.

5.6   Structure, Union, Class and Interface 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").

C++ and Java have the notion of "class", which is in some ways similar to a structure. A class may have "member functions" which are subroutines that are within the scope of a class or structure.

The C++ notion of structure is more general than in C, being equivalent to a class with minor differences. Accordingly, in the following discussion statements about C++ classes may be understood to apply to C++ structures as well.

5.6.1   General Structure, Union and Class 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.

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.

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 integer constant value is the number of bytes required to hold an instance of the structure, union, or class, including any padding bytes. Otherwise, the entry may have a DW_AT_byte_size attribute whose value (see Section 2.18) is the dynamic number of bytes required.

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.

If a structure, union or class entry represents the defining declaration of a structure, class or union member of another structure class or union, the entry has a DW_AT_specification attribute whose value is a reference to the debugging information entry representing the incomplete declaration, as described above.

Structure, union and class 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 an attribute for the name of structure, class or union they represent.

For C and C++, 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 "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 latter 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, function or type 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.

Consider a nested class whose definition occurs outside of the containing class definition, as in:

```
struct A {
    struct B;
};

struct A::B { … };
```
The two different structs can be described in different compilation units to facilitate DWARF space compression (see Appendix E).

5.6.2 General Interface Description

The Java language defines "interface" types. An interface in Java is similar to a C++ or Java class with only abstract methods and constant data members.

Interface types are represented by debugging information entries with the tag DW_TAG_interface_type.

An interface 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 members of an interface are represented by debugging information entries that are owned by the interface type entry and that appear in the same order as the corresponding declarations in the source program.

5.6.3 Derived or Extended Classes and Interfaces

In C++, a class may be "derived from" or be a "subclass of" another class. In Java, an interface may "extend" one or more other interfaces, and a class may "extend" another class and/or "implement" one or more interfaces. All of these relationships may be described using the following. Note that in Java, the distinction between extends and implements is implied by the entities at the two ends of the relationship.

A class type or interface type entry that describes a derived, extended or implementing class or interface owns debugging information entries describing each of the classes or interfaces it is derived from, extending or implementing, respectively, 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 class or interface from which the parent class or structure of the inheritance entry is derived, extended or implementing.

An inheritance entry for a class that derives from or extends another class also has a DW_AT_data_member_location attribute, whose value describes the location of the beginning of the inherited type relative to the beginning address of the derived class. If that value is a constant, it is the offset in bytes from the beginning of the class to the beginning of the inherited type. Otherwise, the value must be a location description. In this latter case, the beginning address of the derived class is pushed on the expression stack before the location expression is evaluated and the result of the evaluation is the location of the inherited type.

The interpretation of the value of this attribute for inherited types is the same as the interpretation for data members (see Section 5.6.6).

An inheritance entry may have a DW_AT_accessibility attribute. If no accessibility attribute is present, private access is assumed for an entry of a class and public access is assumed for an entry of an interface.

If the class referenced by the inheritance entry serves as a C++ virtual base class, the inheritance entry has a DW_AT_virtuality attribute.

For a C++ virtual base, the data member location attribute will usually consist of a non-trivial location expression.

5.6.4 Access Declarations

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 class or structure 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.

5.6.5 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.

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

A data member entry may have a DW_AT_accessibility attribute. If no accessibility attribute is present, private access is assumed for an entry of a class and public access is assumed for an entry of a structure, union, or interface.

For a data member of a structure, union or class, the corresponding member entry has a DW_AT_data_member_location attribute whose value describes the location of that member relative to the base address of the structure, union, or class that most closely encloses the member declaration. If that value is a constant, it is the offset in bytes from the beginning of the innermost enclosing structure, union or class to the beginning of the data member. Otherwise, the value must be a location description. In this latter case, the base address of the innermost enclosing structure, union, or class is pushed on the expression stack before the location expression is evaluated and the result of the evaluation is the location of the member.

The interpretation of a constant operand as a byte offset from the base of the containing structure, union or class optimizes the size of the DWARF representation for the predominant case of a data member at a fixed offset. Note that no DWARF location expression evaluation is involved.

For a location description, the implicit push on the DWARF expression stack of the base address of the containing construct is equivalent to execution of the DW_OP_push_object_address operation (see Section 2.4.1.3); DW_OP_push_object_address therefore is not needed at the beginning of a location expression for a data member. The result of the evaluation is a location-- either an address or the name of a register, not an offset to the member.

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

- A DW_AT_byte_size attribute whose value (see Section 2.18) 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.

- A DW_AT_bit_offset attribute whose value (see Section 2.18) is the number of bits to the left of the leftmost (most significant) bit of the bit field value.

- A DW_AT_bit_size attribute whose value (see Section 2.18) 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
```

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

5.6.7 Member Function Entries

A member function is represented 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).

A member function entry may have a DW_AT_accessibility attribute. If no accessibility attribute is present, private access is assumed for an entry of a class and public access is assumed for an entry of a structure, union or interface.

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. The address of an object of the enclosing type is pushed onto the expression stack before the location description is evaluated.

If a subroutine entry represents the defining declaration of a member function and that definition appears outside of the body of the enclosing class 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.

5.6.8 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 entry with the tag DW_TAG_class_type, DW_TAG_structure_type or DW_TAG_union_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 may have 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 template definition is represented by a debugging information entry with the tag DW_TAG_template_value_parameter. Each such entry may have 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.

5.6.9 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 an integer constant that may have one of the values given in Figure 15.

| DW_DSC_label |
| DW_DSC_range |
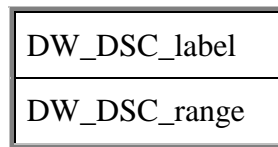
Figure 15. Discriminant descriptor values

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.

5.7 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. This entry also has a

DW_AT_byte_size attribute whose integer 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.

If the enumeration type occurs as the description of a dimension of an array type, and the stride for that dimension is different than what would otherwise be determined, then the enumeration type entry has a DW_AT_stride attribute, which specifies the number of bytes between successive elements along the dimension as described in Section 2.18.

5.8   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. The formal parameters of a parameter list (that have a specific type) are represented by a debugging information entry with the tag DW_TAG_formal_parameter. Each formal parameter entry has a DW_AT_type attribute that refers to the type of the formal parameter.

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

5.9   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. Note that "string" in this context refers to a target machine concept, not the class string as used in this document (except for the name attribute).

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 value (see Section 2.18) 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 value (see Section 2.18) is the length in bytes of the string.

5.10 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 value (see Section 2.18) is the size in bytes of an instance of the set type.

5.11 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, called the basis type, 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 value (see Section 2.18) is 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. The value of each of these attributes is determined as described in Section 2.18.

If the lower bound value is missing, the 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 lower bound values are currently defined.

If the upper bound and count are missing, then the upper bound value is *unknown*.

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 that attribute does not reference an object, the basis type is

the type of the upper bound or count attribute (if either of them references an object). If there is no upper bound or count attribute, or neither references 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.

If the subrange type occurs as the description of a dimension of an array type, and the stride for that dimension is different than what would otherwise be determined, then the subrange type entry has a DW_AT_stride attribute which specifies the signed number of bytes between successive elements along the dimension as described in Section 2.18.

Note that the stride can be negative.

5.12 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 to which the pointer to member entry points (the value is meaningless if the pointer to member does not currently point to anything).

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 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 is the value of the pointer to member object itself. The second value pushed is the base address of the entire structure or union instance containing the member whose address is being calculated.

For an expression such as

```
        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 given in the type of* `mbr_ptr`*.*

5.13 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 (see Section 2.18) is the size in bytes of an instance of this file type.

5.14 Dynamic Type Properties

5.14.1 Data Location

Some languages may represent objects using descriptors to hold information, including a location and/or run-time parameters, about the data that represents the value for that object.

The DW_AT_data_location attribute may be used with any type that provides one or more levels of hidden indirection and/or run-time parameters in its representation. Its value is a location expression. The result of evaluating this expression yields the address of the data for an object. When this attribute is omitted, the address of the data is the same as the address of the object.

This location expression will typically begin with DW_OP_push_object_address which loads the address of the object which can then serve as a descriptor in subsequent calculation. For an example using DW_AT_data_location for a Fortran 90 array, see Appendix D.2.1.

5.14.2 Allocation and Association Status

Some languages, such as Fortran 90, provide types whose values may be dynamically allocated or associated with a variable under explicit program control.

The DW_AT_allocated attribute may optionally be used with any type for which objects of the type can be explicitly allocated and deallocated. The presence of the attribute indicates that objects of the type are allocatable and deallocatable. The value of the attribute (see below) specifies whether an object of the type is currently allocated or not.

The DW_AT_associated attribute may optionally be used with any type for which objects of the type can be dynamically associated with other objects. The presence of the attribute indicates that objects of the type can be associated. The value of the attribute (see below) indicates whether an object of the type is currently associated or not.

While these attributes are defined specifically with Fortran 90 ALLOCATABLE and POINTER types in mind, usage is not limited to just that language.

The value of these attributes is determined as described in Section 2.18.

A non-zero value is interpreted as allocated or associated, and zero is interpreted as not allocated or not associated.

For Fortran 90, if both attributes are present, then the type should be assumed to have the POINTER property (and not ALLOCATABLE); the DW_AT_allocated attribute may then be used to indicate that the allocation status of the object resulted from execution of an ALLOCATE statement rather than pointer assignment.

For examples using DW_AT_allocated for Ada and Fortran 90 arrays, see Appendix D.2.

5.15 DWARF Procedures

A DWARF procedure is represented by any kind of debugging information entry that has a DW_AT_location attribute. If a suitable entry is not otherwise available, a DWARF procedure can be represented using a debugging information entry with the tag DW_TAG_dwarf_procedure.

A DWARF procedure is called by a DW_OP_call2, DW_OP_call4 or DW_OP_call_ref DWARF expression operator (see Section 2.4.1.5).

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.

In the descriptions of that follow, the following terms are used to specify the representation of DWARF sections:

- Initial length, section offset and section length, which are defined in Sections 7.2.2 and 7.4.

- Sbyte, ubyte, uhalf, and uword, which are defined in Section 7.26.

6.1 Accelerated Access

A debugger frequently needs to find the debugging information for a program entity defined outside of the compilation unit where the debugged program is currently stopped. Sometimes it will know only the name of the entity; sometimes only the address. To find the debugging information associated with a global entity 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.

Similarly, in languages in which the name of a type is guarenteed to always refer to the same concrete type (such as C++), a compiler may choose to elide type definitions in all compilation units except one. In this case a debugger needs a rapid way of locating the concrete type definition by name. As with the definition of global data objects, this would require a search of all the top level type definitions of all compilation units in a program.

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 entities (data objects, functions and types) by name or by address faster, a producer of DWARF information may provide three 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, two tables are maintained in separate object file sections called `.debug_pubnames` for objects and functions, and `.debug_pubtypes` for types. Each table consists of sets of variable length entries, each set describing the names of global objects and functions, or global types, respectively, whose definitions are represented by debugging information entries owned by a single compilation unit.

C++ member functions with a definition in the class declaration are definitions in every compilation unit containing the class declaration, but if there is no concrete instance there is no need to have a `.debug_pubnames` entry for the member function.

Each set begins with a header containing four values:

1. `unit_length` (initial length)

   The length of the entries for that set, not including the length field itself.

2. `version` (uhalf)

   A version number. This number is specific to the name lookup table and is independent of the DWARF version number.

3. `debug_info_offset` (section offset)

   The offset from the beginning of the `.debug_info` section of the compilation unit header referenced by the set.

4. `debug_info_length` (section length)

   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 section offset from the beginning of the compilation unit 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 an offset field containing zero (and no following string).

In the case of the name of a function member or static data 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 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. `unit_length` (initial length)

    The length of the entries for that set, not including the length field itself.

2. `version` (uhalf)

    A version number. This number is specific to the address lookup table and is independent of the DWARF version number.

3. `debug_info_offset` (section offset)

    The offset from the beginning of the `.debug_info` section of the compilation unit header referenced by the set.

4. `address_size` (ubyte)

    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. `segment_size` (ubyte)

    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, followed by the non-zero 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.

If the range of addresses covered by the text and/or data of a compilation unit is not contiguous, then there may be multiple address range descriptors for that compilation unit.

6.2   Line Number Information

A source-level debugger will need to know how to associate locations 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 locations. 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 line to line, or statement to statement.

As mentioned in Section 3.1.1, 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.

Some computer architectures employ more than one instruction set (for example, the ARM and MIPS architectures support a 32-bit as well as a 16-bit instruction set). Because the instruction set is a function of the program counter, it is convenient to encode the applicable instruction set in the `.debug_line` *section as well.*

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*

- *and so on*

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. Any deleted row would never be the beginning of a source statement. 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 line number 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.

When the set of addresses of a compilation unit cannot be described as a single contiguous range, there will be a separate matix for each contiguous subrange.

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.
line number 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).

6.2.2 State Machine Registers

The line number 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 a recommended breakpoint location. A recommended breakpoint location is intended to "represent" a line, a statement and/or a semantically distinct subpart 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.
prologue_end	A boolean indicating that the current address is one (of possibly many) where execution should be suspended for an entry breakpoint of a function.
epilogue_begin	A boolean indicating that the current address is one (of possibly many) where execution should be suspended for an exit breakpoint of a function.
isa	An unsigned integer whose value encodes the applicable instruction set architecture for the current instruction.
	The encoding of instruction sets should be shared by all users of a given architecture. It is recommended that this encoding be defined by the ABI authoring committee for each architecture.

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

address	0
file	1
line	1
column	0
is_stmt	determined by `default_is_stmt` in the line number program header
basic_block	"false"
end_sequence	"false"
prologue_end	"false"
epilogue_begin	"false"
isa	architecturally determined default

In the case of an architecture that supports one instruction set, the isa register will be set to some default initial value and thereafter will never change.

6.2.3 Line Number Program Instructions

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

special opcodes   These have a ubyte opcode field and no operands. Most of the instructions in a line number program are special opcodes.

standard opcodes   These have a ubyte opcode field which may be followed by zero or more LEB128 operands (except for DW_LNS_fixed_advance_pc, see below). The opcode implies the number of operands and their meanings, but the line number program header also specifies the number of operands 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 (which begins with a ubyte extended opcode).

6.2.4 The Line Number Program Header

The optimal encoding of line number information depends to a certain degree upon the architecture of the target machine. The line number program header provides information used by consumers in decoding the line number program instructions for a particular compilation unit and also provides information used throughout the rest of the line number program.

The line number program for each compilation unit begins with a header containing the following fields in order:

1. `unit_length` (initial length)

   The size in bytes of the line number information for this compilation unit, not including the `unit_length` field itself.

2. `version` (uhalf)

   A version number. This number is specific to the line number information and is independent of the DWARF version number.

3. `header_length`

   The number of bytes following the `header_length` field to the beginning of the first byte of the line number program itself. In the 32-bit DWARF format, this is a 4-byte unsigned length; in the 64-bit DWARF format, this field is an 8-byte unsigned length (see Section 7.4).

4. `minimum_instruction_length` (ubyte)

   The size in bytes of the smallest target machine instruction. Line number 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 approach to building line number information when machine instructions are emitted in an order corresponding to the source program is to set *default_is_stmt* to "true" and to not change the value of the *is_stmt* register within the line number program. One matrix entry is produced for each line that has code generated for it. The effect is that every entry in the matrix recommends the beginning of each represented line as a breakpoint location. This is the traditional practice for unoptimized code.

   A more sophisticated approach might involve multiple entries in the matrix for a line number; in this case, at least one entry (often but not necessarily only one) specifies a recommended breakpoint location for the line number. DW_LNS_negate_stmt opcodes in the line number program control which matrix entries constitute such a recommendation and *default_is_stmt* might be either "true" or "false". This approach might be used as part of support for debugging optimized code.

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.

    This number may be larger or smaller than the number of standard opcodes defined for the specified version of the line number information (12 in DWARF V3, 9 in DWARF V2). If smaller, then the higher numbered opcodes are not used in the line number table of this unit (and the codes are treated as special opcodes). If greater, then the numbers between that of the highest standard opcode and the first special opcode (not inclusive) are necessarily code for vendor specific extensions.

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.

    Codes for vendor specific extensions, if any, are described just like standard opcodes.

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 line number information for this compilation unit or is used in other contexts, such as in a declaration coordinate or a macro file inclusion. Each entry consists of the following values:

    *   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 (implemntation-defined) time of last modification for the file.

- 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 line number 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 DW_LNE_define_file.

6.2.5 The Line Number Program

As stated before, the goal of a line number 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 or other optimization.)

6.2.5.1 Special Opcodes

Each ubyte 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 line number program header 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."

5. Set the `prologue_end` register to "false."

6. Set the `epilogue_begin` register to "false."

All of the special opcodes do those same six 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 line number program uses several parameters in the header to configure the instruction set. There are two reasons for this. First, although the opcode space available for special opcodes now ranges from 13 through 255, the lower bound may increase if one adds new standard opcodes. Thus, the `opcode_base` *field of the line number program header 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 line number program header 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 header, 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 header. 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 to give the adjusted opcode. The amount to increment the `address` register is the result of the adjusted opcode divided by the `line_range` multiplied by the `minimum_instruction_length` field from the header. That is,

```
address increment =
    (adjusted opcode / line_range) * minimim_instruction_length
```

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 12 standard ubyte opcodes. In the future, additional opcodes may be defined by setting the `opcode_base` field in the line number program header to a value greater than 13. The applicable operands and the actions performed by these opcodes are as follows:

1. **DW_LNS_copy**

   The DW_LNS_copy opcode takes no operands. It appends a row to the matrix using the current values of the state-machine registers. Then it sets the `basic_block`, `prologue_end` and `epilogue_begin` registers to "false."

2. **DW_LNS_advance_pc**

   The DW_LNS_advance_pc opcode takes a single unsigned LEB128 operand, multiplies it by the `minimum_instruction_length` field of the header, and adds the result to the `address` register of the state machine.

3. **DW_LNS_advance_line**

   The DW_LNS_advance_line opcode takes a single signed LEB128 operand and adds that value to the `line` register of the state machine.

4. **DW_LNS_set_file**

   The DW_LNS_set_file opcode takes a single unsigned LEB128 operand and stores it in the `file` register of the state machine.

5. **DW_LNS_set_column**

   The DW_LNS_set_column opcode takes a single unsigned LEB128 operand and stores it in the `column` register of the state machine.

6. **DW_LNS_negate_stmt**

   The DW_LNS_negate_stmt opcode takes no operands. It sets the `is_stmt` register of the state machine to the logical negation of its current value.

7. **DW_LNS_set_basic_block**

   The DW_LNS_set_basic_block opcode takes no operands. It sets the `basic_block` register of the state machine to "true."

8. **DW_LNS_const_add_pc**

   The DW_LNS_const_add_pc opcode takes no operands. It multiplies the address increment value corresponding to special opcode 255 by the `minimum_instruction_length` field of the header, and adds the result to the `address` register of the state machine.

   When the line number 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**

   The DW_LNS_fixed_advance_pc opcode takes a single uhalf (unencoded) operand and adds it to the `address` register of the state machine. This is the only standard opcode whose operand is <u>not</u> a variable length number. It also does <u>not</u> multiply the operand by the `minimum_instruction_length` field of the header.

   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.

10. **DW_LNS_set_prologue_end**

    The DW_LNS_set_prologue_end opcode takes no operands. It sets the `prologue_end` register to "true".

    When a breakpoint is set on entry to a function, it is generally desirable for execution to be suspended, not on the very first instruction of the function, but rather at a point after the function's frame has been set up, after any language defined local declaration processing has been completed, and before execution of the first statement of the function begins. Debuggers generally cannot properly determine where this point is. This command allows a compiler to communicate the location(s) to use.

    In the case of optimized code, there may be more than one such location; for example, the code might test for a special case and make a fast exit prior to setting up the frame.

Note that the function to which the prologue end applies cannot be directly determined from the line number information alone; it must be determined in combination with the subroutine information entries of the compilation (including inlined subroutines).

11. DW_LNS_set_epilogue_begin

The DW_LNS_set_epilogue_begin opcode takes no operands. It sets the `epilogue_begin` register to "true".

When a breakpoint is set on the exit of a function or execution steps over the last executable statement of a function, it is generally desirable to suspend execution after completion of the last statement but prior to tearing down the frame (so that local variables can still be examined). Debuggers generally cannot properly determine where this point is. This command allows a compiler to communicate the location(s) to use.

Note that the function to which the epilogue end applies cannot be directly determined from the line number information alone; it must be determined in combination with the subroutine information entries of the compilation (including inlined subroutines).

In the case of a trivial function, both prologue end and epilogue begin may occur at the same address.

12. DW_LNS_set_isa

The DW_LNS_set_isa opcode takes a single unsigned LEB128 operand and stores that value in the `isa` register of the state machine.

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

The DW_LINE_end_sequence opcode takes no operands. It sets the `end_sequence` register of the state machine to "true" and appends a row to the matrix using the current values of the state-machine registers. Then it resets the registers to the initial values specified above (see Section 6.2.2). Every line number 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**

   The DW_LNE_set_address opcode 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. It sets the `address` register to the value given by the relocatable address.

   All of the other line number 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**

   The DW_LNE_define_file opcode takes four operands:

   1. A null-terminated string containing a source file name.

   2. An unsigned LEB128 number representing the directory index of the directory in which the file was found.

   3. An unsigned LEB128 number representing the time of last modification of the file.

   4. 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 line number program header. 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 header come before names defined by the DW_LNE_define_file instruction. These numbers are used in the `file` register of the state machine.

Appendix D.4 gives some sample line number 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 undefinition.

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.

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 that 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 are, however, separated by commas.) Also, exactly one space character should separate the right parenthesis that 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 line number information table for the relevant compilation unit. This index indicates (indirectly) the name of the file that 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 designate a line number information file name entry consisting of a null string.

A DW_MACINFO_start_file entry representing the base source file should encode the value 0 in its line number operand.

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. Typically, the CFA is defined to be the value of the stack pointer at the call site in the previous frame (which may be different from its value on entry to the current frame).*

- *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 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 rule which computes the CFA value; it may be either a register and a signed offset that are added together or a DWARF expression that is evaluated.

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.

expression(E)   The previous value of this register is located at the address produced by executing the DWARF expression E.

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).

If the range of code addresses for a function is not contiguous, there may be multiple CIEs and FDEs corresponding to the parts of that function.

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

1. `length` (initial length)

   A constant that gives the number of bytes of the CIE structure, not including the `length` field itself (where `length mod` *`addressing_unit_size`* `== 0`).

2. `CIE_id` (see Section 7.4)

   A constant that is used to distinguish CIEs from FDEs. In the 32-bit DWARF format, this is a 4-byte unsigned integer; in the 64-bit DWARF format, this is an 8-byte unsigned integer.

3. `version` (ubyte)

   A version number. This number is specific to the call frame information and is independent of the DWARF version number.

4. `augmentation` (string)

   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` (unsigned LEB128)

   A constant that is factored out of all advance location instructions (see below).

6. `data_alignment_factor` (signed LEB128)

   A constant that is factored out of all offset instructions (see below.)

7. `return_address_register`

   An unsigned LEB128 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.

   In DWARF V2, this field is a ubyte.

8. `initial_instructions` (array of ubyte)

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

9. `padding` (array of ubyte)

   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` (initial length)

   A constant that gives the number of bytes of the header and instruction stream for this function, not including the `length` field itself (where `length` mod *addressing_unit_size* == 0).

2. `CIE_pointer` (section offset)

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

3. `initial_location` (target address)

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

4. `address_range` (target address)

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

5. `instructions` (array of ubyte)

   A sequence of table defining instructions that are described below.

6. `padding` (array of ubyte)

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

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**

   The DW_CFA_advance instruction takes a single operand (encoded with the opcode) 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 the value of `delta * code_alignment_factor`. All other values in the new row are initially identical to the current row.

2. **DW_CFA_offset**

   The DW_CFA_offset instruction takes two operands: a register number (encoded with the opcode) and an unsigned LEB128 constant representing a factored offset. 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**

   The DW_CFA_restore instruction takes a single operand (encoded with the opcode) 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**

   The DW_CFA_set_loc instruction takes a single operand 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**

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

6. **DW_CFA_advance_loc2**

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

7. **DW_CFA_advance_loc4**

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

8. **DW_CFA_offset_extended**

   The DW_CFA_offset_extended instruction takes two unsigned LEB128 operands 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 operand.

9. **DW_CFA_restore_extended**

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

10. **DW_CFA_undefined**

    The DW_CFA_undefined instruction takes a single unsigned LEB128 operand that represents a register number. The required action is to set the rule for the specified register to "undefined."

11. **DW_CFA_same_value**

    The DW_CFA_same_value instruction takes a single unsigned LEB128 operand that represents a register number. The required action is to set the rule for the specified register to "same value."

12. **DW_CFA_register**

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

The next two instructions provide the ability to stack and retrieve complete register states. They may be useful, for example, for a compiler that moves epilogue code into the body of a function.

13. DW_CFA_remember_state

The DW_CFA_remember_state instruction takes no operands. The required action is to push the set of rules for every register onto an implicit stack.

14. DW_CFA_restore_state

The DW_CFA_restore_state instruction takes no operands. The required action is to pop the set of rules off the implicit stack and place them in the current row.

15. DW_CFA_def_cfa

The DW_CFA_def_cfa instruction takes two unsigned LEB128 operands representing a register number and a (non-factored) offset. The required action is to define the current CFA rule to use the provided register and offset.

16. DW_CFA_def_cfa_register

The DW_CFA_def_cfa_register instruction takes a single unsigned LEB128 operand 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

The DW_CFA_def_cfa_offset instruction takes a single unsigned LEB128 operand representing a (non-factored) 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

The DW_CFA_nop instruction has no operands and no required actions. It is used as padding to make a CIE or FDE an appropriate size.

19. DW_CFA_def_cfa_expression

The DW_CFA_def_cfa_expression instruction takes a single operand encoded as a DW_FORM_block value representing a DWARF expression. The required action is to establish that expression as the means by which the current CFA is computed.

The DW_OP_call2, DW_OP_call4, and DW_OP_call_ref DWARF operators cannot be used in such a DWARF expression.

20. DW_CFA_expression

The DW_CFA_expression instruction takes two operands: an unsigned LEB128 value representing a register number, and a DW_FORM_block value representing a DWARF expression. The required action is to establish the DWARF expression as the means by which the address in which the given register contents are found may be computed. The value of the CFA is pushed on the DWARF evaluation stack prior to execution of the DWARF expression.

The DW_OP_call2, DW_OP_call4, DW_OP_call_ref and DW_OP_push_object_address DWARF operators (see Section 2.4.1) cannot be used in such a DWARF expression.

DW_OP_call2, DW_OP_call4 and DW_OP_call_ref operators are not meaningful in a DW_CFA_def_cfa_expression or DW_CFA_expression operand because there is no mapping from call frame information to any corresponding debugging compilation unit information, thus there is no way to interpret the call offset.

DW_OP_push_object_address is also not meaningful in a DW_CFA_def_cfa_expression or DW_CFA_expression operand because there is no object context to provide a value to push.

21. DW_CFA_offset_extended_sf

The DW_CFA_offset_extended_sf instruction takes two operands: an unsigned LEB128 value representing a register number and a signed LEB128 factored offset. This instruction is identical to DW_CFA_offset_extended except that the second operand is signed.

22. DW_CFA_def_cfa_sf

The DW_CFA_def_cfa_sf instruction takes two operands: an unsigned LEB128 value representing a register number and a signed LEB128 factored offset. This instruction is identical to DW_CFA_def_cfa except that the second operand is signed and factored.

23. DW_CFA_def_cfa_offset_sf

The DW_CFA_def_cfa_offset_sf instruction takes a signed LEB128 operand representing a factored offset. This instruction is identical to DW_CFA_def_cfa_offset except that the operand is signed and factored.

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 D.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, DW_LNE, DW_CC 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.

There may also be codes for vendor specific extensions between the number of standard line number opcodes and the first special line number opcode. However, since the number of standard opcodes varies with the DWARF version, the range for extensions is also version dependent. Thus, DW_LNS_lo_user and DW_LNS_hi_user symbols are not defined.

Vendor defined tags, attributes, base type encodings, location atoms, language names, line number actions, 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 Values

7.2.1 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, line number 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.

7.2.2 Initial Length Values

An initial length field is one of the length fields that occur at the beginning of those DWARF sections that have a header (`.debug_aranges`, `.debug_info`, `.debug_line`, `.debug_pubnames`, and `.debug_pubtypes`) or the length field that occurs at the beginning of the CIE and FDE structures in the `.debug_frame` section.

In an initial length field, the values `0xfffffff0` through `0xffffffff` are reserved by DWARF to indicate some form of extension relative to DWARF V2; such values must not be interpreted as a length field. The use of one such value, `0xffffffff`, is defined below (see Section 7.4); the use of the other values is reserved for possible future extensions.

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   32-Bit and 64-Bit DWARF Formats

DWARF Version 2 provides the ability to describe programs that operate in a 64-bit address space environment. However, it does not make provision for a DWARF description that itself requires more than 32-bits to represent; this is because the lengths that describe DWARF sections and values that are offsets into DWARF sections are all specified to be 32-bits in size. This revision adds the ability to support a DWARF description that is greater than 4 GBytes in size.

DWARF Version 3 defines two closely related file formats. In the 32-bit DWARF format, all values that represent lengths of DWARF sections and offsets relative to the beginning of DWARF sections are represented using 32-bits (this is the same as defined in DWARF Version 2). In the 64-bit DWARF format, all values that represent lengths of DWARF sections and offsets relative to the beginning of DWARF sections are represented using 64-bits. A special convention applies to the initial length field of certain DWARF sections, as well as the CIE and FDE structures, so that the 32-bit and 64-bit DWARF formats can coexist and be distinguished within a single linked object.

The differences between the 32- and 64-bit DWARF formats are detailed in the following:

1. In the 32-bit DWARF format, an initial length field (see Section 7.2.2) is an unsigned 32-bit integer (which must be less than `0xfffffff00`); in the 64-bit DWARF format, an initial length field is 96 bits in size, and has two parts:

    *   The first 32-bits have the value `0xffffffff`.

    *   The following 64-bits contain the actual length represented as an unsigned 64-bit integer.

    This representation allows a DWARF consumer to dynamically detect that a DWARF section contribution is using the 64-bit format and to adapt its processing accordingly.

2. Section offset and section length fields that occur in the headers of DWARF sections (other than initial length fields) are listed following. In the 32-bit DWARF format these are 32-bit unsigned integer values; in the 64-bit DWARF format, they are 64-bit unsigned integer values.

Section	Name	Role
`.debug_aranges`	`debug_info_offset`	offset in `.debug_info`
`.debug_frame/`CIE	`CIE_id`	CIE distinguished value
`.debug_frame/`FDE	`CIE_pointer`	offset in `.debug_frame`
`.debug_info`	`debug_abbrev_offset`	offset in `.debug_abbrev`
`.debug_line`	`header_length`	length of header itself
`.debug_pubnames`	`debug_info_offset`	offset in `.debug_info`
	`debug_info_length`	length of `.debug_info` contribution
`.debug_pubtypes`	`debug_info_offset`	offset in `.debug_info`
	`debug_info_length`	length of `.debug_info` contribution

The CIE_id field in a CIE structure must be 64 bits because it overlays the CIE_pointer in a FDE structure; this implicit union must be accessed to distinguish whether a CIE or FDE is present, consequently, these two fields must exactly overlay each other (both offset and size).

3. Within the body of the `.debug_info` section, certain forms of attribute value depend on the choice of DWARF format as follows. For the 32-bit DWARF format, the value is a 32-bit unsigned integer; for the 64-bit DWARF format, the value is a 64-bit unsigned integer.

Form	Role
DW_FORM_ref_addr	offset in `.debug_info`
DW_FORM_strp	offset in `.debug_str`

4. Within the body of the `.debug_info` section, certain classes of attribute value use an attribute value form that depends on the DWARF format at follows: in the 32-bit DWARF format, the lineptr, loclistptr, macptr and rangelistptr classes use form DW_FORM_data4; in the 64-bit DWARF format, they use form DW_FORM_data8.

5. Within the body of the `.debug_pubnames` and `.debug_pubtypes` sections, the representation of the first field of each tuple (which represents an offset in the `.debug_info` section) depends on the DWARF format as follows: in the 32-bit DWARF format, this field is a 32-bit unsigned integer; in the 64-bit DWARF format, it is a 64-bit unsigned integer.

The 32-bit and 64-bit DWARF format conventions must <u>not</u> be intermixed within a single compilation unit.

Attribute values and section header fields that represent addresses in the target program are not affected by these rules.

A DWARF consumer that supports the 64-bit DWARF format must support executables in which some compilation units use the 32-bit format and others use the 64-bit format provided that the combination links correctly (that is, provided that there are no link-time errors due to truncation or overflow). (An implementation is not required to guarantee detection and reporting of all such errors.)

It is expected that DWARF producing compilers will <u>not</u> use the 64-bit format <u>by default</u>. In most cases, the division of even very large applications into a number of executable and shared objects will suffice to assure that the DWARF sections within each individual linked object are less than 4 GBytes in size. However, for those cases where needed, the 64-format allows the unusual case to be handled as well. Even in this case, it is expected that only application supplied objects will need be compiled using the 64-bit format; separate 32-bit format versions of system supplied shared executable libraries can still be used.

7.5   Format of Debugging Information

For each compilation unit compiled with a DWARF Version 3 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 and Version 3 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, it is possible 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. `unit_length` (initial length)

   A 4-byte or 12-byte unsigned integer representing the length of the `.debug_info` contribution for that compilation unit, not including the length field itself. In the 32-bit DWARF format, this is a 4-byte unsigned integer (which must be less than `0xffffff00)`; in the 64-bit DWARF format, this consists of the 4-byte value `0xffffffff` followed by an 8-byte unsigned integer that gives the actual length (see Section 7.4).

2. `version` (uhalf)

   A 2-byte unsigned integer representing the version of the DWARF information for the compilation unit. For DWARF Version 3, the value in this field is 3.

3. `debug_abbrev_offset` (section offset)

   A 4-byte or 8-byte unsigned offset into the `.debug_abbrev` section. This offset associates the compilation unit with a particular set of debugging information entry abbreviations. In the 32-bit DWARF format, this is a 4-byte unsigned length; in the 64-bit DWARF format, this is an 8-byte unsigned length (see Section 7.4).

4. `address_size` (ubyte)

   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 abbreviations 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 abbreviation code 0 are considered null entries.

7.5.3 Abbreviations Tables

The abbreviations 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 abbreviations table.

The abbreviations 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 Figure 16.

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 that 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 that entry. (Either the first child or sibling entries may be null entries). The encodings for the child determination byte are given in Figure 17. (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 abbreviations table.

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

See Appendix D.1 for a depiction of the organization of the debugging information.

7.5.4 Attribute Encodings

The encodings for the attribute names are given in Figure 18.

The attribute form governs how the value of the attribute is encoded. There are nine classes of form, listed below. Each class is a set of forms which have related representations and which are given a common interpretation according to the attribute in which the form is used.

Some encodings are members of more than one class; in that case, the list of classes allowed by the applicable attribute in Figure 18 determines the class of the form. DW_FORM_data4 and DW_FORM_data8 may be members of classes constant, lineptr, loclistptr, macptr and rangelistptr. They are members of the class constant if used for the value of an attribute that allows class constant but not class lineptr, loclistptr, macptr or rangelistptr. They are members of the class lineptr, loclistptr, macptr or rangelistptr if used for the value of an attribute that allows one of those classes.

In DWARF V2, each form belonged to exactly one class so that the classes formed a partition of the set of forms. Moreover, the class constant was used for some "pointers" (described as offsets) into other sections. In practice such offsets had to use DW_FORM_data4 since they had to be a form that could be relocated during linking (making such "constants" not very constant).

This revision introduces new classes to more clearly identify when a value is to be interpreted as a pointer or offset into another section. The form DW_FORM_data8 is included in these classes to accommodate the 64-bit DWARF format. (It would have been cleaner to introduce new forms to be used for such pointers by analogy with the DW_FORM_strp form, but this would not be upward compatible.) Since classes as such are not represented in the DWARF format (only the individual forms are encoded), the introduction of new or revised class descriptions does not affect upward compatibility.

The forms DW_FORM_data4 and DW_FORM_data8 continue to be usable as constants when this does not conflict with their possible role as pointers. Even without them, there are no limitations in the value of constants that can be represented using the other forms in class constant (consider especially DW_FORM_sdata and DW_FORM_udata); the block forms may also be appropriate.

Each possible form belongs to one or more of the following classes:

address

> Represented as an object of appropriate size to hold an address on the target machine (DW_FORM_addr). The size is encoded in the compilation unit header (see Section 7.5.1). 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:

>> A 1-byte length followed by 0 to 255 contiguous information bytes (DW_FORM_block1).

>> A 2-byte length followed by 0 to 65,535 contiguous information bytes (DW_FORM_block2).

>> A 4-byte length followed by 0 to 4,294,967,295 contiguous information bytes (DW_FORM_block4).

>> 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. There are fixed length constant data forms for one, two, four and eight byte values (respectively, DW_FORM_data1, DW_FORM_data2, DW_FORM_data4, and DW_FORM_data8). There are also variable 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. Note that DW_FORM_data4 and DW_FORM_data8 are members of class constant only if the attribute in question does not allow one of the classes lineptr, loclistptr, macptr or rangelistptr (see below).

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.

lineptr

> This is an offset into the `.debug_line` section. It consists of a 4- or 8-byte value which is the offset from the beginning of the `.debug_line` section to the first byte of the data making up the line number list for the compilation unit. It is relocatable in a relocatable object file, and relocated in an executable or shared object. It is either form DW_FORM_data4 or form DW_FORM_data8.

loclistptr

> This is an offset into the `.debug_loc` section. It consists of a 4- or 8-byte value which is the offset from the beginning of the `.debug_loc` section to the first byte of the data making up the location list for the compilation unit. It is relocatable in a relocatable object file, and relocated in an executable or shared object. It is either form DW_FORM_data4 or form DW_FORM_data8.

macptr

> This is an offset into the `.debug_macinfo` section. It consists of a 4- or 8-byte value which is the offset from the beginning of the `.debug_macinfo` section to the first byte of the data making up the macro information list for the compilation unit. It is relocatable in a relocatable object file, and relocated in an executable or shared object. It is either form DW_FORM_data4 or form DW_FORM_data8.

rangelistptr

> This is an offset into the `.debug_ranges` section. It consists of a 4- or 8-byte value which is the offset from the beginning of the `.debug_ranges` section to the beginning of the non-contiguous address ranges information for the referencing entity. It is relocatable in a relocatable object file and relocated in an executable or shared object. It is either form DW_FORM_data4 or form DW_FORM_data8.

Because classes lineptr, loclistptr, macptr and rangelistptr share a common representation, it is not possible for an attribute to allow more than one of these classes. If an attribute allows both class constant and one of lineptr, loclistptr, macptr or rangelistptr, then DW_FORM_data4 and DW_FORM_data8 are interpreted as members of the latter as appropriate (not class constant).

reference

> There are two types of reference.

> The first type of reference can identify any debugging information entry within the containing unit. This type of reference is an offset from the first byte of the compilation header for the compilation unit containing the reference. There are five forms for this type of reference. There are fixed length forms for one, two, four and eight byte offsets (respectively, DW_FORM_ref1, DW_FORM_ref2, DW_FORM_ref4, and DW_FORM_ref8). There is also an unsigned variable length offset encoded form that uses LEB128 numbers (DW_FORM_ref_udata). Because this type of reference is within the containing compilation unit no relocation of the value is required.

> The second type of reference can identify any debugging information entry within a program; in particular, it may refer to an entry in a different compilation unit from the unit containing the reference, and may refer to an entry in a different shared object. This type of reference (DW_FORM_ref_addr) is an offset from the beginning of the `.debug_info` section of the target executable or shared object; it is relocatable in a relocatable object file and frequently relocated in an executable file or shared object. For references from one shared object or static executable file to another, the relocation and identification of the target object must be performed by the consumer. In the 32-bit DWARF format, this offset is a 4-byte unsigned value; in the 64-bit DWARF format, it is an 8-byte unsigned value (see Section 7.4).

> *A debugging information entry that may be referenced by another compilation must have a global symbolic name.*

For a reference from one executable or shared object to another, the reference is resolved by the debugger to identify the shared object or executable and the offset into that object's `.debug_info` section in the same fashion as the run time loader, either when the debug information is first read, or when the reference is used.

The use of compilation unit relative references will reduce the number of link-time relocations and so speed up linking. The use of the second type of 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 an offset into a string table contained in the `.debug_str` section of the object file (DW_FORM_strp). In the 32-bit DWARF format, the representation of a DW_FORM_strp value is an 4-byte unsigned offset; in the 64-bit DWARF format, it is an 8-byte unsigned offset (see Section 7.4).

If the DW_AT_use_UTF8 attribute is specified for the compilation unit entry, string values are encoded using the UTF-8 (Unicode Transformation Format-8) from the Universal Character Set standard (ISO/IEC 10646-1:1993). Otherwise, the string representation is unspecified.

The Unicode Standard V3 is fully compatible with ISO/IEC 10646-1:1993. It contains all the same characters and encoding points as ISO/IEC 10646, as well as additional information about the characters and their use.

DWARF Version 2 did not specify the representation of strings; for upward compatibility, this version also does not. However, the UTF-8 representation is strongly recommended.

The form encodings are listed in Figure 19.

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

Tag name	Value
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
DW_TAG_friend	0x2a
DW_TAG_namelist	0x2b
DW_TAG_namelist_item	0x2c
DW_TAG_packed_type	0x2d
DW_TAG_subprogram	0x2e

Tag name	Value
DW_TAG_template_type_parameter	0x2f
DW_TAG_template_value_parameter	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_dwarf_procedure ‡	0x36
DW_TAG_restrict_type ‡	0x37
DW_TAG_interface_type ‡	0x38
DW_TAG_namespace ‡	0x39
DW_TAG_imported_module ‡	0x3a
DW_TAG_unspecified_type ‡	0x3b
DW_TAG_partial_unit ‡	0x3c
DW_TAG_imported_unit ‡	0x3d
DW_TAG_mutable_type ‡	0x3e
DW_TAG_lo_user	0x4080
DW_TAG_hi_user	0xffff

‡ New in DWARF V3

Figure 16. Tag encodings

Child determination name	Value
DW_CHILDREN_no	0
DW_CHILDREN_yes	1

Figure 17. Child determination encodings

Attribute name	Value	Classes
DW_AT_sibling	0x01	reference
DW_AT_location	0x02	block, loclistptr
DW_AT_name	0x03	string
DW_AT_ordering	0x09	constant
DW_AT_byte_size	0x0b	block, constant, reference
DW_AT_bit_offset	0x0c	block, constant, reference
DW_AT_bit_size	0x0d	block, constant, reference
DW_AT_stmt_list	0x10	lineptr
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, loclistptr
DW_AT_common_reference	0x1a	reference
DW_AT_comp_dir	0x1b	string
DW_AT_const_value	0x1c	block, constant, string
DW_AT_containing_type	0x1d	reference

Attribute name	Value	Classes
DW_AT_default_value	0x1e	reference
DW_AT_inline	0x20	constant
DW_AT_is_optional	0x21	flag
DW_AT_lower_bound	0x22	block, constant, reference
DW_AT_producer	0x25	string
DW_AT_prototyped	0x27	flag
DW_AT_return_addr	0x2a	block, loclistptr
DW_AT_start_scope	0x2c	constant
DW_AT_stride_size	0x2e	constant
DW_AT_upper_bound	0x2f	block, constant, reference
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	block, constant, reference
DW_AT_data_member_location	0x38	block, constant, loclistptr
DW_AT_decl_column	0x39	constant
DW_AT_decl_file	0x3a	constant

Attribute name	Value	Classes
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, loclistptr
DW_AT_friend	0x41	reference
DW_AT_identifier_case	0x42	constant
DW_AT_macro_info	0x43	macptr
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, loclistptr
DW_AT_type	0x49	reference
DW_AT_use_location	0x4a	block, loclistptr
DW_AT_variable_parameter	0x4b	flag
DW_AT_virtuality	0x4c	constant
DW_AT_vtable_elem_location	0x4d	block, loclistptr
DW_AT_allocated ‡	0x4e	block, constant, reference

Attribute name	Value	Classes
DW_AT_associated ‡	0x4f	block, constant, reference
DW_AT_data_location ‡	0x50	block
DW_AT_stride ‡	0x51	block, constant, reference
DW_AT_entry_pc ‡	0x52	address
DW_AT_use_UTF8 ‡	0x53	flag
DW_AT_extension ‡	0x54	reference
DW_AT_ranges ‡	0x55	rangelistptr
DW_AT_trampoline ‡	0x56	address, flag, reference, string
DW_AT_call_column ‡	0x57	constant
DW_AT_call_file ‡	0x58	constant
DW_AT_call_line ‡	0x59	constant
DW_AT_description ‡	0x5a	string
DW_AT_lo_user	0x2000	---
DW_AT_hi_user	0x3fff	---

‡ New in DWARF V3

Figure 18. Attribute encodings

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, lineptr, loclistptr, macptr, rangelistptr
DW_FORM_data8	0x07	constant, lineptr, loclistptr, macptr, rangelistptr
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

Form name	Value	Class
DW_FORM_ref_udata	0x15	reference
DW_FORM_indirect	0x16	(see Section 7.5.3)

Figure 19. Attribute form encodings

7.6 Variable Length Data

The special constant data forms DW_FORM_sdata and DW_FORM_udata are encoded using "Little Endian Base 128" (LEB128) numbers. LEB128 is a scheme for encoding integers densely that exploits the assumption that most integers are small in magnitude. (This encoding is equally suitable whether the target machine architecture represents data in big-endian or little- endian order. It is "little endian" only in the sense that it avoids using space to represent the "big" end of an unsigned integer, when the big end is all zeroes or sign extension bits).

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

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

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

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

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

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

Figure 21 gives some examples of *DW_FORM_sdata* numbers .

Appendix C gives algorithms for encoding and decoding these forms.

7.7 DWARF Expressions and Location Descriptions

7.7.1 DWARF Expressions

A DWARF 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 DWARF expression are concatenated from left to right. The encodings for the operations in a DWARF expression are described in Figure 22.

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	

Operation	Code	No. of Operands	Notes
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	

Operation	Code	No. of Operands	Notes
DW_OP_shra	0x26	0	
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	

Operation	Code	No. of Operands	Notes
DW_OP_breg0	0x70	1	SLEB128 offset
DW_OP_breg1	0x71	1	base register 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_push_object_address ‡	0x97	0	
DW_OP_call2 ‡	0x98	1	2-byte offset of DIE
DW_OP_call4 ‡	0x99	1	4-byte offset of DIE
DW_OP_call_ref ‡	0x9a	1	4- or 8-byte offset of DIE
DW_OP_lo_user	0xe0		
DW_OP_hi_user	0xff		

‡ New in DWARF V3

Figure 22. DWARF operation encodings

7.7.2 Location Expressions

A location expression is a DWARF expression that is used to compute the location (possibly a register) of a variable or other entity.

7.7.3 Location Lists

Each entry in a location list is either a location list entry, a base address selection entry, or an end of list entry.

A location list entry 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.

A base address selection entry and an end of list entry each consist of two (constant or relocated) addresses. The two addresses are the same size as used by DW_FORM_addr on the target machine.

For a location list to be specified, the base address of the corresponding compilation unit must be defined (see Section 3.1).

7.8 Base Type Encodings

The values of the constants used in the DW_AT_encoding attribute are given in Figure 23.

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

Base type encoding name	Value
DW_ATE_unsigned_char	0x8
DW_ATE_imaginary_float ‡	0x9
DW_ATE_lo_user	0x80
DW_ATE_hi_user	0xff

‡ New in DWARF V3

Figure 23. Base type encoding values

7.9 Accessibility Codes

The encodings of the constants used in the DW_AT_accessibility attribute are given in Figure 24.

Accessibility code name	Value
DW_ACCESS_public	1
DW_ACCESS_protected	2
DW_ACCESS_private	3

Figure 24. Accessibility encodings

7.10 Visibility Codes

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

Visibility code name	Value
DW_VIS_local	1
DW_VIS_exported	2
DW_VIS_qualified	3

Figure 25. Visibility encodings

7.11 Virtuality Codes

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

Virtuality code name	Value
DW_VIRTUALITY_none	0
DW_VIRTUALITY_virtual	1
DW_VIRTUALITY_pure_virtual	2

Figure 26. Virtuality encodings

7.12 Source Languages

The encodings for source languages are given in Figure 27. Names marked with † and their associated values are reserved, but the languages they represent are not well supported in DWARF Version 3.

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_Java †‡	0x000b
DW_LANG_C99 ‡	0x000c
DW_LANG_Ada95 †‡	0x000d
DW_LANG_Fortran95 ‡	0x000e
DW_LANG_PLI †	0x000f
DW_LANG_lo_user	0x8000
DW_LANG_hi_user	0xffff

† See text ‡ New in DWARF V3

Figure 27. Language encodings

7.13 Address Class Encodings

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

7.14 Identifier Case

The encodings of the constants used in the DW_AT_identifier_case attribute are given in Figure 28.

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 28. Identifier case encodings

7.15 Calling Convention Encodings

The encodings for the values of the DW_AT_calling_convention attribute are given in Figure 29.

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 29. Calling convention encodings

7.16 Inline Codes

The encodings of the constants used in the DW_AT_inline attribute are given in Figure 30.

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 30. Inline encodings

7.17 Array Ordering

The encodings for the values of the ordering attribute of arrays used in the DW_AT_ordering attribute are given in Figure 31.

Ordering name	Value
DW_ORD_row_major	0
DW_ORD_col_major	1

Figure 31. Ordering encodings

7.18 Discriminant Lists

The descriptors used in the DW_AT_discr_list attribute are encoded as 1-byte constants. The defined values are given in Figure 32.

Descriptor Name	Value
DW_DSC_label	0
DW_DSC_range	1

Figure 32. Discriminant descriptor encodings

7.19 Name Lookup Tables

Each set of entries in the table of global names contained in the `.debug_pubnames` and `.debug_pubtypes` sections begins with a header consisting of:

1. A 4-byte or 12-byte length of the set of entries for this compilation unit, not including the length field itself. In the 32-bit DWARF format, this is a 4-byte unsigned integer (which must be less than `0xffffff00`); in the 64-bit DWARF format, this consists of the 4-byte value `0xffffffff` followed by an 8-byte unsigned integer that gives the actual length (see Section 7.4).

2. A 2-byte version identifier containing the value 2 for DWARF Version 3.

3. A 4-byte or 8-byte offset into the `.debug_info` section of the compilation unit header. In the 32-bit DWARF format, this is a 4-byte unsigned offset; in the 64-bit DWARF format, this field is an 8-byte unsigned offset (see Section 7.4).

4. A 4-byte or 8-byte length containing the size in bytes of the contents of the `.debug_info` section generated to represent this compilation unit. In the 32-bit DWARF format, this is a 4-byte unsigned length; in the 64-bit DWARF format, this is an 8-byte unsigned length (see Section 7.4).

This header is followed by a series of tuples. Each tuple consists of a 4-byte or 8-byte offset followed by a string of non-null bytes terminated by one null byte. In the 32-bit DWARF format, this is a 4-byte offset; in the 64-bit DWARF format, it is an 8-byte offset. Each set is terminated by an offset 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 containing:

1. A 4-byte or 12-byte length containing the length of the set of entries for this compilation unit, not including the length field itself. In the 32-bit DWARF format, this is a 4-byte unsigned integer (which must be less than `0xfffffff00`); in the 64-bit DWARF format, this consists of the 4-byte value `0xffffffff` followed by an 8-byte unsigned integer that gives the actual length (see Section 7.4).

2. A 2-byte version identifier containing the value 2 for DWARF Version 3.

3. A 4-byte or 8-byte offset into the `.debug_info` section of the compilation unit header. In the 32-bit DWARF format, this is a 4-byte unsigned offset; in the 64-bit DWARF format, this is an 8-byte unsigned offset (see Section 7.4).

4. 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.

5. 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 version number in the line number program header is 3 for DWARF Version 3.

The boolean values "true" and "false" used by the line number 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 33.

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
DW_LNS_set_prologue_end ‡	10
DW_LNS_set_epilogue_begin ‡	11
DW_LNS_set_isa ‡	12

‡ New in DWARF V3

Figure 33. Line Number Standard Opcode Encodings

The encodings for the predefined extended opcodes are given in Figure 34.

Opcode Name	Value
DW_LNE_end_sequence	1
DW_LNE_set_address	2
DW_LNE_define_file	3
DW_LNE_lo_user ‡	128
DW_LNE_hi_user ‡	255

‡ New in DWARF V3

Figure 34. Line Number 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 35.

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 35. Macinfo Type Encodings

7.23 Call Frame Information

In the 32-bit DWARF format, the value of the CIE id in the CIE header is `0xffffffff`; in the 64-bit DWARF format, the value is `0xffffffffffffffff`.

The value of the CIE version number is 3.

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 36.

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_nop	0	0		
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_def_cfa_expression	0	0x0f	BLOCK	

Instruction	High 2 Bits	Low 6 Bits	Operand 1	Operand 2
DW_CFA_expression ‡	0	0x10	ULEB128 register	BLOCK
DW_CFA_offset_extended_sf ‡	0	0x11	ULEB128 register	SLEB128 offset
DW_CFA_def_cfa_sf ‡	0	0x12	ULEB128 register	SLEB128 offset
DW_CFA_def_cfa_offset_sf ‡	0	0x13	SLEB128 offset	
DW_CFA_lo_user	0	0x1c		
DW_CFA_hi_user	0	0x3f		

‡ New in DWARF V3

Figure 36. Call frame instruction encodings

7.24 Non-contiguous Address Ranges

Each entry in a range list (see Section 2.16.3) is either a range list entry, a base address selection entry, or an end of list entry.

A range list entry consists of two relative addresses. The addresses are the same size as used by DW_FORM_addr on the target machine.

A base address selection entry and an end of list entry each consist of two (constant or relocated) addresses. The two addresses are the same size as used by DW_FORM_addr on the target machine.

For a range list to be specified, the base address of the corresponding compilation unit must be defined (see Section 3.1).

7.25 Dependencies and Constraints

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`, `.debug_pubtypes`, `.debug_ranges` and `.debug_str` sections of an object file. The information is not word-aligned. Consequently:

- For the 32-bit DWARF format and a target architecture with 32-bit addresses, an assembler or compiler must provide a way to produce 2-byte and 4-byte quantities without alignment restrictions, and the linker must be able to relocate a 4-byte address or section offset that occurs at an arbitrary alignment.

- For the 32-bit DWARF format and a target architecture with 64-bit addresses, an assembler or compiler must provide a way to produce 2-byte, 4-byte and 8-byte quantities without alignment restrictions, and the linker must be able to relocate an 8-byte address or section offset that occurs at an arbitrary alignment.

- For the 64-bit DWARF format (and a target architecture with 64-bit addresses), an assembler or compiler must provide a way to produce 2-byte, 4-byte and 8-byte quantities without alignment restrictions, and the linker must be able to relocate an 8-byte address or section offset that occurs at an arbitrary alignment.

While it is possible in principle to use 4-byte addresses in combination with the 64-bit DWARF file format, it is not expected that this will ever be of interest.

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

7.26 Integer Representation Names

The sizes of the integers used in the lookup by name, lookup by address, line number and call frame information sections are given in Figure 37.

Representation name	Representation
sbyte	signed, 1-byte integer
ubyte	unsigned, 1-byte integer
uhalf	unsigned, 2-byte integer
uword	unsigned, 4-byte integer

Figure 37. Integer Representation Names

Appendix A -- Attributes by Tag Value (informative)

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.

In the following table, DECL means DW_AT_decl_column, DW_AT_decl_file, and DW_AT_decl_line.

TAG Name	Applicable Attributes
DW_TAG_access_declaration	DECL DW_AT_accessibility DW_AT_description DW_AT_name DW_AT_sibling

DW_TAG_array_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_byte_size DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_ordering DW_AT_sibling DW_AT_specification DW_AT_start_scope DW_AT_stride_size DW_AT_type DW_AT_visibility
DW_TAG_base_type	DW_AT_allocated DW_AT_associated DW_AT_bit_offset DW_AT_bit_size DW_AT_byte_size DW_AT_data_location DW_AT_description 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_ranges DW_AT_segment DW_AT_sibling

DW_TAG_class_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_byte_size DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_sibling DW_AT_specification DW_AT_start_scope DW_AT_visibility
DW_TAG_common_block	DECL DW_AT_declaration DW_AT_description DW_AT_location DW_AT_name DW_AT_segment 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_ranges DW_AT_segment DW_AT_sibling DW_AT_stmt_list DW_AT_use_UTF8
DW_TAG_const_type	DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_sibling DW_AT_type
DW_TAG_constant	DECL DW_AT_accessibility DW_AT_const_value DW_AT_declaration DW_AT_description DW_AT_external DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_type DW_AT_visibility
DW_TAG_dwarf_procedure	DW_AT_location

DW_TAG_entry_point	DW_AT_address_class DW_AT_description DW_AT_frame_base 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_allocated DW_AT_associated DW_AT_byte_size DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_sibling DW_AT_specification DW_AT_start_scope DW_AT_stride DW_AT_visibility
DW_TAG_enumerator	DECL DW_AT_const_value DW_AT_description DW_AT_name DW_AT_sibling

DW_TAG_file_type	DECL DW_AT_abstract_origin DW_AT_allocated DW_AT_associated DW_AT_byte_size DW_AT_data_location DW_AT_description DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_type DW_AT_visibility
DW_TAG_formal_parameter	DECL DW_AT_abstract_origin DW_AT_artificial DW_AT_const_value DW_AT_default_value DW_AT_description 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_description DW_AT_import DW_AT_name DW_AT_sibling DW_AT_start_scope

DW_TAG_imported_module	DECL DW_AT_import DW_AT_sibling DW_AT_start_scope
DW_TAG_imported_unit	DW_AT_import
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_call_column DW_AT_call_file DW_AT_call_line DW_AT_entry_pc DW_AT_high_pc DW_AT_low_pc DW_AT_ranges DW_AT_return_addr DW_AT_segment DW_AT_sibling DW_AT_start_scope DW_AT_trampoline
DW_TAG_interface_type	DECL DW_AT_accessibility DW_AT_description DW_AT_name DW_AT_sibling DW_AT_start_scope

DW_TAG_label	DW_AT_abstract_origin DW_AT_description DW_AT_low_pc DW_AT_name DW_AT_segment DW_AT_start_scope DW_AT_sibling
DW_TAG_lexical_block	DW_AT_abstract_origin DW_AT_description DW_AT_high_pc DW_AT_low_pc DW_AT_name DW_AT_ranges DW_AT_segment DW_AT_sibling
DW_TAG_member	DECL DW_AT_accessibility DW_AT_bit_offset DW_AT_bit_size DW_AT_byte_size DW_AT_data_member_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_sibling DW_AT_type DW_AT_visibility

DW_TAG_module	DECL DW_AT_accessibility DW_AT_declaration DW_AT_description DW_AT_entry_pc DW_AT_high_pc DW_AT_low_pc DW_AT_name DW_AT_priority DW_AT_ranges DW_AT_segment DW_AT_sibling DW_AT_specification DW_AT_visibility
DW_TAG_mutable_type	DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_sibling DW_AT_type
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_namespace	DECL DW_AT_description DW_AT_extension DW_AT_name DW_AT_sibling DW_AT_start_scope

DW_TAG_packed_type	DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_sibling DW_AT_type
DW_TAG_partial_unit	DW_AT_base_types DW_AT_comp_dir DW_AT_description 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_ranges DW_AT_segment DW_AT_sibling DW_AT_stmt_list DW_AT_use_UTF8
DW_TAG_pointer_type	DW_AT_address_class DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_sibling DW_AT_specification DW_AT_type

DW_TAG_ptr_to_member_type	DECL DW_AT_abstract_origin DW_AT_address_class DW_AT_allocated DW_AT_associated DW_AT_containing_type DW_AT_data_location DW_AT_declaration DW_AT_description 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_allocated DW_AT_associated DW_AT_data_location DW_AT_sibling DW_AT_type
DW_TAG_restrict_type	DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_sibling DW_AT_type

DW_TAG_set_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_byte_size DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_start_scope DW_AT_sibling DW_AT_type DW_AT_visibility
DW_TAG_string_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_byte_size DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_sibling DW_AT_start_scope DW_AT_string_length DW_AT_visibility

DW_TAG_structure_type	DECL
	DW_AT_abstract_origin
	DW_AT_accessibility
	DW_AT_allocated
	DW_AT_associated
	DW_AT_byte_size
	DW_AT_data_location
	DW_AT_declaration
	DW_AT_description
	DW_AT_name
	DW_AT_sibling
	DW_AT_specification
	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_description
	DW_AT_entry_pc
	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_ranges
	DW_AT_return_addr
	DW_AT_segment
	DW_AT_sibling
	DW_AT_specification
	DW_AT_start_scope
	DW_AT_static_link
	DW_AT_trampoline
	DW_AT_type
	DW_AT_visibility
	DW_AT_virtuality
	DW_AT_vtable_elem_location

DW_TAG_subrange_type	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_byte_size DW_AT_count DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_lower_bound DW_AT_name DW_AT_sibling DW_AT_stride 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_allocated DW_AT_associated DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_name DW_AT_prototyped DW_AT_sibling DW_AT_start_scope DW_AT_type DW_AT_visibility
DW_TAG_template_type_parameter	DECL DW_AT_description DW_AT_name DW_AT_sibling DW_AT_type

DW_TAG_template_value_parameter	DECL DW_AT_const_value DW_AT_description DW_AT_name DW_AT_sibling DW_AT_type
DW_TAG_thrown_type	DECL DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_sibling DW_AT_type
DW_TAG_try_block	DW_AT_abstract_origin DW_AT_high_pc DW_AT_low_pc DW_AT_ranges DW_AT_segment DW_AT_sibling
DW_TAG_typedef	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_declaration DW_AT_description 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_allocated DW_AT_associated DW_AT_data_location DW_AT_declaration DW_AT_description DW_AT_friend DW_AT_name DW_AT_sibling DW_AT_specification DW_AT_start_scope DW_AT_visibility
DW_TAG_unspecified_parameters	DECL DW_AT_abstract_origin DW_AT_artificial DW_AT_sibling
DW_TAG_unspecified_type	DECL DW_AT_description DW_AT_name
DW_TAG_variable	DECL DW_AT_abstract_origin DW_AT_accessibility DW_AT_const_value DW_AT_declaration DW_AT_description 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

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_abstract_origin DW_AT_accessibility DW_AT_declaration DW_AT_discr DW_AT_sibling DW_AT_type
DW_TAG_volatile_type	DW_AT_allocated DW_AT_associated DW_AT_data_location DW_AT_sibling DW_AT_type
DW_TAG_with_stmt	DW_AT_accessibility DW_AT_address_class DW_AT_declaration DW_AT_high_pc DW_AT_location DW_AT_low_pc DW_AT_ranges DW_AT_segment DW_AT_sibling DW_AT_type DW_AT_visibility

Figure 38. Attributes by TAG value

Appendix B -- Debug Section R elationships (informative)

DWARF information is organized into multiple program sections, each of which holds a particular kind of information. In some cases, information in one section refers to information in one or more of the others. These relationships are illustrated by the diagram and associated notes on the following pages.
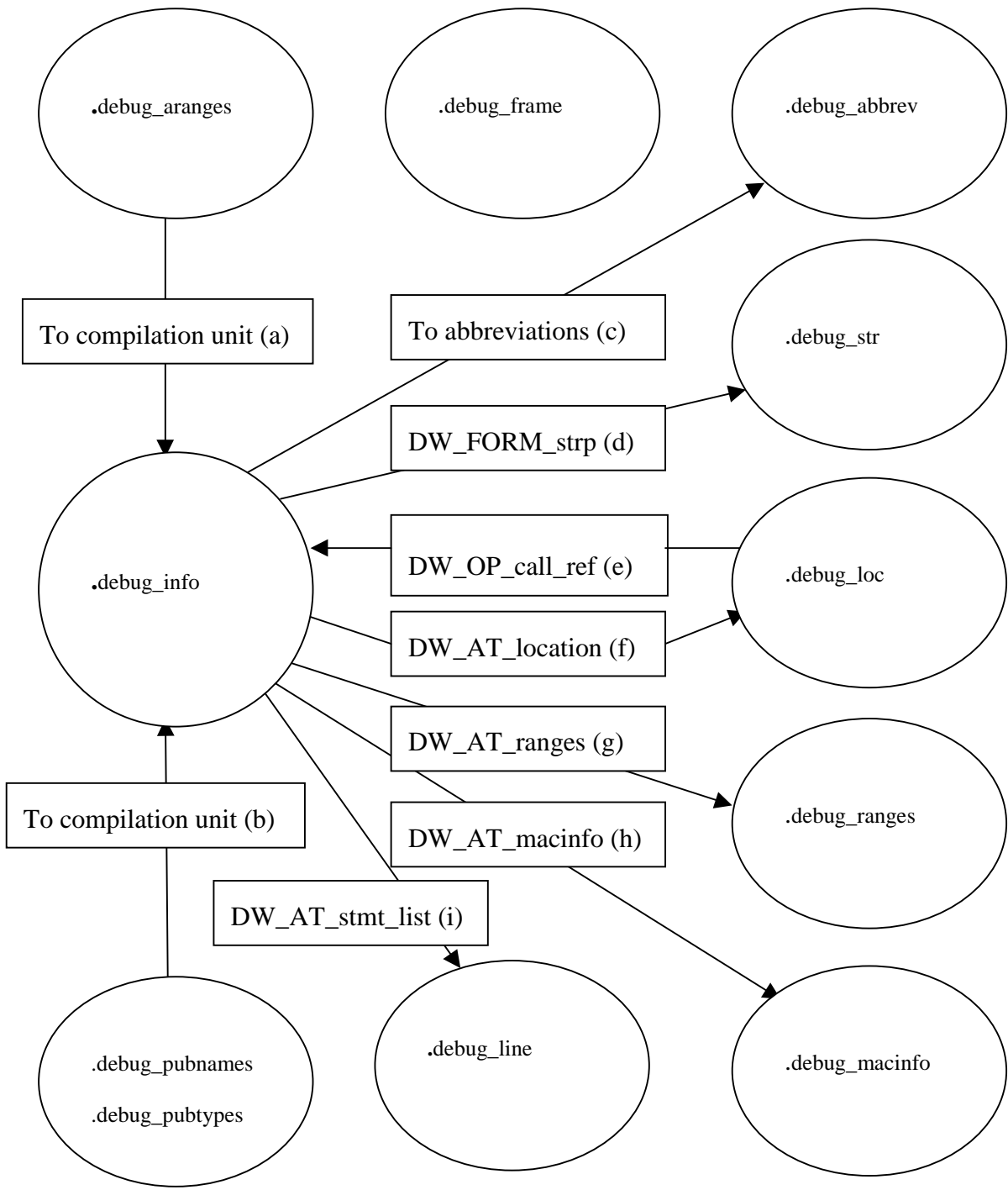
Figure 39. Debug section relationships

(a) `.debug_aranges`   The `debug_info_offset` value in the header is the offset in the
      `.debug_info` section of the corresponding compilation unit header (not
      the compilation unit entry).

(b) `.debug_pubnames` and `.debug_pubtypes`
      The `debug_info_offset` value in the header is the offset in the
      `.debug_info` section of the corresponding compilation unit header (not
      the compilation unit entry). Each pubname has the offset (within the
      corresponding compilation unit) of the applicable debugging information
      entry.

(c) `.debug_info`   The `debug_abbrev_offset` value in the header is the offset in the
      `.debug_abbrev` section of the abbreviations for that compilation unit.

(d) `.debug_info`   Attribute values of class string may have form DW_FORM_strp, whose
      value is the offset in the `.debug_str` section of the corresponding string.

(e) `.debug_loc`   The operand of the DW_OP_call_ref DWARF expression operator is the
      offset of a debugging information entry in the `.debug_info` section.

(f) `.debug_info`   An attribute value of class loclistptr (specifically form
      DW_FORM_data4/8) is an offset within the `.debug_loc` section of a
      location list.

(g) `.debug_info`   An attribute value of class rangelistptr (specifically form
      DW_FORM_data4/8) is an offset within the `.debug_ranges` section of a
      range list.

(h) `.debug_info`   An attribute value of class macptr (specifically form DW_FORM_data4/8)
      is an offset within the `.debug_macinfo` section of the beginning of the
      macro information for the referencing unit.

(i) `.debug_info`   An attribute value of class lineptr (specifically form DW_FORM_data4/8)
      is an offset in the `.debug_line` section of the beginning of the line
      number information for the referencing unit.

Appendix C -- Variable Length D ata: Encoding/Decoding (informative)

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

```
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);
```

Figure 40. Algorithm to encode an unsigned 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;
}
```

Figure 41. Algorithm to encode a signed integer

```
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;
}
```

Figure 42. Algorithm to decode an unsigned LEB128 number

```
result = 0;
shift = 0;
size = number 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);
```

Figure 43. Algorithm to decode a signed LEB128 number

Appendix D -- Examples (inform ative)

The following sections provide examples that illustrate various aspects of the DWARF debugging information format.

D.1 Compilation Units and Abbreviations Table Example

Figure 44 depicts the relationship of the abbreviations tables contained in the `.debug_abbrev` section to the information contained in the `.debug_info` section. Values are given in symbolic form, where possible.

The figure corresponds to the following two trivial source files:

```
---- File myfile.c

char* POINTER;

----- File myfile2.c

typedef char* strp;

-----
```

Compilation Unit 1: `.debug_info`

length   2   *a1 (abbreviations 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 (abbreviations 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_comp_dir	DW_FORM_string
DW_AT_language	DW_FORM_data1
DW_AT_low_pc	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_ref_addr
0	0
0	

Figure 44. Compilation units and abbreviations table

D.2 Aggregate Examples

The following examples illustrate how to represent some of the more complicated forms of array and record aggregates using DWARF.

D.2.1 Fortran 90 Example

Consider the Fortran 90 source fragment in Figure 45.

```
type array_ptr
real :: myvar
real, dimension (:), pointer :: ap
end type array_ptr

type(array_ptr), allocatable, dimension(:) :: arrays

allocate(arrays(20))

do i = 1, 20
allocate(arrays(i)%ap(i+10))
end do
```

Figure 45. Fortran 90 example: source fragment

For allocatable and pointer arrays, it is essentially required by the Fortran 90 semantics that each array consist of two parts, which we here call 1) the descriptor and 2) the raw data. (A descriptor has often been called a dope vector in other contexts, although it is often a structure of some kind rather than a simple vector.) Because there are two parts, and because the lifetime of the descriptor is necessarily longer than and includes that of the raw data, there must be an address somewhere in the descriptor that points to the raw data when, in fact, there is some, that is, when the "variable" is allocated or associated.

For concreteness, suppose that a descriptor looks something like the C structure in Figure 46. Note, however, that it is a property of the design that 1) a debugger needs no builtin knowledge of this structure and 2) there does not need to be an explicit representation of this structure in the DWARF input to the debugger.

```
struct desc {
    long    el_len;         // Element length
    void *  base;           // Address of raw data
    int     ptr_assoc : 1;  // Pointer is associated flag
    int     ptr_alloc : 1;  // Pointer is allocated flag
    int     num_dims  : 6;  // Number of dimensions
    struct  dims_str {      // For each dimension...
        long low_bound;
        long upper_bound;
        long stride;
    } dims[63];
};
```

Figure 46. Fortran 90 example: descriptor representation

In practice, of course, a "real" descriptor will have dimension substructures only for as many dimensions as are specified in the num_dims component. Let us use the notation desc<*n*> to indicate a specialization of the desc struct in which *n* is the bound for the dims component as well as the contents of the num_dims component.

Because the arrays considered here come in two parts, it is necessary to distinguish the parts carefully. In particular, the "address of the variable" or equivalently, the "base address of the object" **always** refers to the descriptor. For arrays that do not come in two parts, an implementation can provide a descriptor anyway, thereby giving it two parts. (This may be convenient for general runtime support unrelated to debugging.) In this case the above vocabulary applies as stated. Alternatively, an implementation can do without a descriptor, in which case the "address of the variable", or equivalently the "base address of the object", refers to the "raw data" (the real data, the only thing around that can be the object).

If an object has a descriptor, then the DWARF type for that object will have a DW_AT_data_location attribute. If an object does not have a descriptor, then usually the DWARF type for the object will not have a DW_AT_data_location. (See the following Ada example for a case where the type for an object without a descriptor does have an DW_AT_data_location attribute. In that case the object doubles as its own descriptor.)

The Fortran 90 derived type array_ptr can now be redescribed in C-like terms that exposes some of the representation as in

```
struct array_ptr {
    float   myvar;
    desc<1> ap;
};
```

Similarly for variable `arrays`:

```
    desc<1> arrays;
```

(Recall that `desc<1>` indicates the 1-dimensional version of `desc`.)

Finally, the following notation is useful:

1. `sizeof(type):` size in bytes of entities of the given type

2. `offset(type, comp):` offset in bytes of the comp component within an entity of the given type

The DWARF description is shown in Figure 47.

```
1$:        DW_TAG_array_type
               ! No name, default (Fortran) ordering, default stride_size
            DW_AT_type(reference to base type REAL)
            DW_AT_associated(machine=        ! Test 'assoc' flag
                DW_OP_push_object_address
                DW_OP_lit<n>                 ! where n == offset(assoc)
                DW_OP_plus
                DW_OP_deref
                DW_OP_lit1                   ! mask for 'assoc' flag
                DW_OP_and)
            DW_AT_data_location(machine=     ! Get raw data address
                DW_OP_push_object_address
                DW_OP_lit<n>                 ! where n == offset(base)
                DW_OP_plus
                DW_OP_deref)
2$:         DW_TAG_subrange_type
               ! No name, default stride
            DW_AT_type(reference to base type INTEGER)
            DW_AT_lower_bound(machine=
                DW_OP_push_object_address
                DW_OP_lit<n>                 ! where n ==
                                             !   offset(desc, dims) +
                                             !   offset(dims_str, lower_bound)
                DW_OP_plus
                DW_OP_deref)
            DW_AT_upper_bound(machine=
                DW_OP_push_object_address
                DW_OP_lit<n>                 ! where n ==
                                             !   offset(desc, dims) +
                                             !   offset(dims_str, upper_bound)
                DW_OP_plus
                DW_OP_deref)
```

```
                    ! Note: for the m'th dimension, the second operator becomes
                    ! DW_OP_lit<x> where
                    !    x == offset(desc, dims)     +
                    !                 (m-1)*sizeof(dims_str)  +
                    !                 offset(dims_str, [lower|upper]_bound)
                    ! That is, the stack machine does not get longer for each
                    ! successive dimension (other than to express the larger
                    ! offsets involved).

3$:        DW_TAG_structure_type
                DW_AT_name("array_ptr")
                DW_AT_byte_size(constant sizeof(REAL) + sizeof(desc<1>))
4$:             DW_TAG_member
                    DW_AT_name("myvar")
                    DW_AT_type(reference to base type REAL)
                    DW_AT_data_member_location(constant 0)
5$:             DW_TAG_member
                    DW_AT_name("ap");
                    DW_AT_type(reference to 1$)
                    DW_AT_data_member_location(constant sizeof(REAL))

6$:        DW_TAG_array_type
                ! No name, default (Fortran) ordering, default stride_size
                DW_AT_type(reference to 3$)
                DW_AT_allocated(machine=         ! Test 'ptr_alloc' flag
                    DW_OP_push_object_address
                    DW_OP_lit<n>                 ! where n == offset(ptr_alloc)
                    DW_OP_plus
                    DW_OP_deref
                    DW_OP_lit2                   ! mask for 'ptr_alloc' flag
                    DW_OP_and)
                DW_AT_data_location(machine=     ! Get raw data address
                    DW_OP_push_object_address
                    DW_OP_lit<n>                 ! where n = offset(base)
                    DW_OP_plus
                    DW_OP_deref)
7$:             DW_TAG_subrange_type
                    ! No name, default stride
                    DW_AT_type(reference to base type INTEGER)
                    DW_AT_lower_bound(machine=
                        DW_OP_push_object_address
                        DW_OP_lit<n>             ! where n == ...
                        DW_OP_plus
                        DW_OP_deref)
                    DW_AT_upper_bound(machine=
                        DW_OP_push_object_address
                        DW_OP_lit<n>             ! where n == ...
                        DW_OP_plus
                        DW_OP_deref)
```

```
8$:        DW_TAG_variable
             DW_AT_name("arrays")
             DW_AT_type(reference to 6$)
             DW_AT_location(machine=
                 ...as appropriate...)        ! Assume static allocation
```

Figure 47. Fortran 90 example: DWARF description

Suppose the program is stopped immediately following completion of the do loop. Suppose
further that the user enters the following debug command:

```
    debug> print arrays(5)%ap(2)
```

Interpretation of this expression proceeds as follows:

1) Lookup name `arrays`. We find that it is a variable, whose type is given by the unnamed type
   at 6$. Notice that the type is an array type.

2) Find the 5th element of that array object. To do array indexing requires several pieces of
   information:

   a) the address of the array data

   b) the lower bounds of the array

      [To check that 5 is within bounds would require the upper bound too, but we'll skip that
      for this example.]

   c) the stride size

   For a), check for a DW_AT_data_location attribute. Since there is one, go execute the stack
   machine, whose result is the address needed. The object address used in this case is the object
   we are working on, namely the variable named `arrays`, whose address was found in step 1.
   (Had there been no DW_AT_data_location attribute, the desired address would be the same
   as the address from step 1.)

   For b), for each dimension of the array (only one in this case), go interpret the usual lower
   bound attribute. Again this is a stack machine, which again begins with
   DW_OP_push_object_address. This object is **still** `arrays`, from step 1, because we haven't
   begun to actually perform any indexing yet.

For c), the default stride size applies. Since there is no DW_AT_stride attribute, use the size of the array element type, which is the size of type `array_ptr` (at 3$).

Having acquired all the necessary data, perform the indexing operation in the usual manner—which has nothing to do with any of the attributes involved up to now. Those just provide the actual values used in the indexing step.

The result is an object within the memory that was dynamically allocated for `arrays`.

3) Find the `ap` component of the object just identified, whose type is `array_ptr`.

This is a conventional record component lookup and interpretation. It happens that the `ap` component in this case begins at offset 4 from the beginning of the containing object. Component `ap` has the unnamed array type defined at 1$ in the symbol table.

4) Find the 2nd element of the array object found in step 3. To do array indexing requires several pieces of information:

   a) the address of the array storage

   b) the lower bounds of the array

   [To check that 2 is within bounds we would require the upper bound too, but we'll skip that for this example]

   c) the stride size

This is just like step 2), so the details are omitted. Recall that because the DWARF type 1$ has a DW_AT_data_location, the address that results from step 4) is that of a descriptor, and that address is the address pushed by the DW_OP_push_object_address operations in 1$ and 2$.

Note: we happen to be accessing a pointer array here instead of an allocatable array; but because there is a common underlying representation, the mechanics are the same. There could be completely different descriptor arrangements and the mechanics would still be the same—only the stack machines would be different.

D.2.2   Ada Example

Figure 48 illustrates two kinds of Ada parameterized array, one embedded in a record.

```
M : INTEGER := <exp>;
VEC1 : array (1..M) of INTEGER;

subtype TEENY is INTEGER range 1..100;
type ARR is array (INTEGER range <>) of INTEGER;
type REC2(N : TEENY := 100) is record
    VEC2 : ARR(1..N);
end record;

OBJ2B : REC2;
```

Figure 48. Ada example: source fragment

VEC1 illustrates an (unnamed) array type where the upper bound of the first and only dimension is determined at runtime. Ada semantics require that the value of an array bound is fixed at the time the array type is elaborated (where *elaboration* refers to the runtime executable aspects of type processing). For the purposes of this example, we assume that there are no other assignments to M so that it safe for the REC1 type description to refer directly to that variable (rather than a compiler generated copy).

REC2 illustrates another array type (the unnamed type of component VEC2) where the upper bound of the first and only bound is also determined at runtime. In this case, the upper bound is contained in a discriminant of the containing record type. (A discriminant is a component of a record whose value cannot be changed independently of the rest of the record because that value is potentially used in the specification of other components of the record.)

The DWARF description is shown in Figure 49.

Interesting aspects about this example are:

1) The array VEC2 is "immediately" contained within structure REC2 (there is no intermediate descriptor or indirection), which is reflected in the absence of a DW_AT_data_location attribute on the array type at 28$.

2) One of the bounds of VEC2 is nonetheless dynamic and part of the same containing record. It is located using an address calculation relative to the VEC2 component itself.

3) The lack of a subtype concept in DWARF means that DWARF types serve the role of subtypes and must replicate information from what should be the parent type. For this reason, DWARF for the unconstrained array ARR is not needed for the purposes of this example and therefore not shown.

```
11$:    DW_TAG_variable
            DW_AT_name("M")
            DW_AT_type(reference to base type INTEGER)

12$:    DW_TAG_array_type
            ! No name, default (Ada) order, default stride
            DW_AT_type(reference to base type INTEGER)
13$:        DW_TAG_subrange_type
                DW_AT_type(reference to base type INTEGER)
                DW_AT_lower_bound(constant 1)
                DW_AT_upper_bound(reference to variable M at 11$)

14$:    DW_TAG_variable
            DW_AT_name("VEC1")
            DW_AT_type(reference to array type at 12$)

        . . .

21$:    DW_TAG_subrange_type
            DW_AT_name("TEENY")
            DW_AT_type(reference to base type INTEGER)
            DW_AT_lower_bound(constant 1)
            DW_AT_upper_bound(constant 100)

        . . .
```

```
26$:     DW_TAG_structure_type
             DW_AT_name("REC2")
27$:         DW_TAG_member
                 DW_AT_name("N")
                 DW_AT_type(reference to subtype TEENY at 21$)
                 DW_AT_data_member_location(constant 0)
28$:         DW_TAG_array_type
                 ! No name, default (Ada) order, default stride
                 ! Default data location
                 DW_AT_TYPE(reference to base type INTEGER)
29$:             DW_TAG_subrange_type
                     DW_AT_type(reference to subrange TEENY at 21$)
                     DW_AT_lower_bound(constant 1)
                     DW_AT_upper_bound(machine=
                         DW_OP_push_object_address,
                         DW_OP_lit<n>    ! where n ==
                                         !   offset(REC2, VEC2) -
                                         !   offset(REC2, N)
                         DW_OP_neg
                         DW_OP_plus      ! compute value of N
                         DW_OP_deref)    !   given address of VEC
30$:         DW_TAG_member
                 DW_AT_name("VEC2")
                 DW_AT_type(reference to array "subtype" at 28$)
                 DW_AT_data_member_location(machine=
                     DW_OP_lit<n>           ! where n == offset(REC2, VEC2)
                     DW_OP_plus)

        . . .

41$:     DW_TAG_variable
             DW_AT_name("OBJ2B")
             DW_AT_type(reference to type REC2 at 26$)
             DW_AT_location(...as appropriate...)
```

Figure 49. Ada example: DWARF description

D.3 Namespace Examples

The C++ example in Figure 50 is used to illustrate the representation of namespaces.

```
namespace {
    int i;
}

namespace A {
    namespace B {
        int j;
        int   myfunc (int a);
        float myfunc (float f) { return f – 2.0; }
        int   myfunc2(int a)   { return a + 2; }
    }
}

namespace Y {
    using A::B::j;                 // (1) using declaration
    int foo;
}

using A::B::j;                     // (2) using declaration

namespace Foo = A::B;              // (3) namespace alias

using Foo::myfunc;                 // (4) using declaration

using namespace Foo;               // (5) using directive

namespace A {
    namespace B {
        using namespace Y;        // (6) using directive
        int k;
    }
}

int Foo::myfunc(int a)
{
    i = 3;
    j = 4;
    return myfunc2(3) + j + i + a + 2;
}
```

Figure 50. Namespace example: source fragment

The DWARF representation in Figure 51 is appropriate.

```
1$:      DW_TAG_basetype
             DW_AT_name("int")
             ...
2$:      DW_TAG_basetype
             DW_AT_name("float")
             ...
6$:      DW_TAG_namespace
             ! no DW_AT_name attribute
7$:          DW_TAG_variable
                 DW_AT_name("i")
                 DW_AT_type(reference to 1$)
                 DW_AT_location ...
                 ...

10$:     DW_TAG_namespace
             DW_AT_name("A")
20$:         DW_TAG_namespace
                 DW_AT_name("B")
30$:             DW_TAG_variable
                     DW_AT_name("j")
                     DW_AT_type(reference to 1$)
                     DW_AT_location ...
                     ...
34$:             DW_TAG_subprogram
                     DW_AT_name("myfunc")
                     DW_AT_type(reference to 1$)
                     ...
36$:             DW_TAG_subprogram
                     DW_AT_name("myfunc")
                     DW_AT_type(reference to 2$)
                     ...
38$:             DW_TAG_subprogram
                     DW_AT_name("myfunc2")
                     DW_AT_low_pc ...
                     DW_AT_high_pc ...
                     DW_AT_type(reference to 1$)
                     ...
```

```
40$:        DW_TAG_namespace
                DW_AT_name("Y")
                DW_TAG_imported_declaration            ! (1) using-declaration
                    DW_AT_import(reference to 30$)
                DW_TAG_variable
                    DW_AT_name("foo")
                    DW_AT_type(reference to 1$)
                    DW_AT_location ...
                    ...

            DW_TAG_imported_declaration                ! (2) using declaration
                DW_AT_import(reference to 30$)

            DW_TAG_imported_declaration                ! (3) namespace alias
                DW_AT_name("Foo")
                DW_AT_import(reference to 20$)

            DW_TAG_imported_declaration                ! (4) using declaration
                DW_AT_import(reference to 34$)         !     -  part 1
            DW_TAG_imported_declaration                ! (4) using declaration
                DW_AT_import(reference to 36$)         !     -  part 2

            DW_TAG_imported_module                     ! (5) using directive
                DW_AT_import(reference to 20$)

            DW_TAG_namespace
                DW_AT_extension(reference to 10$)
                DW_TAG_namespace
                    DW_AT_extension(reference to 20$)
                    DW_TAG_imported_module             ! (6) using directive
                        DW_AT_import(reference to 40$)
                        DW_TAG_variable
                            DW_AT_name("k")
                            DW_AT_type(reference to 1$)
                            DW_AT_location ...
                            ...

60$:        DW_TAG_subprogram
                DW_AT_name("myfunc")
                DW_AT_specification(reference to 34$)
                DW_AT_low_pc ...
                DW_AT_high_pc ...
                ...
```

Figure 51. Namespace example: DWARF description

D.4 Line Number Program Example

Consider the simple source file and the resulting machine code for the Intel 8086 processor in Figure 52.

```
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:
```

Figure 52. Line number program example: machine code

Suppose the line number program header specifies the following:

```
version                      3
minimum_instruction_length   1
opcode_base                  10    ! Opcodes 10-12 not needed
line_base                    1
line_range                   15
```

Figure 53 shows one encoding of the line number program, which occupies 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
```

Figure 53. Line number program example: one encoding

Figure 54 shows an alternate encoding of the same program using standard opcodes to advance the program counter; this encoding occupies 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
```

Figure 54. Line number program example: alternate encoding

D.5 Call Frame Information Example

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. <fs> denotes the stack frame size in bytes, namely 12.

```
        ;; start prologue
foo     sub   R7, R7, <fs>       ; Allocate frame
foo+4   store R1, R7, (<fs>-4)   ; Save the return address
foo+8   store R6, R7, (<fs>-8)   ; Save R6
foo+12  add   R6, R7, 0          ; R6 is now the Frame ptr
foo+16  store R4, R6, (<fs>-12)  ; Save a preserved reg
        ;; This subroutine does not change R5

        ...
        ;; Start epilogue (R7 is returned to entry value)
foo+64  load R4, R6, (<fs>-12)   ; Restore R4
foo+68  load R6, R7, (<fs>-8)    ; Restore R6
foo+72  load R1, R7, (<fs>-4)    ; Restore return address
foo+76  add   R7, R7, <fs>       ; Deallocate frame
foo+80  jump R1                  ; Return
foo+84
```

Figure 55. Call frame information example: machine code fragments

The conceptual matrix for the foo subroutine is shown in Figure 56. Corresponding fragments from the `.debug_frame` section are shown in Figure 57.

Location	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]+fs	s	u	u	u	s	s	s	s	r1
foo+8	[R7]+fs	s	u	u	u	s	s	s	s	c-4
foo+12	[R7]+fs	s	u	u	u	s	s	c-8	s	c-4
foo+16	[R6]+fs	s	u	u	u	s	s	c-8	s	c-4
foo+20	[R6]+fs	s	u	u	u	c-12	s	c-8	s	c-4
foo+64	[R6]+fs	s	u	u	u	c-12	s	c-8	s	c-4
foo+68	[R6]+fs	s	u	u	u	s	s	c-8	s	c-4
foo+72	[R7]+fs	s	u	u	u	s	s	s	s	c-4
foo+76	[R7]+fs	s	u	u	u	s	s	s	s	r1
foo+80	[R7]+0	s	u	u	u	s	s	s	s	r1

Figure 56. Call frame information example: conceptual matrix

The following notes apply to Figure 56:

       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

Address	Value	Comment
cie	32	length
cie+4	0xffffffff	CIE_id
cie+8	1	version
cie+9	0	augmentation
cie+10	4	code_alignment_factor, <caf>
cie+11	-4	data_alignment_factor, <daf>
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		

Figure 57. Call frame information example: common information entry encoding

Address	Value	Comment
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(12)	<fs>
fde+19	DW_CFA_advance_loc(1)	4/<caf>
fde+20	DW_CFA_offset(8,1)	-4/<daf> (2nd parameter)
fde+22	DW_CFA_advance_loc(1)	
fde+23	DW_CFA_offset(6,2)	-8/<daf> (2nd parameter)
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)	-12/<daf> (2nd parameter)
fde+31	DW_CFA_advance_loc(11)	44/<caf>
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		

Figure 58. Call frame information example: frame description entry encoding

D.6 Inlining Examples

The pseudo-source in Figure 59 is used to illustrate the use of DWARF to describe inlined subroutine calls. This example involves a nested subprogram INNER that makes uplevel references to the formal parameter and local variable of the containing subprogram OUTER.

```
inline procedure OUTER (OUTER_FORMAL : integer) =
    begin

    OUTER_LOCAL : integer;

    procedure INNER (INNER_FORMAL : integer) =
        begin

        INNER_LOCAL : integer;

        print(INNER_FORMAL + OUTER_LOCAL);

        end;

    INNER(OUTER_LOCAL);
    ...
    INNER(31);

    end;

! Call OUTER
!
OUTER(7);
```

Figure 59. Inlining examples: pseudo-source fragment

There are several approaches that a compiler might take to inlining for this sort of example. This presentation considers three such approaches, all of which involve inline expansion of subprogram OUTER. (If OUTER is not inlined, the inlining reduces to a simpler single level subset of the two level approaches considered here.)

The approaches are:

1. Inline both OUTER and INNER in all cases

2. Inline OUTER, multiple INNERs

    Treat INNER as a non-inlinable part of OUTER, compile and call a distinct normal version of INNER defined within each inlining of OUTER.

3. Inline OUTER, one INNER

    Compile INNER as a single normal subprogram which is called from every inlining of OUTER.

This discussion does not consider why a compiler might choose one of these approaches; it considers only how to describe the result.

In the examples that follow in this section, the debugging information entries are given mnemonic labels of the following form

        <io>.<ac>.<n>.<s>

where `<io>` is either INNER or OUTER to indicate to which subprogram the debugging information entry applies, `<ac>` is either AI or CI to indicate "abstract instance" or "concrete instance" respectively, `<n>` is the number of the alternative being considered, and `<s>` is a sequence number that distinguishes the individual entries. There is no implication that symbolic labels, nor any particular naming convention, are required in actual use.

For conciseness, declaration coordinates and call coordinates are omitted.

D.6.1   Alternative #1: inline both OUTER and INNER

A suitable abstract instance for an alternative where both OUTER and INNER are always inlined is shown in Figure 60.

Notice in Figure 60 that the debugging information entry for INNER (labelled INNER.AI.1.1) is nested in (is a child of) that for OUTER (labelled OUTER.AI.1.1). Nonetheless, the abstract instance tree for INNER is considered to be separate and distinct from that for OUTER.

The call of OUTER shown in Figure 59 might be described as shown in Figure 61.

```
        ! Abstract instance for OUTER
        !
OUTER.AI.1.1:
        DW_TAG_subprogram
            DW_AT_name("OUTER")
            DW_AT_inline(DW_INL_declared_inlined)
            ! No low/high PCs
OUTER.AI.1.2:
            DW_TAG_formal_parameter
                DW_AT_name("OUTER_FORMAL")
                DW_AT_type(reference to integer)
                ! No location
OUTER.AI.1.3:
            DW_TAG_variable
                DW_AT_name("OUTER_LOCAL")
                DW_AT_type(reference to integer)
                ! No location
            !
            ! Abstract instance for INNER
            !
INNER.AI.1.1:
            DW_TAG_subprogram
                DW_AT_name("INNER")
                DW_AT_inline(DW_INL_declared_inlined)
                ! No low/high PCs
INNER.AI.1.2:   DW_TAG_formal_parameter
                    DW_AT_name("INNER_FORMAL")
                    DW_AT_type(reference to integer)
                    ! No location
INNER.AI.1.3:   DW_TAG_variable
                    DW_AT_name("INNER_LOCAL")
                    DW_AT_type(reference to integer)
                    ! No location
            ...
            0

        ! No DW_TAG_inlined_subroutine (concrete instance)
        ! for INNER corresponding to calls of INNER

        ...
        0
```

Figure 60. Inlining example #1: abstract instance

```
        ! Concrete instance for call "OUTER(7)"
        !
OUTER.CI.1.1:
        DW_TAG_inlined_subroutine
            ! No name
            DW_AT_abstract_origin(reference to OUTER.AI.1.1)
            DW_AT_low_pc(...)
            DW_AT_high_pc(...)
OUTER.CI.1.2:
            DW_TAG_formal_parameter
                ! No name
                DW_AT_abstract_origin(reference to OUTER.AI.1.2)
                DW_AT_const_value(7)
OUTER.CI.1.3:
            DW_TAG_variable
                ! No name
                DW_AT_abstract_origin(reference to OUTER.AI.1.3)
                DW_AT_location(...)
            !
            ! No DW_TAG_subprogram (abstract instance) for INNER
            !
            ! Concrete instance for call INNER(OUTER_LOCAL)
            !
INNER.CI.1.1:
            DW_TAG_inlined_subroutine
                ! No name
                DW_AT_abstract_origin(reference to INNER.AI.1.1)
                DW_AT_low_pc(...)
                DW_AT_high_pc(...)
                DW_AT_static_link(...)
INNER.CI.1.2:   DW_TAG_formal_parameter
                    ! No name
                    DW_AT_abstract_origin(reference to INNER.AI.1.2)
                    DW_AT_location(...)
INNER.CI.1.3:   DW_TAG_variable
                    ! No name
                    DW_AT_abstract_origin(reference to INNER.AI.1.3)
                    DW_AT_location(...)
                ...
                0

        ! Another concrete instance of INNER within OUTER
        ! for the call "INNER(31)"

        ...
        0
```

Figure 61. Inlining example #1: concrete instance

D.6.2 Alternative #2: Inline OUTER, multiple INNERs

In the second alternative we assume that subprogram INNER is not inlinable for some reason, but subprogram OUTER is inlinable. Each concrete inline instance of OUTER has its own normal instance of INNER. The abstract instance for OUTER, which includes INNER, is shown in Figure 62.

Note that the debugging information in this Figure differs from that in Figure 60 in that INNER lacks a DW_AT_inline attribute and therefore is not a distinct abstract instance. INNER is merely an out-of-line routine that is part of OUTER's abstract instance. This is reflected in the Figure by the fact that the labels for INNER use the substring OUTER instead of INNER.

A resulting concrete inline instance of OUTER is shown in Figure 63.

Notice in Figure 63 that OUTER is expanded as a concrete inlined instance, and that INNER is nested within it as a concrete out-of-line subprogram. Because INNER is cloned for each inline expansion of OUTER, only the invariant attributes of INNER (for example, DW_AT_name) are specified in the abstract instance of OUTER, and the low-level, instance-specific attributes of INNER (for example, DW_AT_low_pc) are specified in each concrete instance of OUTER.

The several calls of INNER within OUTER are compiled as normal calls to the instance of INNER that is specific to the same instance of OUTER that contains the calls.

```
          ! Abstract instance for OUTER
          !
OUTER.AI.2.1:
        DW_TAG_subprogram
            DW_AT_name("OUTER")
            DW_AT_inline(DW_INL_declared_inlined)
            ! No low/high PCs
OUTER.AI.2.2:
            DW_TAG_formal_parameter
                DW_AT_name("OUTER_FORMAL")
                DW_AT_type(reference to integer)
                ! No location
OUTER.AI.2.3:
            DW_TAG_variable
                DW_AT_name("OUTER_LOCAL")
                DW_AT_type(reference to integer)
                ! No location
            !
            ! Nested out-of-line INNER subprogram
            !
OUTER.AI.2.4:
            DW_TAG_subprogram
                DW_AT_name("INNER")
                ! No DW_AT_inline
                ! No low/high PCs, frame_base, etc.
OUTER.AI.2.5:
                DW_TAG_formal_parameter
                    DW_AT_name("INNER_FORMAL")
                    DW_AT_type(reference to integer)
                    ! No location
OUTER.AI.2.6:   DW_TAG_variable
                    DW_AT_name("INNER_LOCAL")
                    DW_AT_type(reference to integer)
                    ! No location
                ...
                0
            ...
            0
```

Figure 62. Inlining example #2: abstract instance

```
          ! Concrete instance for call "OUTER(7)"
          !
OUTER.CI.2.1:
        DW_TAG_inlined_subroutine
            ! No name
            DW_AT_abstract_origin(reference to OUTER.AI.2.1)
            DW_AT_low_pc(...)
            DW_AT_high_pc(...)
OUTER.CI.2.2:
            DW_TAG_formal_parameter
                ! No name
                DW_AT_abstract_origin(reference to OUTER.AI.2.2)
                DW_AT_location(...)
OUTER.CI.2.3:
            DW_TAG_variable
                ! No name
                DW_AT_abstract_origin(reference to OUTER.AI.2.3)
                DW_AT_location(...)
            !
            ! Nested out-of-line INNER subprogram
            !
OUTER.CI.2.4:
            DW_TAG_subprogram
                ! No name
                DW_AT_abstract_origin(reference to OUTER.AI.2.4)
                DW_AT_low_pc(...)
                DW_AT_high_pc(...)
                DW_AT_frame_base(...)
                DW_AT_static_link(...)
OUTER.CI.2.5:
                DW_TAG_formal_parameter
                    ! No name
                    DW_AT_abstract_origin(reference to OUTER.AI.2.5)
                    DW_AT_location(...)
OUTER.CI.2.6:
                DW_TAG_variable
                    ! No name
                    DW_AT_abstract_origin(reference to OUTER.AT.2.6)
                    DW_AT_location(...)
                ...
                0
            ...
            0
```

Figure 63. Inlining example #2: concrete instance

D.6.3 Alternative #3: inline OUTER, one normal INNER

In the third approach, one normal subprogram for INNER is compiled which is called from all concrete inline instances of OUTER. The abstract instance for OUTER is shown in Figure 64.

The most distinctive aspect of that Figure is that subprogram INNER exists only within the abstract instance of OUTER, and not in OUTER's concrete instance. In the abstract instance of OUTER, the description of INNER has the full complement of attributes that would be expected for a normal subprogram. While attributes such as DW_AT_low_pc, DW_AT_high_pc, DW_AT_location, and so on, typically are omitted from an abstract instance because they are not invariant across instances of the containing abstract instance, in this case those same attributes are included precisely because they are invariant--there is only one subprogram INNER to be described and every description is the same.

A concrete inline instance of OUTER is illustrated in Figure 65.

Notice in Figure 65 that there is no DWARF representation for INNER at all; the representation of INNER does not vary across instances of OUTER and the abstract instance of OUTER includes the complete description of INNER, so that the description of INNER may be (and for reasons of space efficiency, should be) omitted from each concrete instance of OUTER.

There is one aspect of this approach that is problematical from the DWARF perspective. The single compiled instance of INNER is assumed to access up-level variables of OUTER; however, those variables may well occur at varying positions within the frames that contain the concrete inline instances. A compiler might implement this in several ways, including the use of additional compiler generated parameters that provide reference parameters for the up-level variables, or a compiler generated static link like parameter that points to the group of up-level entities, among other possibilities. In either of these cases, the DWARF description for the location attribute of each uplevel variable needs to be different if accessed from within INNER compared to when accessed from within the instances of OUTER. An implementation is likely to require vendor-specific DWARF attributes and/or debugging information entries to describe such cases.

Note that in C++, a member function of a class defined within a function definition does not require any vendor-specific extensions because the C++ language disallows access to entities that would give rise to this problem. (Neither `extern` variables nor `static` members require any form of static link for accessing purposes.)

```
        ! Abstract instance for OUTER
        !
OUTER.AI.3.1:
        DW_TAG_subprogram
            DW_AT_name("OUTER")
            DW_AT_inline(DW_INL_declared_inlined)
            ! No low/high PCs
OUTER.AI.3.2:
            DW_TAG_formal_parameter
                DW_AT_name("OUTER_FORMAL")
                DW_AT_type(reference to integer)
                ! No location
OUTER.AI.3.3:
            DW_TAG_variable
                DW_AT_name("OUTER_LOCAL")
                DW_AT_type(reference to integer)
                ! No location
            !
            ! Normal INNER
            !
OUTER.AI.3.4:
            DW_TAG_subprogram
                DW_AT_name("INNER")
                DW_AT_low_pc(...)
                DW_AT_high_pc(...)
                DW_AT_frame_base(...)
                DW_AT_static_link(...)
OUTER.AI.3.5:
                DW_TAG_formal_parameter
                    DW_AT_name("INNER_FORMAL")
                    DW_AT_type(reference to integer)
                    DW_AT_location(...)
OUTER.AI.3.6:
                DW_TAG_variable
                    DW_AT_name("INNER_LOCAL")
                    DW_AT_type(reference to integer)
                    DW_AT_location(...)
                ...
                0
            ...
            0
```

Figure 64. Inlining example #3: abstract instance

```
        ! Concrete instance for call "OUTER(7)"
        !
OUTER.CI.3.1:
        DW_TAG_inlined_subroutine
            ! No name
            DW_AT_abstract_origin(reference to OUTER.AI.3.1)
            DW_AT_low_pc(...)
            DW_AT_high_pc(...)
            DW_AT_frame_base(...)
OUTER.CI.3.2:
            DW_TAG_formal_parameter
                ! No name
                DW_AT_abstract_origin(reference to OUTER.AI.3.2)
                ! No type
                DW_AT_location(...)
OUTER.CI.3.3:
            DW_TAG_variable
                ! No name
                DW_AT_abstract_origin(reference to OUTER.AI.3.3)
                ! No type
                DW_AT_location(...)

            ! No DW_TAG_subprogram for "INNER"

            ...
            0
```

Figure 65. Inlining example #3: concrete instance

Appendix E -- DWARF Compres sion and Duplicate Elimination (informative)

DWARF can use a lot of disk space.

This is especially true for C++, where the depth and complexity of headers can mean that many, many (possibly thousands of) declarations are repeated in every compilation unit. C++ templates can also mean that some functions and their DWARF descriptions get duplicated.

This Appendix describes techniques for using the DWARF representation in combination with features and characteristics of some common object file representations to reduce redundancy without losing information. It is worth emphasizing that none of these techniques are necessary to provide a complete and accurate DWARF description; they are solely concerned with reducing the size of DWARF information.

The techniques described here depend more directly and more obviously on object file concepts and linker mechanisms than most other parts of DWARF. While the presentation tends to use the vocabulary of specific systems, this is primarily to aid in describing the techniques by appealing to well-known terminology. These techniques can be employed on any system that supports certain general functional capabilities (described below).

E.1 Overview

The general approach is to break up the debug information of a compilation into separate compilation units, each consisting of one or more sections. By arranging that a sufficiently similar partitioning occurs in other compilations, a suitable system linker can delete redundant groups of sections when combining object files.

The following uses some traditional section naming here but aside from the DWARF sections, the names are just mean to suggest traditional contents as a way of explaining the approach, not to be limiting.

A traditional relocatable object output from a single compilation might contain sections named:

```
.data
.text
.debug_info
.debug_abbrev
.debug_line
.debug_aranges
```

A relocatable object from a compilation system attempting duplicate DWARF elimination might contain sections as in:

```
.data
.text
.debug_info
.debug_abbrev
.debug_line
.debug_aranges
```

followed (or preceded, the order is not significant) by a series of section groups:

```
==== Section group 1
      .debug_info
      .debug_abbrev
      .debug_line

==== ...

==== Section group N
      .debug_info
      .debug_abbrev
      .debug_line
```

where each section group might or might not contain executable code (`.text` sections) or data (`.data` sections).

A *section group* is a named set of section contributions within an object file with the property that the entire set of section contributions must be retained or discarded as a whole; no partial elimination is allowed. Section groups can generally be handled by a linker in two ways:

11. Given multiple identical (duplicate) section groups, one of them is chosen to be kept and used, while the rest are discarded.

12. Given a section group that is not referenced from any section outside of the section group, the section group is discarded.

13. Which handling applies may be indicated by the section group itself and/or selection of certain linker options.

For example, if a linker determins that section group 1 from A.o and section group 3 from B.o are identical, it could discard one group and arrange that all references in A.o and B.o apply to the remaining one of the two identical section groups. This saves space.

An important part of making it possible to "redirect" references to the surviving section group is the use of consistently chosen linker global symbols for referring to locations within each section group. It follows that references are simply to external names and the linker already knows how to match up references and definitions.

What is minimally needed from the object file format and system linker (outside of DWARF itself, and normal object/linker facilities such as simple relocations) are:

1. A means of referencing from inside one `.debug_info` compilation unit to another `.debug_info` compilation unit (DW_FORM_ref_addr provides this).

2. A means of having multiple contributions to specific sections (for example, `.debug_info`, and so on) in a single object file.

3. A means of identifying a section group (giving it a name).

4. A means of identifying which sections go together to make up a section group, so that the group can be treated as a unit (kept or discarded).

5. A means of indicating how each section group should be processed by the linker.

The notion of section and section contribution used here corresponds closely to the similarly named concepts in the ELF object file representation. The notion of section group is an abstraction of common extensions of the ELF representation widely known as "COMDATs" or "COMDAT sections". (Other object file representations provide COMDAT-style mechanisms as well.) There are several variations in the COMDAT schemes in common use, any of which should be sufficient for the purposes of the DWARF duplicate elimination techniques described here.

E.2 Naming and Usage Considerations

A precise description of the means of deriving names usable by the linker to access DWARF entities is not part of this specification. Nonetheless, an outline of a usable approach is given here to make this more understandable and to guide implementors.

Implementations should clearly document their naming conventions.

In the following, it will be helpful to refer to the examples in Figure 66 through Figure 73 of Section E.3.

E.2.1   Section Group Names

Section groups must have a section group name. For the subsequent C++ example, a name like

```
<producer-prefix>.<file-designator>.<gid-number>
```

will suffice, where

- `<producer-prefix>` is some string specific to the producer, which has a language-designation embedded in the name when appropriate. (Alternatively, the language name could be embedded in the <gid-number>).

- `<file-designator>` names the file, such as wa.h in the example.

- `<gid-number>` is a string generated to identify the specific wa.h header file in such a way that

  - a 'matching' output from another compile generates the same <gid-number>, and

  - a non-matching output (say because of #defines) generates a different <gid-number>.

It may be useful to think of a <gid-number> as a kind of "digital signature" that allows a fast test for the equality of two section groups.

So, for example, the section group corresponding to file `wa.h` above is given the name `my.compiler.company.cpp.wa.h.123456`.

E.2.2   Debugging Information Entry Names

Global labels for debugging information entries (need explained below) within a section group can be given names of the form

```
<prefix>.<file-designator>.<gid-number>.<die-number>
```

such as

```
my.compiler.company.wa.h.123456.987
```

where

- `<prefix>` distinguishes this as a DWARF debug info name, and should identify the producer and, when appropriate, the language.

- `<file-designator>` and `<gid-number>` are as above.

- `<die-number>` could be a number sequentially assigned to entities (tokens, perhaps) found during compilation.

In general, every point in the section group `.debug_info` that could be referenced from outside by **any** compilation unit must normally have an external name generated for it in the linker symbol table, whether the current compilation references all those points or not.

The completeness of the set of names generated is a quality of implementation issue.

It is up to the producer to ensure that if <die-numbers> in separate compilations would not match properly then a distinct <gid-number> is generated.

Note that only section groups that are designated as duplicate-removal-applies actually require the

        `<prefix>.<file-designator>.<gid-number>.<die-number>`

external labels for debugging information entries as all other section group sections can use 'local' labels (section-relative relocations).

(This is a consequence of separate compilation, not a rule imposed by this document).

Local labels use references with form DW_FORM_ref4 or DW_FORM_ref8 (these are affected by relocations so DW_FORM_ref_udata, DW_FORM_ref1 and DW_FORM_ref2 are normally not usable and DW_FORM_ref_addr is not necessary for a local label).

E.2.3   Use of DW_TAG_compile_unit versus DW_TAG_partial_unit

A section group compilation unit that uses DW_TAG_compile_unit is like any other compilation unit, in that its contents are evaluated by consumers as though it were an ordinary compilation unit.

A #include directive appearing outside any other declarations is a good candidate to be represented using DW_TAG_compile_unit. However, a #include appearing inside a C++ namespace declaration or a function, for example, is not because the entities included are not necessarily file level entities.

This also applies to Fortran INCLUDE lines when declarations are included into a procedure or module context.

Consequently a compiler must use DW_TAG_partial_unit (instead of DW_TAG_compile_unit) in a section group whenever the section group contents are not necessarily globally visible. This directs consumers to ignore that compilation unit when scanning top level declarations and definitions.

The DW_TAG_partial_unit compilation unit will be referenced from elsewhere and the referencing locations give the appropriate context for interpreting the partial compilation unit.

A DW_TAG_partial_unit may have, as appropriate, any of the attributes assigned to a DW_TAG_compile_unit.

E.2.4   Use of DW_TAG_imported_unit

A DW_TAG_imported_unit debugging information entry has an DW_AT_import attribute referencing a DW_TAG_compile_unit or DW_TAG_partial_unit debugging information entry.

A DW_TAG_imported_unit debugging information entry refers to a DW_TAG_compile_unit or DW_TAG_partial_unit debugging information entry to specify that the DW_TAG_compile_unit or DW_TAG_partial_unit contents logically appear at the point of the DW_TAG_imported_unit entry.

E.2.5   Use of DW_FORM_ref_addr

Use DW_FORM_ref_addr to reference from one compilation unit's debugging information entries to those of another compilation unit.

When referencing into a removable section group `.debug_info` from another `.debug_info` (from anywhere), the

```
<prefix>.<file-designator>.<gid-number>.<die-number>
```

name should be used for an external symbol and a relocation generated based on that name.

When referencing into a non-section group `.debug_info`, from another `.debug_info` (from anywhere) DW_FORM_ref_addr is still the form to be used, but a section-relative relocation generated by use of a non-exported name (often called an "internal name") may be used for references within the same object file.

E.3 Examples

This section provides several examples in order to have a concrete basis for discussion.

In these examples, the focus is on the arrangement of DWARF information into sections (specifically the `.debug_info` section) and the naming conventions used to achieve references into section groups. In practice, all of the examples that follow involve DWARF sections other than just `.debug_info` (for example, `.debug_line`, `.debug_aranges`, or others); however, only the `.debug_info` section is shown to keep the figures compact and easier to read.

The grouping of sections into a named set is shown, but the means for achieving this in terms of the underlying object language is not (and varies from system to system).

E.3.1   C++ Example

The C++ source in Figure 66 is used to illustrate the DWARF representation intended to allow duplicate elimination.

```
---- File wa.h ----
struct A {
    int i;
};

---- File wa.C ----
#include "wa.h";
int
f(A &a)
{
    return a.i + 2;
}
```

Figure 66. Duplicate elimination example #1: C++ source

Figure 67 shows the section group corresponding to the included file `wa.h`.

```
==== Section group name:
        my.compiler.company.cpp.wa.h.123456

== section .debug_info

DW.cpp.wa.h.123456.1:                   ! linker global symbol
        DW_TAG_compile_unit
            DW_AT_language(DW_LANG_C_plus_plus)
            ...                          ! other unit attributes
DW.cpp.wa.h.123456.2:                   ! linker global symbol
            DW_TAG_base_type
                DW_AT_name("int")
DW.cpp.wa.h.123456.3:                   ! linker global symbol
            DW_TAG_structure_type
DW.cpp.wa.h.123456.4:                   ! linker global symbol
                DW_TAG_member
                    DW_AT_name("i")
                    DW_AT_type(DW_FORM_refn to DW.cpp.wa.h.123456.2)
                        ! (This is a local reference, so the more
                        ! compact form DW_FORM_refn can be used)
```

Figure 67. Duplicate elimination example #1: DWARF section group

Figure 68 shows the "normal" DWARF sections, which are not part of any section group, and how they make use of the information in the section group shown above.

```
== section .text
        [generated code for function f]

== section .debug_info

        DW_TAG_compile_unit
.L1:                                     ! local (non-linker) symbol
            DW_TAG_reference_type
                DW_AT_type(reference to DW.cpp.wa.h.123456.3)
            DW_TAG_subprogram
                DW_AT_name("f")
                DW_AT_type(reference to DW.cpp.wa.h.123456.2)
                DW_TAG_variable
                    DW_AT_name("a")
                    DW_AT_type(reference to .L1)
            ...
```

Figure 68. Duplicate elimination example #1: primary compilation unit

This example uses DW_TAG_compile_unit for the section group, implying that the contents of the compilation unit are globally visible (in accordance with C++ language rules). DW_TAG_partial_unit is not needed for the same reason.

Other than linker support for section groups as such, DWARF duplicate elimination for C++ requires only DWARF debugging information entries or attributes that are already defined in DWARF Version 2.

E.3.2   Fortran Example

For a Fortran example, consider Figure 69.

```
---- File CommonStuff.fh ----

      IMPLICIT INTEGER(A-Z)
      COMMON /Common1/ C(100)
      PARAMETER(SEVEN = 7)


---- File Func.f ----

      FUNCTION FOO (N)
      INCLUDE 'CommonStuff.fh'
      FOO = C(N + SEVEN)
      RETURN
      END
```

Figure 69. Duplicate elimination example #2: Fortran source

Figure 70 shows the section group corresponding to the included file CommonStuff.fh.

```
==== Section group name:

        my.f90.company.f90.CommonStuff.fh.654321

== section .debug_info

DW.myf90.CommonStuff.fh.654321.1:                   ! linker global symbol
        DW_TAG_partial_unit
            ! ...compilation unit attributes, including...
            DW_AT_language(DW_LANG_Fortran90)
            DW_AT_identifier_case(DW_ID_case_insensitive)

DW.myf90.CommonStuff.fh.654321.2:                   ! linker global symbol
3$:         DW_TAG_array_type
                ! unnamed
                DW_AT_type(reference to DW.f90.F90$main.f.2)
                            ! base type INTEGER
                DW_TAG_subrange_type
                    DW_AT_type(reference to DW.f90.F90$main.f.2)
                            ! base type INTEGER)
                DW_AT_lower_bound(constant 1)
                DW_AT_upper_bound(constant 100)

DW.myf90.CommonStuff.fh.654321.3:                   ! linker global symbol
            DW_TAG_common_block
                DW_AT_name("Common1")
                DW_AT_location(Address of common block Common1)
                DW_TAG_variable
                    DW_AT_name("C")
                    DW_AT_type(reference to 3$)
                    DW_AT_location(address of C)

DW.myf90.CommonStuff.fh.654321.4:                   ! linker global symbol
            DW_TAG_constant
                DW_AT_name("SEVEN")
                DW_AT_type(reference to DW.f90.F90$main.f.2)
                            ! base type INTEGER
                DW_AT_const_value(constant 7)
```

Figure 70. Duplicate elimination example #2: DWARF section group

Figure 71 shows the sections for the primary compilation unit.

```
== section .text
        [code for function Foo]

== section .debug_info

        DW_TAG_compile_unit
            DW_TAG_subprogram
                DW_AT_name("Foo")
                DW_AT_type(reference to DW.f90.F90$main.f.2)
                                    ! base type INTEGER
                DW_TAG_imported_unit
                    DW_AT_import(reference to
                        DW.myf90.CommonStuff.fh.654321.1)
                DW_TAG_common_inclusion ! For Common1
                    DW_AT_common_reference(reference to
                        DW.myf90.CommonStuff.fh.654321.3)

                DW_TAG_variable ! For function result
                    DW_AT_name("Foo")
                    DW_AT_type(reference to DW.f90.F90$main.f.2)
                                    ! base type INTEGER
```

Figure 71. Duplicate elimination example #2: primary unit

A companion main program is shown in Figure 72.

```
    ---- File Main.f ----

        INCLUDE 'CommonStuff.fh'
        C(50) = 8
        PRINT *, 'Result = ', FOO(50 - SEVEN)
        END
```

Figure 72. Duplicate elimination example #2: companion source

That main program results in an object file that contained a duplicate of the section group named `my.f90.company.f90.CommonStuff.fh.654321` corresponding to the included file as well as the remainder of the main subprogram as shown in Figure 73.

```
== section .debug_info

        DW_TAG_compile_unit
            DW_AT_name(F90$main)
            DW_TAG_base_type
                DW_AT_name("INTEGER")
                DW_AT_encoding(DW_ATE_signed)
                DW_AT_byte_size(...)

            DW_TAG_base_type
                ...
            ...                             ! other base types
            DW_TAG_subprogram
                DW_AT_name("F90$main")
                DW_TAG_imported_unit
                    DW_AT_import(reference to
                        DW.myf90.CommonStuff.fh.654321.1)
                DW_TAG_common_inclusion       ! for Common1
                    DW_AT_common_reference(reference to
                        DW.myf90.CommonStuff.fh.654321.3)
                ...
```

Figure 73. Duplicate elimination example #2: companion DWARF

This example uses DW_TAG_partial_unit for the section group because the included declarations are not independently visible as global entities.

E.3.3   C Example

The C++ example in Section E.3.1 might appear to be equally valid as a C example. However, it is prudent to include a DW_TAG_imported_unit in the primary unit (see Figure 68) with an DW_AT_import attribute that refers to the proper unit in the section group.

The C rules for consistency of global (file scope) symbols across compilations are less strict than for C++; inclusion of the import unit attribute assures that the declarations of the proper section group are considered before declarations from other compilations.

E.4 Summary of Compression Techniques

E.4.1 #include compression

C++ has a much greater problem than C with the number and size of the headers included and the amount of data in each, but even with C there is substantial header file information duplication.

A reasonable approach is to put each header file in its own section group, using the naming rules mentioned above. The section groups are marked to ensure duplicate removal.

All data instances and code instances (even if they came from the header files above) are put into non-section group sections such as the base object file `.debug_info` section.

E.4.2 Eliminating function duplication

Function templates (C++) result in code for the same template instantiation being compiled into multiple archives or relocatable objects. The linker wants to keep only one of a given entity. The DWARF description, and everything else for this function, should be reduced to just a single copy.

For each such code group (function template in this example) the compiler assigns a name for the group which will match all other instantiations of this function but match nothing else. The second and subsequent definitions seen by the static linker are simply discarded.

References to other `.debug_info` sections follow the approach suggested above, but the naming rule might be slightly different in that the <file-designator> should be interpreted as a <function-designator>.

E.4.3 Single-function-per-DWARF-compilation-unit

Section groups can help make it easy for a linker to completely remove unused functions.

Such section groups are not marked for duplicate removal, since the functions are not duplicates of anything.

Each function is given a compilation unit and a section group. Each such compilation unit is complete, with its own text, data, and DWARF sections.

There will also be a compilation unit that has the file-level declarations and definitions. Other per-function compilation unit DWARF information (`.debug_info`) points to this common file-level compilation unit using DW_TAG_imported_unit.

Section groups can use DW_FORM_ref_addr and internal labels (section-relative relocations) to refer to the main object file sections, as the section groups here are either deleted as unused or kept. There is no possibility (aside from error) of a group from some other compilation being used in place of one of these groups.

E.4.4 Inlining and out-of-line-instances

Abstract instances and concrete-out-of-line instances may be put in distinct compilation units using section groups. This makes possible some useful duplicate DWARF elimination.

No special provision for eliminating class duplication resulting from template instantiation is made here, though nothing prevents eliminating such duplicates using section groups.

Index

DW_LNS_advance_pc, 95, 96, 148, 194
DW_LNS_const_add_pc, 96, 148
DW_LNS_copy, 95, 148
DW_LNS_fixed_advance_pc, 89, 96, 148, 194
DW_LNS_hi_user omission, 113
DW_LNS_lo_user omission, 113
DW_LNS_negate_stmt, 90, 95, 148
DW_LNS_set_basic_block, 95, 148
DW_LNS_set_column, 95, 148
DW_LNS_set_epilog_begin, 97
DW_LNS_set_epilogue_begin, 148
DW_LNS_set_file, 95, 148
DW_LNS_set_isa, 97, 148
DW_LNS_set_prologue_end, 96, 148
DW_MACINFO_define, 99, 101, 150
DW_MACINFO_end_file, 99, 100, 150
DW_MACINFO_start_file, 99, 100, 101, 150
DW_MACINFO_undef, 99, 101, 150
DW_MACINFO_vendor_ext, 99, 100, 150
DW_OP_abs, 17, 137
DW_OP_addr, 14, 23, 136
DW_OP_and, 18, 137, 183, 184
DW_OP_bra, 20, 138
DW_OP_breg0, 15, 139
DW_OP_breg1, 15, 139
DW_OP_breg11, 23
DW_OP_breg31, 15, 139
DW_OP_bregx, 15, 24, 139
DW_OP_call_ref, 20, 81, 109, 110, 139, 175
DW_OP_call2, 20, 81, 109, 110, 139
DW_OP_call4, 20, 81, 110, 139
DW_OP_const1s, 14, 136
DW_OP_const1u, 14, 136
DW_OP_const2s, 14, 136
DW_OP_const2u, 14, 136
DW_OP_const4s, 15, 136
DW_OP_const4u, 14, 136
DW_OP_const8s, 15, 136
DW_OP_const8u, 15, 136
DW_OP_consts, 15, 136
DW_OP_constu, 15, 136
DW_OP_deref, 16, 24, 136, 183, 184, 189
DW_OP_deref_size, 16, 139
DW_OP_div, 18, 137
DW_OP_drop, 16, 21, 137
DW_OP_dup, 16, 21, 136
DW_OP_eq, 19, 138
DW_OP_fbreg, 15, 23, 139
DW_OP_ge, 19, 138
DW_OP_gt, 19, 138
DW_OP_hi_user, 139
DW_OP_le, 19, 138
DW_OP_lit0, 14, 138

DW_OP_lit1, 14, 138, 183
DW_OP_lit2, 184
DW_OP_lit31, 14, 138
DW_OP_lo_user, 139
DW_OP_lt, 19, 138
DW_OP_minus, 18, 137
DW_OP_mod, 18, 137
DW_OP_mul, 18, 137
DW_OP_ne, 19, 137, 138, 189
DW_OP_neg, 18, 137, 189
DW_OP_nop, 21, 139
DW_OP_not, 18, 137
DW_OP_or, 18, 137
DW_OP_over, 16, 21, 137
DW_OP_pick, 16, 21, 137
DW_OP_piece, 20, 24, 139
DW_OP_plus, 18, 137, 183, 184, 189
DW_OP_plus_uconst, 18, 24, 137
DW_OP_push_object_address, 17, 70, 80, 110, 139, 183,
     184, 185, 186, 189
DW_OP_reg0, 22, 138
DW_OP_reg1, 22, 138
DW_OP_reg10, 24
DW_OP_reg3, 24
DW_OP_reg31, 22, 138
DW_OP_regx, 22, 23, 139
DW_OP_rot, 16, 21, 137
DW_OP_shl, 19, 137
DW_OP_shr, 19, 137, 138
DW_OP_shra, 19, 138
DW_OP_skip, 19, 138
DW_OP_swap, 16, 21, 137
DW_OP_xderef, 17, 137
DW_OP_xderef_size, 17, 139
DW_OP_xor, 19, 138
DW_ORD_col_major, 65, 145
DW_ORD_row_major, 65, 145
DW_TAG_access_declaration, 5, 69, 126, 155
DW_TAG_array_type, 5, 64, 125, 156, 183, 184, 188, 189,
     219
DW_TAG_base_type, 5, 61, 64, 126, 156, 180, 216, 221
DW_TAG_catch_block, 5, 56, 126, 156
DW_TAG_class_type, 5, 66, 72, 125, 157
DW_TAG_common_block, 5, 59, 125, 157, 219
DW_TAG_common_inclusion, 5, 46, 126, 157, 220, 221
DW_TAG_compile_unit, 5, 35, 118, 125, 158, 180, 213,
     214, 216, 217, 220, 221
DW_TAG_const_type, 5, 63, 64, 126, 158
DW_TAG_constant, 5, 57, 126, 158, 219
DW_TAG_dwarf_procedure, 5, 81, 127, 158
DW_TAG_entry_point, 5, 43, 125, 159
DW_TAG_enumeration_type, 5, 65, 74, 125, 159
DW_TAG_enumerator, 5, 75, 126, 159

file type entry, 79
flag class, 12, 122
formal parameter, 46
formal parameter entry, 57, 76
    in catch block, 56
    with default value, 58
formal type parameter. *See* template type parameter entry
Fortran, 1, 3, 36, 42, 43, 60, 76, 77, 80, 181
    common block, 46, 59
    main program, 44
    module (Fortran 90), 40
    use statement, 42, 43
frame base attribute, 46
frame description entry, 106
friend attribute, 69
friend entry, 69
function entry. *See* subroutine entry
fundamental type. *See* base type entry

G

global namespace. See namespace (C++), global

H

hidden indirection. *See* data location attribute
high PC attribute, 27, 30, 31, 36, 39, 45, 50, 55, 56
    and abstract instance, 49

I

identifier case attribute, 38
    encoding, *144*
implementing type (Java). *See* inheritance entry
import attribute, 39, 41, 42
imported declaration entry, 41
imported module entry, 42
imported unit entry, 39
incomplete class/structure/union, 67
incomplete declaration, *29*
inheritance entry, 68
initial length, 118
initial length field, 84, 85, 89, 105, 106
    encoding, 114
inline attribute, 48, 49
    encoding, *145*
inlined subprogram call
    examples, 199
inlined subprogram entry, 43, 50
    in concrete instance, 51
interface type entry, 68
is optional attribute, 58

J

Java, 3, 36, 66, 67, 68

L

label entry, 55
language attribute, 36, 65
language name encoding, *142*
LEB128
    examples, 134
    signed, decoding of, *178*
    signed, encoding as, 134, *177*
    unsigned, decoding of, *178*
    unsigned, encoding as, 134, *177*
lexical block entry, 55
line number information. *See also* statement list attribute
line number of declaration, 29
line number opcodes
    extended opcode encoding, 149
    standard opcode encoding, 148
lineptr class, 12, 122
Little Endian Base 128. *See* LEB128
little endian byte order
    effect of, 71
location attribute, 56, 57, 59, 81
    and abstract instance, 49
location description, 22
location expression, 22, 23. *See also* DWARF expression
    examples, 23
    use in location list, 24
location list, 22, 24, 46, 122, 140, 175
loclistptr class, 12, 122
lookup
    by address, 85
    by name, 84
low PC attribute, 27, 30, 31, 36, 39, 45, 50, 55, 56
    and abstract instance, 49
lower bound attribute, 77
    default, 77

M

macinfo types, 99
    encoding, *150*
macptr class, 12, 122
macro formal parameter list, 99
macro information, 99
macro information attribute, 37
member entry (data), 69
    as discriminant, 73
member function entry, 72
MIPS instruction set architecture, 86

Modula2, 1, 37, 40, 56
  definition module, 40
module entry, 39
mutable type entry, 63

N

name attribute, 30, 34, 36, 38, 39, 40, 41, 43, 48, 52, 55, 57,
    59, 60, 61, 62, 63, 64, 66, 68, 69, 73, 74, 75, 76, 77, 78,
    79, 84
namelist entry, 60
namelist item attribute, 60
namelist item entry, 60
namespace (C++), 40
  alias, 42
  example, 190
  global, 41
  std, 41
  unnamed, 41
  using declaration, 42
  using directive, 42
namespace declaration entry, 40
namespace extension entry, 40
nested abstract instance, 53
nested concrete inline instance, 53
non-contiguous address ranges, 31
non-defining declaration, 29
normal compilation unit, 35

O

optional parameter, 58
ordering attribute, 65
  encoding, *145*
out-of-line instance. *See* concrete out-of-line instance

P

packed type entry, 63
parameter. *See* macro formal parameter list. *See* this
    parameter. *See* variable parameter attribute. *See* optional
    parameter attribute. *See* unspecified parameters entry.
    See template value parameter entry. *See* template type
    parameter entry. *See* formal parameter entry
partial compilation unit, 35
Pascal, 1, 37, 56, 63, 66, 76, 79
pointer to member type entry, 78
pointer type entry, 63
priority attribute, 40
producer attribute, 37
prototyped attribute, 45, 76

R

range list, 31, 152, 175
rangelistptr class, 12, 123
ranges attribute, 27, 31, 36, 39, 45, 50, 55, 56
  and abstract instance, 49
reference class, 12, 123
reference type entry, 63
renamed declaration. *See* imported declaration entry
restricted type entry, 63
return address attribute, 46
  and abstract instance, 49
return type of subroutine, 45

S

sbyte, 83, 90, 154
section group, 210, 213, 215, 216, 218
  name, 212
section length
  in .debug_aranges header, 85
  in .debug_pubnames header, 84
  in .debug_pubtypes header, 84
  use in headers, 116
section offset
  alignment of, 153
  in .debug_info header, 118
  in .debug_pubnames header, 84
  in .debug_pubnames offset/name pair, 84
  in .debug_pubtypes header, 84
  in .debug_pubtypes name/offset pair, 84
  in class lineptr value, 122
  in class loclistptr value, 122
  in class macptr value, 122
  in class rangelistptr value, 123
  in class reference value, 123
  in class string value, 124
  in FDE header, 106
  in macro information attribute, 37
  in statement list attribute, 37
  use in headers, 116
segment attribute, 27, 45
  and abstract instance, 49
  and data segment, 58
segmented addressing. *See* address space
set type entry, 76
sibling attribute, 13
specification attribute, 29, 58, 67, 72
start scope attribute, 59, 61
  and abstract instance, 49
statement list attribute, 37
static link attribute, 47
stride attribute, 75, 78

T

U

V

W