DWARF Debugging Information Format
Version 5

DWARF Debugging Information Format Committee

http://www.dwarfstd.org

February 13, 2017

Copyright

Foreword

The DWARF Debugging Information Format Committee was originally organized in 1988 as the Programming Languages Special Interest Group (PLSIG) of Unix International, Inc., a trade group organized to promote Unix System V Release 4 (SVR4).

PLSIG drafted a standard for DWARF Version 1, compatible with the DWARF debugging format used at the time by SVR4 compilers and debuggers from AT&T. This was published as Revision 1.1.0 on October 6, 1992. PLSIG also designed the DWARF Version 2 format, which followed the same general philosophy as Version 1, but with significant new functionality and a more compact, though incompatible, encoding. An industry review draft of DWARF Version 2 was published as Revision 2.0.0 on July 27, 1993.

Unix International dissolved shortly after the draft of Version 2 was released; no industry comments were received or addressed, and no final standard was released. The committee mailing list was hosted by OpenGroup (formerly XOpen).

The Committee reorganized in October, 1999, and met for the next several years to address issues that had been noted with DWARF Version 2 as well as to add a number of new features. In mid-2003, the Committee became a workgroup under the Free Standards Group (FSG), an industry consortium chartered to promote open standards. DWARF Version 3 was published on December 20, 2005, following industry review and comment.

The DWARF Committee withdrew from the Free Standards Group in February, 2007, when FSG merged with the Open Source Development Labs to form The Linux Foundation, more narrowly focused on promoting Linux. The DWARF Committee has been independent since that time.

It is the intention of the DWARF Committee that migrating from an earlier version of the DWARF standard to the current version should be straightforward and easily accomplished. Almost all constructs from DWARF Version 2 onward have been retained unchanged in DWARF Version 5, although a few have been compatibly superseded by improved constructs which are more compact and/or more expressive.

This document was created using the LaTeX document preparation system.

The DWARF Debugging Information Format Committee

The DWARF Debugging Information Format Committee is open to compiler and debugger developers who have experience with source language debugging and debugging formats, and have an interest in promoting or extending the DWARF debugging format.

DWARF Committee members contributing to Version 5 are:

Todd Allen	Concurrent Computer
David Anderson, Associate Editor	
John Bishop	Intel
Ron Brender, Editor	
Andrew Cagney	
Soumitra Chatterjee	Hewlett-Packard Enterprise
Eric Christopher	Google
Cary Coutant	Google
John DelSignore	Rogue Wave
Michael Eager, Chair	Eager Consulting
Jini Susan George	Hewlett-Packard
Mathhew Gretton-Dan	ARM
Tommy Hoffner	Altera
Jakub Jelínek	Red Hat
Andrew Johnson	Linaro
Jason Merrill	Red Hat
Jason Molenda	Apple
Adrian Prantl	Apple
Hafiz Abid Qadeer	Mentor Graphics
Paul Robinson	Sony
Syamala Sarma	Hewlett-Packard
Keith Walker	ARM
Kendrick Wong	IBM
Brock Wyma	Intel
Jian Xu	IBM

For further information about DWARF or the DWARF Committee, see:

http://www.dwarfstd.org

How to Use This Document

This document is intended to be usable in online as well as traditional paper forms. Both online and paper forms include page numbers, a Table of Contents, a List of Figures, a List of Tables and an Index.

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.

Online Form

In the online form, blue text is used to indicate hyperlinks. Most hyperlinks link to the definition of a term or construct, or to a cited Section or Figure. However, attributes in particular are often used in more than one way or context so that there is no single definition; for attributes, hyperlinks link to the introductory table of all attributes which in turn contains hyperlinks for the multiple usages.

The occurrence of a DWARF name in its definition (or one of its definitions in the case of some attributes) is shown in red text. Other occurrences of the same name in the same or possibly following paragraphs are generally in normal text color.)

The Table of Contents, List of Figures, List of Tables and Index provide hyperlinks to the respective items and places.

Paper Form

In the traditional paper form, the appearance of the hyperlinks and definitions on a page of paper does not distract the eye because the blue hyperlinks and the color used for definitions are typically imaged by black and white printers in a manner nearly indistinguishable from other text. (Hyperlinks are not underlined for this same reason.)

Contents

List of Figures

List of Tables

(empty page)

Chapter 1

Introduction

This document defines a format for describing programs to facilitate user source level debugging. This description can be generated by compilers, assemblers and linkage editors. It can be used by debuggers and other tools. 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 compatibility.unittype

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. Aspects of individual languages, 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 Ada, C, C++, COBOL, and Fortran; it also covers the basic needs of various other languages.

This document describes DWARF Version 5, the fifth generation of debugging information based on the DWARF format. DWARF Version 5 extends DWARF Version 4 in a compatible manner.

The intended audience for this document is the developers of both producers and consumers of debugging information, typically 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 Chapters 2 through 6. Chapter 2 describes the overall structure of the information and attributes that are common to many or all of the different debugging information entries. Chapters 3, 4 and 5 describe the specific debugging information entries and how they communicate the necessary information about the source program to a debugger. Chapter 6 describes debugging information contained outside of the debugging information entries. The encoding of the DWARF information is presented in Chapter 7.

This organization closely follows that used in the DWARF Version 4 document. Except where needed to incorporate new material or to correct errors, the DWARF Version 4 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 consist only of explanatory or supplementary material, and are not part of the formal definition.

1.3 Objectives and Rationale

DWARF has had a set of objectives since its inception which have guided the design and evolution of the debugging format. A discussion of these objectives and the rationale behind them may help with an understanding of the DWARF Debugging Format.

Although DWARF Version 1 was developed in the late 1980’s as a format to support debugging C programs written for AT&T hardware running SVR4, DWARF Version 2 and later has evolved far beyond this origin. One difference between DWARF and other formats is that the latter are often specific to a particular language, architecture, and/or operating system.

1.3.1 Language Independence

DWARF is applicable to a broad range of existing procedural languages and is designed to be extensible to future languages. These languages may be considered to be "C-like" but the characteristics of C are not incorporated into DWARF Version 2 and later, unlike DWARF Version 1 and other debugging formats. DWARF abstracts concepts as much as possible so that the description can be used to describe a program in any language. As an example, the DWARF descriptions used to describe C functions, Pascal subroutines, and Fortran subprograms are all the same, with different attributes used to specify the differences between these similar programming language features.

On occasion, there is a feature which is specific to one particular language and which doesn't appear to have more general application. For these, DWARF has a description designed to meet the language requirements, although, to the extent possible, an effort is made to generalize the attribute. An example of this is the DW_TAG_condition debugging information entry, used to describe COBOL level 88 conditions, which is described in abstract terms rather than COBOL-specific terms. Conceivably, this TAG might be used with a different language which had similar functionality.

1.3.2 Architecture Independence

DWARF can be used with a wide range of processor architectures, whether byte or word oriented, linear or segmented, with any word or byte size. DWARF can be used with Von Neumann architectures, using a single address space for both code and data; Harvard architectures, with separate code and data address spaces; and potentially for other architectures such as DSPs with their idiosyncratic memory organizations. DWARF can be used with common register-oriented architectures or with stack architectures.

DWARF assumes that memory has individual units (words or bytes) which have unique addresses which are ordered. (Some architectures like the i386 can represent the same physical machine location with different segment and offset pairs. Identifying aliases is an implementation issue.)

1.3.3 Operating System Independence

DWARF is widely associated with SVR4 Unix and similar operating systems like BSD and Linux. DWARF fits well with the section organization of the ELF object file format. Nonetheless, DWARF attempts to be independent of either the OS or the object file format. There have been implementations of DWARF debugging data in COFF, Mach-O and other object file formats.

DWARF assumes that any object file format will be able to distinguish the various DWARF data sections in some fashion, preferably by name.

DWARF makes a few assumptions about functionality provided by the underlying operating system. DWARF data sections can be read sequentially and independently. Each DWARF data section is a sequence of 8-bit bytes, numbered starting with zero. The presence of offsets from one DWARF data section into other data sections does not imply that the underlying OS must be able to position files randomly; a data section could be read sequentially and indexed using the offset.

1.3.4 Compact Data Representation

The DWARF description is designed to be a compact file-oriented representation.

There are several encodings which achieve this goal, such as the TAG and attribute abbreviations or the line number encoding. References from one section to another, especially to refer to strings, allow these sections to be compacted to eliminate duplicate data.

There are multiple schemes for eliminating duplicate data or reducing the size of the DWARF debug data associated with a given file. These include COMDAT, used to eliminate duplicate function or data definitions, the split DWARF object files which allow a consumer to find DWARF data in files other than the executable, or the type units, which allow similar type definitions from multiple compilations to be combined.

In most cases, it is anticipated that DWARF debug data will be read by a consumer (usually a debugger) and converted into a more efficiently accessed internal representation. For the most part, the DWARF data in a section is not the same as this internal representation.

1.3.5 Efficient Processing

DWARF is designed to be processed efficiently, so that a producer (a compiler) can generate the debug descriptions incrementally and a consumer can read only the descriptions which it needs at a given time. The data formats are designed to be efficiently interpreted by a consumer.

As mentioned, there is a tension between this objective and the preceding one. A DWARF data representation which resembles an internal data representation may lead to faster processing, but at the expense of larger data files. This may also constrain the possible implementations.

1.3.6 Implementation Independence

DWARF attempts to allow developers the greatest flexibility in designing implementations, without mandating any particular design decisions. Issues which can be described as quality-of-implementation are avoided.

1.3.7 Explicit Rather Than Implicit Description

DWARF describes the source to object translation explicitly rather than using common practice or convention as an implicit understanding between producer and consumer. For example, where other debugging formats assume that a debugger knows how to virtually unwind the stack, moving from one stack frame to the next using implicit knowledge about the architecture or operating system, DWARF makes this explicit in the Call Frame Information description.

1.3.8 Avoid Duplication of Information

DWARF has a goal of describing characteristics of a program once, rather than repeating the same information multiple times. The string sections can be compacted to eliminate duplicate strings, for example. Other compaction schemes or references between sections support this. Whether a particular implementation is effective at eliminating duplicate data, or even attempts to, is a quality-of-implementation issue.

1.3.9 Leverage Other Standards

Where another standard exists which describes how to interpret aspects of a program, DWARF defers to that standard rather than attempting to duplicate the description. For example, C++ has specific rules for deciding which function to call depending name, scope, argument types, and other factors. DWARF describes the functions and arguments, but doesn't attempt to describe how one would be selected by a consumer performing any particular operation.

1.3.10 Limited Dependence on Tools

DWARF data is designed so that it can be processed by commonly available assemblers, linkers, and other support programs, without requiring additional functionality specifically to support DWARF data. This may require the implementer to be careful that they do not generate DWARF data which cannot be processed by these programs. Conversely, an assembler which can generate LEB128 (Little-Endian Base 128) values may allow the compiler to generate more compact descriptions, and a linker which understands the format of string sections can merge these sections. Whether or not an implementation includes these functions is a quality-of-implementation issue, not mandated by the DWARF specification.

1.3.11 Separate Description From Implementation

DWARF intends to describe the translation of a program from source to object, while neither mandating any particular design nor making any other design difficult. For example, DWARF describes how the arguments and local variables in a function are to be described, but doesn't specify how this data is collected or organized by a producer. Where a particular DWARF feature anticipates that it will be implemented in a certain fashion, informative text will suggest but not require this design.

1.3.12 Permissive Rather Than Prescriptive

The DWARF Standard specifies the meaning of DWARF descriptions. It does not specify in detail what a particular producer must generate for any source to object conversion. One producer may generate a more complete description than another, it may describe features in a different order (unless the standard explicitly requires a particular order), or it may use different abbreviations or compression methods. Similarly, DWARF does not specify exactly what a particular consumer should do with each part of the description, although we believe that the potential uses for each description should be evident.

DWARF is permissive, allowing different producers to generate different descriptions for the same source to object conversion, and permitting different consumers to provide more or less functionality or information to the user. This may result in debugging information being larger or smaller, compilers or debuggers which are faster or slower, and more or less functional. These are described as differences in quality-of-implementation.

Each producer conforming to the DWARF standard must follow the format and meaning as specified in the standard. As long as the DWARF description generated follows this specification, the producer is generating valid DWARF. For example, DWARF allows a producer to identify the end of a function prologue in the Line Information so that a debugger can stop at this location. A producer which does this is generating valid DWARF, as is another which doesn't. As another example, one producer may generate descriptions for variables which are moved from memory to a register in a certain range, while another may only describe the variable's location in memory. Both are valid DWARF descriptions, while a consumer using the former would be able to provide more accurate values for the variable while executing in that range than a consumer using the latter.

In this document, where the word "may" is used, the producer has the option to follow the description or not. Where the text says "may not", this is prohibited. Where the text says "should", this is advice about best practice, but is not a requirement.

1.3.13 Vendor Extensibility

This document does not attempt to cover all interesting languages or even to cover all of the possible debugging information needs for its primary target languages. 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 subset of the valid values for these constructs for vendor specific additions and defining related naming conventions. 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.

Where this specification provides a means for describing the source language, implementors are expected to adhere to that specification. For language features that are not supported, implementors may use existing attributes in novel ways or add vendor-defined attributes. Implementors who make extensions are strongly encouraged to design them to be compatible with this specification in the absence of those extensions.

The DWARF format is organized so that a consumer can skip over data which it does not recognize. This may allow a consumer to read and process files generated according to a later version of this standard or which contain vendor extensions, albeit possibly in a degraded manner.

1.4 Changes from Version 4 to Version 5

The following is a list of the major changes made to the DWARF Debugging Information Format since Version 4 was published. The list is not meant to be exhaustive.

- Eliminate the `.debug_types` section introduced in DWARF Version 4 and move its contents into the `.debug_info` section.
- Add support for collecting common DWARF information (debugging information entries and macro definitions) across multiple executable and shared files and keeping it in a single supplementary object file.
- Replace the line number program header format with a new format that provides the ability to use an MD5 hash to validate the source file version in use, allows pooling of directory and file name strings and makes provision for vendor-defined extensions. Also add a string section specific to the line number table (`.debug_line_str`) to properly support the common practice of stripping all DWARF sections except for line number information.
- Add a split object file and package representations to allow most DWARF information to be kept separate from an executable or shared image. This includes new sections `.debug_addr`, `.debug_str_offsets`, `.debug_abbrev.dwo`, `.debug_info.dwo`, `.debug_line.dwo`, `.debug_loclists.dwo`, `.debug_macro.dwo`, `.debug_str.dwo`, `.debug_str_offsets.dwo`, `.debug_cu_index` and `.debug_tu_index` together with new forms of attribute value for referencing these sections. This enhances DWARF support by reducing executable program size and by improving link times.
- Replace the `.debug_macinfo` macro information representation with with a `.debug_macro` representation that can potentially be much more compact.

- Replace the `.debug_pubnames` and `.debug_pubtypes` sections with a single and more functional name index section, `.debug_names`.
- Replace the location list and range list sections (`.debug_loc` and `.debug_ranges`, respectively) with new sections (`.debug_loclists` and `.debug_rnglists`) and new representations that save space and processing time by eliminating most related object file relocations.
- Add a new debugging information entry (DW_TAG_call_site), related attributes and DWARF expression operators to describe call site information, including identification of tail calls and tail recursion.
- Add improved support for FORTRAN assumed rank arrays (DW_TAG_generic_subrange), dynamic rank arrays (DW_AT_rank) and co-arrays (DW_TAG_coarray_type).
- Add new operations that allow support for a DWARF expression stack containing typed values.
- Add improved support for the C++: `auto` return type, deleted member functions (DW_AT_deleted), as well as defaulted constructors and destructors (DW_AT_defaulted).
- Add a new attribute (DW_AT_noreturn), to identify a subprogram that does not return to its caller.
- Add language codes for C 2011, C++ 2003, C++ 2011, C++ 2014, Dylan, Fortran 2003, Fortran 2008, Go, Haskell, Julia, Modula 3, Ocaml, OpenCL, Rust and Swift.
- Numerous other more minor additions to improve functionality and performance.

DWARF Version 5 is compatible with DWARF Version 4 except as follows:

- The compilation unit header (in the `.debug_info` section) has a new `unit_type` field. In addition, the `debug_abbrev_offset` and `address_size` fields are reordered.
- New operand forms for attribute values are defined (DW_FORM_addrx, DW_FORM_addrx1, DW_FORM_addrx2, DW_FORM_addrx3, DW_FORM_addrx4, DW_FORM_data16, DW_FORM_implicit_const, DW_FORM_line_strp, DW_FORM_loclistx, DW_FORM_rnglistx, DW_FORM_ref_sup4, DW_FORM_ref_sup8, DW_FORM_strp_sup, DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 and DW_FORM_strx4.

Because a pre-DWARF Version 5 consumer will not be able to interpret these even to ignore and skip over them, new forms must be considered incompatible additions.

- The line number table header is substantially revised.
- The `.debug_loc` and `.debug_ranges` sections are replaced by new `.debug_loclists` and `.debug_rnglists` sections, respectively. These new sections have a new (and more efficient) list structure. Attributes that reference the predecessor sections must be interpreted differently to access the new sections. The new sections encode the same information as their predecessors, except that a new default location list entry is added.
- In a string type, the DW_AT_byte_size attribute is re-defined to always describe the size of the string type. (Previously it described the size of the optional string length data field if the DW_AT_string_length attribute was also present.) In addition, the DW_AT_string_length attribute may now refer directly to an object that contains the length value.

While not strictly an incompatibility, the macro information representation is completely new; further, producers and consumers may optionally continue to support the older representation. While the two representations cannot both be used in the same compilation unit, they can co-exist in executable or shared images.

Similar comments apply to replacement of the `.debug_pubnames` and `.debug_pubtypes` sections with the new `.debug_names` section.

1.5 Changes from Version 3 to Version 4

The following is a list of the major changes made to the DWARF Debugging Information Format since Version 3 was published. The list is not meant to be exhaustive.

- Reformulate Section 2.6 (Location Descriptions) to better distinguish DWARF location descriptions, which compute the location where a value is found (such as an address in memory or a register name) from DWARF expressions, which compute a final value (such as an array bound).
- Add support for bundled instructions on machine architectures where instructions do not occupy a whole number of bytes.
- Add a new attribute form for section offsets, DW_FORM_sec_offset, to replace the use of DW_FORM_data4 and DW_FORM_data8 for section offsets.

- Add an attribute, DW_AT_main_subprogram, to identify the main subprogram of a program.
- Define default array lower bound values for each supported language.
- Add a new technique using separate type units, type signatures and COMDAT sections to improve compression and duplicate elimination of DWARF information.
- Add support for new C++ language constructs, including rvalue references, generalized constant expressions, Unicode character types and template aliases.
- Clarify and generalize support for packed arrays and structures.
- Add new line number table support to facilitate profile based compiler optimization.
- Add additional support for template parameters in instantiations.
- Add support for strongly typed enumerations in languages (such as C++) that have two kinds of enumeration declarations.
- Add the option for the DW_AT_high_pc value of a program unit or scope to be specified as a constant offset relative to the corresponding DW_AT_low_pc value.

DWARF Version 4 is compatible with DWARF Version 3 except as follows:

- DWARF attributes that use any of the new forms of attribute value representation (for section offsets, flag compression, type signature references, and so on) cannot be read by DWARF Version 3 consumers because the consumer will not know how to skip over the unexpected form of data.
- DWARF frame and line number table sections include additional fields that affect the location and interpretation of other data in the section.

1.6 Changes from Version 2 to Version 3

The following is a list of the major differences between Version 2 and Version 3 of the DWARF Debugging Information Format. The list is not meant to be exhaustive.

- Make provision for DWARF information files that are larger than 4 GBytes.
- Allow attributes to refer to debugging information entries in other shared libraries.

- Add support for Fortran 90 modules as well as allocatable array and pointer types.
- Add additional base types for C (as revised for 1999).
- Add support for Java and COBOL.
- Add namespace support for C++.
- Add an optional section for global type names (similar to the global section for objects and functions).
- Adopt UTF-8 as the preferred representation of program name strings.
- Add improved support for optimized code (discontiguous scopes, end of prologue determination, multiple section code generation).
- Improve the ability to eliminate duplicate DWARF information during linking.

DWARF Version 3 is 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 on page 196).
- 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 on page 196 and 7.5.4 on page 207).
- The return_address_register field in a Common Information Entry record for call frame information is changed to unsigned LEB representation (see Section 6.4.1 on page 172).

1.7 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 compared to the DWARF Version 1 may be found in the DWARF Version 2 document.

(empty page)

Chapter 2

General Description

2.1 The Debugging Information Entry (DIE)

DWARF uses a series of debugging information entries (DIEs) to define a low-level representation of a source program. Each debugging information entry consists of an identifying tag and a series of attributes. An entry, or group of entries together, provide a description of a corresponding entity in the source program. The tag specifies the class to which an entry belongs and the attributes define the specific characteristics of the entry.

The set of tag names is listed in Table 2.1 on the following page. The debugging information entries they identify are described in Chapters 3, 4 and 5.

The debugging information entry descriptions in Chapters 3, 4 and 5 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 many 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 are contained in the `.debug_info` and/or `.debug_info.dwo` sections of an object file.

Optionally, debugging information may be partitioned such that the majority of the debugging information can remain in individual object files without being processed by the linker. See Section 7.3.2 on page 187 and Appendix F on page 391 for details.

Table 2.1: Tag names

DW_TAG_access_declaration	DW_TAG_module
DW_TAG_array_type	DW_TAG_namelist
DW_TAG_atomic_type	DW_TAG_namelist_item
DW_TAG_base_type	DW_TAG_namespace
DW_TAG_call_site	DW_TAG_packed_type
DW_TAG_call_site_parameter	DW_TAG_partial_unit
DW_TAG_catch_block	DW_TAG_pointer_type
DW_TAG_class_type	DW_TAG_ptr_to_member_type
DW_TAG_coarray_type	DW_TAG_reference_type
DW_TAG_common_block	DW_TAG_restrict_type
DW_TAG_common_inclusion	DW_TAG_rvalue_reference_type
DW_TAG_compile_unit	DW_TAG_set_type
DW_TAG_condition	DW_TAG_shared_type
DW_TAG_const_type	DW_TAG_skeleton_unit
DW_TAG_constant	DW_TAG_string_type
DW_TAG_dwarf_procedure	DW_TAG_structure_type
DW_TAG_dynamic_type	DW_TAG_subprogram
DW_TAG_entry_point	DW_TAG_subrange_type
DW_TAG_enumeration_type	DW_TAG_subroutine_type
DW_TAG_enumerator	DW_TAG_template_alias
DW_TAG_file_type	DW_TAG_template_type_parameter
DW_TAG_formal_parameter	DW_TAG_template_value_parameter
DW_TAG_friend	DW_TAG_thrown_type
DW_TAG_generic_subrange	DW_TAG_try_block
DW_TAG_immutable_type	DW_TAG_typedef
DW_TAG_imported_declaration	DW_TAG_type_unit
DW_TAG_imported_module	DW_TAG_union_type
DW_TAG_imported_unit	DW_TAG_unspecified_parameters
DW_TAG_inheritance	DW_TAG_unspecified_type
DW_TAG_inlined_subroutine	DW_TAG_variable
DW_TAG_interface_type	DW_TAG_variant
DW_TAG_label	DW_TAG_variant_part
DW_TAG_lexical_block	DW_TAG_volatile_type
DW_TAG_member	DW_TAG_with_stmt

As a further option, debugging information entries and other debugging information that are the same in multiple executable or shared object files may be found in a separate supplementary object file that contains supplementary debug sections. See Section 7.3.6 on page 194 for further details.

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 attributes are listed in Table 2.2 following.

Table 2.2: Attribute names

Attribute[*]	**Usage**
DW_AT_abstract_origin	Inline instances of inline subprograms
	Out-of-line instances of inline subprograms
DW_AT_accessibility	Access declaration (C++, Ada)
	Accessibility of base or inherited class (C++)
	Accessibility of data member or member function
DW_AT_address_class	Pointer or reference types
	Subroutine or subroutine type
DW_AT_addr_base	Base offset for address table
DW_AT_alignment	Non-default alignment of type, subprogram or variable
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_binary_scale	Binary scale factor for fixed-point type
DW_AT_bit_size	Size of a base type in bits
	Size of a data member in bits

Continued on next page

[*]Links for attributes come to the left column of this table; links in the right column "fan-out" to one or more descriptions.

Attribute[*]	Identifies or Specifies
DW_AT_bit_stride	Array element stride (of array type)
	Subrange stride (dimension of array type)
	Enumeration stride (dimension of array type)
DW_AT_byte_size	Size of a data object or data type in bytes
DW_AT_byte_stride	Array element stride (of array type)
	Subrange stride (dimension of array type)
	Enumeration stride (dimension of array type)
DW_AT_call_all_calls	All tail and normal calls in a subprogram are described by call site entries
DW_AT_call_all_source_calls	All tail, normal and inlined calls in a subprogram are described by call site and inlined subprogram entries
DW_AT_call_all_tail_calls	All tail calls in a subprogram are described by call site entries
DW_AT_call_column	Column position of inlined subroutine call
	Column position of call site of non-inlined call
DW_AT_call_data_location	Address of the value pointed to by an argument passed in a call
DW_AT_call_data_value	Value pointed to by an argument passed in a call
DW_AT_call_file	File containing inlined subroutine call
	File containing call site of non-inlined call
DW_AT_call_line	Line number of inlined subroutine call
	Line containing call site of non-inlined call
DW_AT_call_origin	Subprogram called in a call
DW_AT_call_parameter	Parameter entry in a call
DW_AT_call_pc	Address of the call instruction in a call
DW_AT_call_return_pc	Return address from a call
DW_AT_call_tail_call	Call is a tail call
DW_AT_call_target	Address of called routine in a call

Continued on next page

[*]Links for attributes come to the left column of this table; links in the right column "fan-out" to one or more descriptions.

Attribute[*]	Identifies or Specifies
DW_AT_call_target_clobbered	Address of called routine, which may be clobbered, in a call
DW_AT_call_value	Argument value passed in a call
DW_AT_calling_convention	Calling convention for subprograms
	Calling convention for types
DW_AT_common_reference	Common block usage
DW_AT_comp_dir	Compilation directory
DW_AT_const_expr	Compile-time constant object
	Compile-time constant function
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_bit_offset	Base type bit location
	Data member bit location
DW_AT_data_location	Indirection to actual data
DW_AT_data_member_location	Data member location
	Inherited member location
DW_AT_decimal_scale	Decimal scale factor
DW_AT_decimal_sign	Decimal sign representation
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_defaulted	Whether a member function has been declared as default
DW_AT_default_value	Default value of parameter
DW_AT_deleted	Whether a member has been declared as deleted
DW_AT_description	Artificial name or description

Continued on next page

[*]Links for attributes come to the left column of this table; links in the right column "fan-out" to one or more descriptions.

Attribute[*]	**Identifies or Specifies**
DW_AT_digit_count	Digit count for packed decimal or numeric string type
DW_AT_discr	Discriminant of variant part
DW_AT_discr_list	List of discriminant values
DW_AT_discr_value	Discriminant value
DW_AT_dwo_name	Name of split DWARF object file
DW_AT_elemental	Elemental property of a subroutine
DW_AT_encoding	Encoding of base type
DW_AT_endianity	Endianity of data
DW_AT_entry_pc	Entry address of a scope (compilation unit, subprogram, and so on)
DW_AT_enum_class	Type safe enumeration definition
DW_AT_explicit	Explicit property of member function
DW_AT_export_symbols	Export (inline) symbols of namespace
	Export symbols of a structure, union or class
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

Continued on next page

[*] Links for attributes come to the left column of this table; links in the right column "fan-out" to one or more descriptions.

Attribute*[*]	**Identifies or Specifies**
DW_AT_linkage_name	Object file linkage name of an entity
DW_AT_location	Data object location
DW_AT_loclists_base	Location lists base
DW_AT_low_pc	Code address or range of addresses
	Base address of scope
DW_AT_lower_bound	Lower bound of subrange
DW_AT_macro_info	Macro preprocessor information (legacy)
	(reserved for coexistence with DWARF Version 4 and earlier)
DW_AT_macros	Macro preprocessor information
	(`#define`, `#undef`, and so on in C, C++ and similar languages)
DW_AT_main_subprogram	Main or starting subprogram
	Unit containing main or starting subprogram
DW_AT_mutable	Mutable property of member data
DW_AT_name	Name of declaration
	Path name of compilation source
DW_AT_namelist_item	Namelist item
DW_AT_noreturn	“no return” property of a subprogram
DW_AT_object_pointer	Object (`this`, `self`) pointer of member function
DW_AT_ordering	Array row/column ordering
DW_AT_picture_string	Picture string for numeric string type
DW_AT_priority	Module priority
DW_AT_producer	Compiler identification
DW_AT_prototyped	Subroutine prototype
DW_AT_pure	Pure property of a subroutine
DW_AT_ranges	Non-contiguous range of code addresses
DW_AT_rank	Dynamic number of array dimensions
DW_AT_recursive	Recursive property of a subroutine
DW_AT_reference	&-qualified non-static member function *(C++)*

Continued on next page

[*]Links for attributes come to the left column of this table; links in the right column "fan-out" to one or more descriptions.

Attribute[*]	**Identifies or Specifies**
DW_AT_return_addr	Subroutine return address save location
DW_AT_rnglists_base	Base offset for range lists
DW_AT_rvalue_reference	&&-qualified non-static member function *(C++)*
DW_AT_segment	Addressing information
DW_AT_sibling	Debugging information entry relationship
DW_AT_small	Scale factor for fixed-point type
DW_AT_signature	Type signature
DW_AT_specification	Incomplete, non-defining, or separate declaration corresponding to a declaration
DW_AT_start_scope	Reduced scope of declaration
DW_AT_static_link	Location of uplevel frame
DW_AT_stmt_list	Line number information for unit
DW_AT_string_length	String length of string type
DW_AT_string_length_bit_size	Size of string length of string type
DW_AT_string_length_byte_size	Size of string length of string type
DW_AT_str_offsets_base	Base of string offsets table
DW_AT_threads_scaled	Array bound THREADS scale factor (UPC)
DW_AT_trampoline	Target subroutine
DW_AT_type	Type of call site
	Type of string type components
	Type of subroutine return
	Type of declaration
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 attribute
DW_AT_visibility	Visibility of declaration
DW_AT_vtable_elem_location	Virtual function vtable slot

[*]Links for attributes come to the left column of this table; links in the right column "fan-out" to one or more descriptions.

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 example, 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, including fixed length data of one, two, four, eight or 16 bytes in size, 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 belong to one of the classes shown in Table 2.3 following.

Table 2.3: Classes of attribute value

Attribute Class	**General Use and Encoding**
address	Refers to some location in the address space of the described program.
addrptr	Specifies a location in the DWARF section that holds a series of machine address values. Certain attributes use one of these addresses by indexing relative to this location.
block	An arbitrary number of uninterpreted bytes of data. The number of data bytes may be implicit from context or explicitly specified by an initial unsigned LEB128 value (see Section 7.6 on page 221) that precedes that number of data bytes.
constant	One, two, four, eight or sixteen bytes of uninterpreted data, or data encoded in the variable length format known as LEB128 (see Section 7.6 on page 221).
exprloc	A DWARF expression for a value or a location in the address space of the described program. A leading unsigned LEB128 value (see Section 7.6 on page 221) specifies the number of bytes in the expression.
flag	A small constant that indicates the presence or absence of an attribute.
lineptr	Specifies a location in the DWARF section that holds line number information.
loclist, loclistsptr	Specifies a location in the DWARF section that holds location lists, which describe objects whose location can change during their lifetime.

Continued on next page

Attribute Class	General Use and Encoding
macptr	Specifies a location in the DWARF section that holds macro definition information.
reference	Refers to one of the debugging information entries that describe the program. There are four 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. The third type of reference is an indirect reference to a type definition using an 8-byte signature for that type. The fourth type of reference is a reference from within the `.debug_info` section of the executable or shared object file to a debugging information entry in the `.debug_info` section of a supplementary object file.
rnglist, rnglistsptr	Specifies a location in the DWARF section that holds non-contiguous address ranges.
string	A null-terminated sequence of zero or more (non-null) bytes. Data in this class are generally printable strings. Strings may be represented directly in the debugging information entry or as an offset in a separate string table.
stroffsetsptr	Specifies a location in the DWARF section that holds a series of offsets into the DWARF section that holds strings. Certain attributes use one of these offsets by indexing relative to this location. The resulting offset is then used to index into the DWARF string section.

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, for example, to describe the static block structure within a source file, to show the members of a structure, union, or class, and to associate declarations with source files or source files with shared object files.

The ownership relationship 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 reference 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 on page 203). 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 Target Addresses

Addresses, bytes and bits in DWARF use the numbering and direction conventions that are appropriate to the current language on the target system.

Many places in this document refer to the size of an address on the target architecture (or equivalently, target machine) to which a DWARF description applies. For processors which can be configured to have different address sizes or different instruction sets, the intent is to refer to the configuration which is either the default for that processor or which is specified by the object file or executable file which contains the DWARF information.

For example, if a particular target architecture supports both 32-bit and 64-bit addresses, the compiler will generate an object file which specifies that it contains executable code generated for one or the other of these address sizes. In that case, the DWARF debugging information contained in this object file will use the same address size.

2.5 DWARF Expressions

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

A DWARF expression is encoded as a stream of operations, each consisting of an opcode followed by zero or more literal operands. The number of operands is implied by the opcode.

In addition to the general operations that are defined here, operations that are specific to location descriptions are defined in Section 2.6 on page 38.

2.5.1 General Operations

Each general operation represents a postfix operation on a simple stack machine. Each element of the stack has a type and a value, and can represent a value of any supported base type of the target machine. Instead of a base type, elements can have a generic type, which is an integral type that has the size of an address on the target machine and unspecified signedness. 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, the desired value itself, and so on).

The generic type is the same as the unspecified type used for stack operations defined in DWARF Version 4 and before.

2.5.1.1 Literal Encodings

The following operations all push a value onto the DWARF stack. Operations other than DW_OP_const_type push a value with the generic type, and if the value of a constant in one of these operations is larger than can be stored in a single stack element, the value is truncated to the element size and the low-order bits are pushed on the stack.

1. **DW_OP_lit0**, **DW_OP_lit1**, . . . , **DW_OP_lit31**
   The DW_OP_lit<n> operations encode the unsigned literal values from 0 through 31, inclusive.
2. **DW_OP_addr**
   The DW_OP_addr operation has a single operand that encodes a machine address and whose size is the size of an address on the target machine.

3. **DW_OP_const1u**, **DW_OP_const2u**, **DW_OP_const4u**, **DW_OP_const8u**
   The single operand of a DW_OP_const<n>u operation provides a 1, 2, 4, or 8-byte unsigned integer constant, respectively.

4. **DW_OP_const1s**, **DW_OP_const2s**, **DW_OP_const4s**, **DW_OP_const8s**
   The single operand of a DW_OP_const<n>s operation provides a 1, 2, 4, or 8-byte signed integer constant, respectively.

5. **DW_OP_constu**
   The single operand of the DW_OP_constu operation provides an unsigned LEB128 integer constant.

6. **DW_OP_consts**
   The single operand of the DW_OP_consts operation provides a signed LEB128 integer constant.

7. **DW_OP_addrx**
   The DW_OP_addrx operation has a single operand that encodes an unsigned LEB128 value, which is a zero-based index into the `.debug_addr` section, where a machine address is stored. This index is relative to the value of the DW_AT_addr_base attribute of the associated compilation unit.

8. **DW_OP_constx**
   The DW_OP_constx operation has a single operand that encodes an unsigned LEB128 value, which is a zero-based index into the `.debug_addr` section, where a constant, the size of a machine address, is stored. This index is relative to the value of the DW_AT_addr_base attribute of the associated compilation unit.

   The DW_OP_constx operation is provided for constants that require link-time relocation but should not be interpreted by the consumer as a relocatable address (for example, offsets to thread-local storage).

9. **DW_OP_const_type**
   The DW_OP_const_type operation takes three operands. The first operand is an unsigned LEB128 integer that represents the offset of a debugging information entry in the current compilation unit, which must be a DW_TAG_base_type entry that provides the type of the constant provided. The second operand is 1-byte unsigned integer that specifies the size of the constant value, which is the same as the size of the base type referenced by the first operand. The third operand is a sequence of bytes of the given size that is interpreted as a value of the referenced type.

   While the size of the constant can be inferred from the base type definition, it is encoded explicitly into the operation so that the operation can be parsed easily without reference to the `.debug_info` section.

2.5.1.2 Register Values

The following operations push a value onto the stack that is either the contents of a register or the result of adding the contents of a register to a given signed offset. DW_OP_regval_type pushes the contents of the register together with the given base type, while the other operations push the result of adding the contents of a register to a given signed offset together with the generic type.

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.

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 contents of the specified register.

3. **DW_OP_bregx**
   The DW_OP_bregx operation provides the sum of two values specified by its two operands. The first operand is a register number which is specified by an unsigned LEB128 number. The second operand is a signed LEB128 offset.

4. **DW_OP_regval_type**
   The DW_OP_regval_type operation provides the contents of a given register interpreted as a value of a given type. The first operand is an unsigned LEB128 number, which identifies a register whose contents is to be pushed onto the stack. The second operand is an unsigned LEB128 number that represents the offset of a debugging information entry in the current compilation unit, which must be a DW_TAG_base_type entry that provides the type of the value contained in the specified register.

2.5.1.3 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.

Each entry on the stack has an associated type.

1. **DW_OP_dup**
   The DW_OP_dup operation duplicates the value (including its type identifier) at the top of the stack.
2. **DW_OP_drop**
   The DW_OP_drop operation pops the value (including its type identifier) at the top of the stack.
3. **DW_OP_pick**
   The single operand of the DW_OP_pick operation provides a 1-byte index. A copy of the stack entry (including its type identifier) with the specified index (0 through 255, inclusive) is pushed onto 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 (including its type identifier) becomes the second stack entry, and the second entry (including its type identifier) 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 (including its type identifier) becomes the third stack entry, the second entry (including its type identifier) becomes the top of the stack, and the third entry (including its type identifier) 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 popped value must have an integral type. The value retrieved from that address is pushed, and has the generic type. 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 popped value must have an integral type. The value retrieved from that address is pushed, and has the generic type. 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 the generic type. The data retrieved is zero extended to the size of an address on the target machine before being pushed onto the expression stack.

9. **DW_OP_deref_type**
   The DW_OP_deref_type operation behaves like the DW_OP_deref_size operation: it pops the top stack entry and treats it as an address. The popped value must have an integral type. The value retrieved from that address is pushed together with a type identifier. In the DW_OP_deref_type operation, the size in bytes of the data retrieved from the dereferenced address is specified by the first operand. This operand is a 1-byte unsigned integral constant whose value which is the same as the size of the base type referenced by the second operand. The second operand is an unsigned LEB128 integer that represents the offset of a debugging information entry in the current compilation unit, which must be a DW_TAG_base_type entry that provides the type of the data pushed.

   While the size of the pushed value could be inferred from the base type definition, it is encoded explicitly into the operation so that the operation can be parsed easily without reference to the `.debug_info` section.

10. **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. Both of these entries must have integral type identifiers. The top two stack elements are popped, and a data item is retrieved through an implementation-defined address calculation and pushed as the new stack top together with the generic type identifier. The size of the data retrieved from the dereferenced address is the size of the generic type.

11. **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. Both of these entries must have integral type identifiers. The top two stack elements are popped, and 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 onto the expression stack together with the generic type identifier.

12. **DW_OP_xderef_type**
    The DW_OP_xderef_type operation behaves like the DW_OP_xderef_size operation: it pops the top two stack entries, treats them as an address and an address space identifier, and pushes the value retrieved. In the DW_OP_xderef_type operation, the size in bytes of the data retrieved from the dereferenced address is specified by the first operand. This operand is a 1-byte unsigned integral constant whose value value which is the same as the size of the base type referenced by the second operand. The second operand is an unsigned LEB128 integer that represents the offset of a debugging information entry in the current compilation unit, which must be a DW_TAG_base_type entry that provides the type of the data pushed.

13. **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 debugging information entry 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 explicit functionality (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 on page 292.

14. **DW_OP_form_tls_address**
    The DW_OP_form_tls_address operation pops a value from the stack, which must have an integral type identifier, translates this value into an address in the thread-local storage for a thread, and pushes the address onto the stack together with the generic type identifier. The meaning of the value on the top of the stack prior to this operation is defined by the run-time environment. If the run-time environment supports multiple thread-local storage blocks for a single thread, then the block corresponding to the executable or shared library containing this DWARF expression is used.

    Some implementations of C, C++, Fortran, and other languages, support a thread-local storage class. Variables with this storage class have distinct values and addresses in distinct threads, much as automatic variables have distinct values and addresses in each function invocation. Typically, there is a single block of storage containing all thread-local variables declared in the main executable, and a separate block for the variables declared in each shared library. Each thread-local variable can then be accessed in its block using an identifier. This identifier is typically an offset into the block and pushed onto the DWARF stack by one of the DW_OP_const<n><x> operations prior to the DW_OP_form_tls_address operation. Computing the address of the appropriate block can be complex (in some cases, the compiler emits a function call to do it), and difficult to describe using ordinary DWARF location descriptions. Instead of forcing complex thread-local storage calculations into the DWARF expressions, the DW_OP_form_tls_address allows the consumer to perform the computation based on the run-time environment.

15. **DW_OP_call_frame_cfa**
    The DW_OP_call_frame_cfa operation pushes the value of the CFA, obtained from the Call Frame Information (see Section 6.4 on page 171).

    Although the value of DW_AT_frame_base can be computed using other DWARF expression operators, in some cases this would require an extensive location list because the values of the registers used in computing the CFA change during a subroutine. If the Call Frame Information is present, then it already encodes such changes, and it is space efficient to reference that.

Examples illustrating many of these stack operations are found in Appendix D.1.2 on page 289.

2.5.1.4 Arithmetic and Logical Operations

The following provide arithmetic and logical operations. Operands of an operation with two operands must have the same type, either the same base type or the generic type. The result of the operation which is pushed back has the same type as the type of the operand(s).

If the type of the operands is the generic type, except as otherwise specified, the arithmetic operations perform addressing arithmetic, that is, unsigned arithmetic that is performed modulo one plus the largest representable address.

Operations other than DW_OP_abs, DW_OP_div, DW_OP_minus, DW_OP_mul, DW_OP_neg and DW_OP_plus require integral types of the operand (either integral base type or the generic type). 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 cannot 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, interprets it as a signed value and pushes its negation. If the negation cannot be represented, the result is undefined.

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 interpreted as the same type as the operand popped from the top of the stack and pushes the result.

    This operation is supplied specifically to be able to encode more field offsets in two bytes than can be done with "DW_OP_lit<n> DW_OP_plus."

12. **DW_OP_shl**
    The DW_OP_shl operation pops the top two stack entries, shifts the former second entry left (filling with zero bits) 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 a bitwise exclusive-or operation on the two, and pushes the result.

2.5.1.5 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, which have the same type, either the same base type or the generic type,

   - compare the operands:
     < former second entry >< relational operator >< former top entry >

   - 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 pushed value has the generic type.

   If the operands have the generic type, the comparisons are performed as signed operations.

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 forward or backward 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 forward or backward 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 DWARF procedure calls during evaluation of a DWARF expression or location description. 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 following). The operand is used as the offset of a debugging information entry in a `.debug_info` section which may be contained in an executable or shared object file other than that containing the operator. For references from one executable or shared object file 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 (see Section 7.5.4 on page 207).

   These operations transfer control of DWARF expression evaluation to the DW_AT_location attribute of the referenced debugging information entry. 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.

2.5.1.6 Type Conversions

The following operations provides for explicit type conversion.

1. **DW_OP_convert**
   The DW_OP_convert operation pops the top stack entry, converts it to a different type, then pushes the result. It takes one operand, which is an unsigned LEB128 integer that represents the offset of a debugging information entry in the current compilation unit, or value 0 which represents the generic type. If the operand is non-zero, the referenced entry must be a DW_TAG_base_type entry that provides the type to which the value is converted.

2. **DW_OP_reinterpret**
   The DW_OP_reinterpret operation pops the top stack entry, reinterprets the bits in its value as a value of a different type, then pushes the result. It takes one operand, which is an unsigned LEB128 integer that represents the offset of a debugging information entry in the current compilation unit, or value 0 which represents the generic type. If the operand is non-zero, the referenced entry must be a DW_TAG_base_type entry that provides the type to which the value is converted. The type of the operand and result type must have the same size in bits.

2.5.1.7 Special Operations

There are these special operations currently defined:

1. **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. **DW_OP_entry_value**
   The DW_OP_entry_value operation pushes the value that the described location held upon entering the current subprogram. It has two operands: an unsigned LEB128 length, followed by a block containing a DWARF expression or a register location description (see Section 2.6.1.1.3 on page 39). The length operand specifies the length in bytes of the block. If the block contains a DWARF expression, the DWARF expression is evaluated as if it had been evaluated upon entering the current subprogram. The DWARF expression assumes no values are present on the DWARF stack initially and results in exactly one value being pushed on the DWARF stack when completed. If the block contains a register location description, DW_OP_entry_value pushes the value that register had upon entering the current subprogram.

   DW_OP_push_object_address is not meaningful inside of this DWARF operation.

The register location description provides a more compact form for the case where the value was in a register on entry to the subprogram.

The values needed to evaluate DW_OP_entry_value could be obtained in several ways. The consumer could suspend execution on entry to the subprogram, record values needed by DW_OP_entry_value expressions within the subprogram, and then continue; when evaluating DW_OP_entry_value, the consumer would use these recorded values rather than the current values. Or, when evaluating DW_OP_entry_value, the consumer could virtually unwind using the Call Frame Information (see Section 6.4 on page 171) to recover register values that might have been clobbered since the subprogram entry point.

2.6 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. *Single location descriptions*, which are a language independent representation of addressing rules of arbitrary complexity built from DWARF expressions (See Section 2.5 on page 26) and/or other DWARF operations specific to describing locations. 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 during its lifetime.

2. *Location lists*, which are used to describe objects that have a limited lifetime or change their location during their lifetime. Location lists are described in Section 2.6.2 on page 43 below.

Location descriptions are distinguished in a context sensitive manner. As the value of an attribute, a location description is encoded using class exprloc and a location list is encoded using class loclist (which serves as an index into a separate section containing location lists).

2.6.1 Single Location Descriptions

A single location description is either:

1. A simple location description, representing an object which exists in one contiguous piece at the given location, or
2. A composite location description consisting of one or more simple location descriptions, each of which is followed by one composition operation. Each simple location description describes the location of one piece of the object; each composition operation describes which part of the object is located there. Each simple location description that is a DWARF expression is evaluated independently of any others.

2.6.1.1 Simple Location Descriptions

A simple location description consists of one contiguous piece or all of an object or value.

2.6.1.1.1 Empty Location Descriptions

An empty location description consists of a DWARF expression containing no operations. It represents a piece or all of an object that is present in the source but not in the object code (perhaps due to optimization).

2.6.1.1.2 Memory Location Descriptions

A memory location description consists of a non-empty DWARF expression (see Section 2.5 on page 26), whose value is the address of a piece or all of an object or other entity in memory.

2.6.1.1.3 Register Location Descriptions

A register location description consists of a register name operation, which represents a piece or all of an object located in a given register.

Register location descriptions describe an object (or a piece of an object) that resides in a register, while the opcodes listed in Section 2.5.1.2 on page 28 are used to describe an object (or a piece of an object) that is located in memory at an address that is contained in a register (possibly offset by some constant). A register location description must stand alone as the entire description of an object or a piece of an object.

The following DWARF operations can be used to specify a register location.

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.

These operations name a register location. To fetch the contents of a register, it is necessary to use one of the register based addressing operations, such as DW_OP_bregx (Section 2.5.1.2 on page 28).

2.6.1.1.4 Implicit Location Descriptions

An implicit location description represents a piece or all of an object which has no actual location but whose contents are nonetheless either known or known to be undefined.

The following DWARF operations may be used to specify a value that has no location in the program but is a known constant or is computed from other locations and values in the program.

1. **DW_OP_implicit_value**
   The DW_OP_implicit_value operation specifies an immediate value using two operands: an unsigned LEB128 length, followed by a sequence of bytes of the given length that contain the value.

2. **DW_OP_stack_value**
   The DW_OP_stack_value operation specifies that the object does not exist in memory but its value is nonetheless known and is at the top of the DWARF expression stack. In this form of location description, the DWARF expression represents the actual value of the object, rather than its location. The DW_OP_stack_value operation terminates the expression.

3. **DW_OP_implicit_pointer**
   An optimizing compiler may eliminate a pointer, while still retaining the value that the pointer addressed. DW_OP_implicit_pointer allows a producer to describe this value.

   The DW_OP_implicit_pointer operation specifies that the object is a pointer that cannot be represented as a real pointer, even though the value it would point to can be described. In this form of location description, the DWARF expression refers to a debugging information entry that represents the actual value of the object to which the pointer would point. Thus, a consumer of the debug information would be able to show the value of the dereferenced pointer, even when it cannot show the value of the pointer itself.

   The DW_OP_implicit_pointer operation has two operands: a reference to a debugging information entry that describes the dereferenced object's value, and a signed number that is treated as a byte offset from the start of that value. The first operand is a 4-byte unsigned value in the 32-bit DWARF format, or an 8-byte unsigned value in the 64-bit DWARF format (see Section 7.4 on page 196). The second operand is a signed LEB128 number.

   The first operand is used as the offset of a debugging information entry in a `.debug_info` section, which may be contained in an executable or shared object file other than that containing the operator. For references from one executable or shared object file to another, the relocation must be performed by the consumer.

   The debugging information entry referenced by a DW_OP_implicit_pointer operation is typically a DW_TAG_variable or DW_TAG_formal_parameter entry whose DW_AT_location attribute gives a second DWARF expression or a location list that describes the value of the object, but the referenced entry may be any entry that contains a DW_AT_location or DW_AT_const_value attribute (for example, DW_TAG_dwarf_procedure). By using the second DWARF expression, a consumer can reconstruct the value of the object when asked to dereference the pointer described by the original DWARF expression containing the DW_OP_implicit_pointer operation.

DWARF location descriptions are intended to yield the ***location*** *of a value rather than the value itself. An optimizing compiler may perform a number of code transformations where it becomes impossible to give a location for a value, but it remains possible to describe the value itself. Section 2.6.1.1.3 on page 39 describes operators that can be used to describe the location of a value when that value exists in a register but not in memory. The operations in this section are used to describe values that exist neither in memory nor in a single register.*

2.6.1.2 Composite Location Descriptions

A composite location description describes an object or value which may be contained in part of a register or stored in more than one location. Each piece is described by a composition operation, which does not compute a value nor store any result on the DWARF stack. There may be one or more composition operations in a single composite location description. A series of such operations describes the parts of a value in memory address order.

Each composition operation is immediately preceded by a simple location description which describes the location where part of the resultant value is contained.

1. **DW_OP_piece**
   The DW_OP_piece operation 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 preceding simple location description. If the piece is located in a register, but does not occupy the entire register, the placement of the piece within that register is defined by the ABI.

   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 location description refers to.

2. **DW_OP_bit_piece**
   The DW_OP_bit_piece operation takes two operands. The first is an unsigned LEB128 number that gives the size in bits of the piece. The second is an unsigned LEB128 number that gives the offset in bits from the location defined by the preceding DWARF location description.

   Interpretation of the offset depends on the location description. If the location description is empty, the offset doesn't matter and the DW_OP_bit_piece operation describes a piece consisting of the given number of bits whose values are undefined. If the location is a register, the offset is from the least significant bit end of the register. If the location is a memory address, the DW_OP_bit_piece operation describes a sequence of bits relative to the location whose address is on the top of the DWARF stack using the bit numbering and direction conventions that are appropriate to the current language on the target system. If the location is any implicit value or stack value, the DW_OP_bit_piece operation describes a sequence of bits using the least significant bits of that value.

DW_OP_bit_piece is used instead of DW_OP_piece when the piece to be assembled into a value or assigned to is not byte-sized or is not at the start of a register or addressable unit of memory.

2.6.2 Location Lists

Location lists are used in place of location descriptions 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_loclists` or `.debug_loclists.dwo` (for split DWARF object files).

A location list is indicated by a location or other attribute whose value is of class loclist (see Section 7.5.5 on page 212).

This location list representation, the loclist class, and the related DW_AT_loclists_base attribute are new in DWARF Version 5. Together they eliminate most or all of the object language relocations previously needed for location lists.

A location list consists of a series of location list entries. Each location list entry is one of the following kinds:

- Bounded location description. This kind of entry provides a location description that specifies the location of an object that is valid over a lifetime bounded by a starting and ending address. The starting address is the lowest address of the address range over which the location is valid. The ending address is the address of the first location past the highest address of the address range. When the current PC is within the given range, the location description may be used to locate the specified object.

  There are several kinds of bounded location description entries which differ in the way that they specify the starting and ending addresses.

  The address ranges defined by the bounded location descriptions of a location list 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, and there is no following default location description, it is assumed that the object is not available for the portion of the range that is not covered.

- Default location description. This kind of entry provides a location description that specifies the location of an object that is valid when no bounded location description applies.

- Base address. This kind of entry provides an address to be used as the base address for beginning and ending address offsets given in certain kinds of bounded location description. The applicable base address of a bounded location description entry is the address specified by the closest preceding base address entry in the same location list. If there is no preceding base address entry, then the applicable base address defaults to the base address of the compilation unit (see Section 3.1.1 on page 60).

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

- End-of-list. This kind of entry marks the end of the location list.

A location list consists of a sequence of zero or more bounded location description or base address entries, optionally followed by a default location entry, and terminated by an end-of-list entry.

Each location list entry begins with a single byte identifying the kind of that entry, followed by zero or more operands depending on the kind.

In the descriptions that follow, these terms are used for operands:

- A counted location description operand consists of an unsigned ULEB integer giving the length of the location description (see Section 2.6.1 on page 39) that immediately follows.

- An address index operand is the index of an address in the `.debug_addr` section. This index is relative to the value of the DW_AT_addr_base attribute of the associated compilation unit. The address given by this kind of operand is not relative to the compilation unit base address.

- A target address operand is an address on the target machine. (Its size is the same as used for attribute values of class address, specifically, DW_FORM_addr.)

The following entry kinds are defined for use in both split or non-split units:

1. **DW_LLE_end_of_list**
   An end-of-list entry contains no further data.

   A series of this kind of entry may be used for padding or alignment purposes.

2. **DW_LLE_base_addressx**
   This is a form of base address entry that has one unsigned LEB128 operand. The operand value is an address index (into the `.debug_addr` section) that indicates the applicable base address used by subsequent DW_LLE_offset_pair entries.

3. **DW_LLE_startx_endx**
   This is a form of bounded location description entry that has two unsigned LEB128 operands. The operand values are address indices (into the `.debug_addr` section). These indicate the starting and ending addresses, respectively, that define the address range for which this location is valid. These operands are followed by a counted location description.

4. **DW_LLE_startx_length**
   This is a form of bounded location description that has two unsigned ULEB operands. The first value is an address index (into the `.debug_addr` section) that indicates the beginning of the address range over which the location is valid. The second value is the length of the range. These operands are followed by a counted location description.

5. **DW_LLE_offset_pair**
   This is a form of bounded location description entry that has two unsigned LEB128 operands. The values of these operands are the starting and ending offsets, respectively, relative to the applicable base address, that define the address range for which this location is valid. These operands are followed by a counted location description.

6. **DW_LLE_default_location**
   The operand is a counted location description which defines where an object is located if no prior location description is valid.

The following kinds of location list entries are defined for use only in non-split DWARF units:

7. **DW_LLE_base_address**
   A base address entry has one target address operand. This address is used as the base address when interpreting offsets in subsequent location list entries of kind DW_LLE_offset_pair.

8. **DW_LLE_start_end**
   This is a form of bounded location description entry that has two target address operands. These indicate the starting and ending addresses, respectively, that define the address range for which the location is valid. These operands are followed by a counted location description.

9. **DW_LLE_start_length**
   This is a form of bounded location description entry that has one target address operand value and an unsigned LEB128 integer operand value. The address is the beginning address of the range over which the location description is valid, and the length is the number of bytes in that range. These operands are followed by a counted location description.

2.7 Types of Program Entities

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, such as 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 Chapter 5 following for descriptions of the entries describing base types, user-defined types and type modifiers.

2.8 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 Table 2.4.

Table 2.4: Accessibility codes

DW_ACCESS_public
DW_ACCESS_private
DW_ACCESS_protected

2.9 Visibility of Declarations

Several languages (such as Modula-2) 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 Table 2.5 on the following page.

Table 2.5: Visibility codes

DW_VIS_local
DW_VIS_exported
DW_VIS_qualified

2.10 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 Table 2.6.

Table 2.6: Virtuality codes

DW_VIRTUALITY_none
DW_VIRTUALITY_virtual
DW_VIRTUALITY_pure_virtual

2.11 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, which is a flag.

2.12 Segmented Addresses

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 selector 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:

Table 2.7: Example address class codes

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*

2.13 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);
  }
}
```

C scoping rules require that the value of the variable `x` passed to the function `g` is the value of the global `float` variable `x` rather than of the local `int` variable `x`.

2.13.1 Non-Defining Declarations

A debugging information entry that represents a non-defining or otherwise incomplete declaration of a program entity has a DW_AT_declaration attribute, which is a flag.

A non-defining type declaration may nonetheless have children as illustrated in Section E.2.3 on page 387.

2.13.2 Declarations Completing Non-Defining Declarations

A debugging information entry that represents 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. A debugging information entry with a DW_AT_specification attribute does not need to duplicate information provided by the debugging information entry referenced by that specification attribute.

When the non-defining declaration is contained within a type that has been placed in a separate type unit (see Section 3.1.4 on page 68), the DW_AT_specification attribute cannot refer directly to the entry in the type unit. Instead, the current compilation unit may contain a “skeleton” declaration of the type, which contains only the relevant declaration and its ancestors as necessary to provide the context (including containing types and namespaces). The DW_AT_specification attribute would then be a reference to the declaration entry within the skeleton declaration tree. The debugging information entry for the top-level type in the skeleton tree may contain a DW_AT_signature attribute whose value is the type signature (see Section 7.32 on page 245).

Not all attributes of the debugging information entry referenced by a DW_AT_specification attribute apply to the referring debugging information entry. For example, DW_AT_sibling and DW_AT_declaration cannot apply to a referring entry.

2.14 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 unsigned 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 on page 148). 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.

2.15 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 of class string represents the name. 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 DW_AT_name attribute, including the keyword operator (in names such as "operator +"), if present. See also Section 2.22 following regarding the use of DW_AT_linkage_name for mangled names. Sequences of multiple whitespace characters may be compressed.

For additional discussion, see the Best Practices section of the DWARF Wiki (`http://wiki.dwarfstd.org/index.php?title=Best_Practices`.)

2.16 Data Locations and DWARF Procedures

Any debugging information entry describing a data object (which includes variables and parameters) or common blocks may have a DW_AT_location attribute, whose value is a location description (see Section 2.6 on page 38).

A DWARF procedure is represented by any 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 together with a DW_AT_location attribute.

A DWARF procedure is called by a DW_OP_call2, DW_OP_call4 or DW_OP_call_ref DWARF expression operator (see Section 2.5.1.5 on page 35).

2.17 Code Addresses, Ranges and Base Addresses

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, lexical blocks, try/catch blocks (see Section 3.8 on page 93), 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.

The base address of the scope for any of the debugging information entries listed above is given by either the DW_AT_low_pc attribute or the first address in the first range entry in the list of ranges given by the DW_AT_ranges attribute. If there is no such attribute, the base address is undefined.

2.17.1 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 address for the entity.

2.17.2 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 address of the first instruction associated with the entity. If the value of the DW_AT_high_pc is of class address, it is the address of the first location past the last instruction associated with the entity; if it is of class constant, the value is an unsigned integer offset which when added to the low PC gives the 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.

2.17.3 Non-Contiguous Address Ranges

Range lists are used when the set of addresses for a debugging information entry cannot be described as a single contiguous range. Range lists are contained in a separate object file section called `.debug_rnglists` or `.debug_rnglists.dwo` (in split units).

A range list is identified by a DW_AT_ranges or other attribute whose value is of class rnglist (see Section 7.5.5 on page 212).

This range list representation, the rnglist class, and the related DW_AT_rnglists_base attribute are new in DWARF Version 5. Together they eliminate most or all of the object language relocations previously needed for range lists.

Each range list entry is one of the following kinds:

- Bounded range. This kind of entry defines an address range that is included in the range list. The starting address is the lowest address of the address range. The ending address is the address of the first location past the highest address of the address range.

  There are several kinds of bounded range entries which specify the starting and ending addresses in different ways.

- Base address. This kind of entry provides an address to be used as the base address for the beginning and ending address offsets given in certain bounded range entries. The applicable base address of a range list entry is determined by the closest preceding base address entry in the same range list. If there is no preceding base address entry, then the applicable base address defaults to the base address of the compilation unit (see Section 3.1.1 on page 60).

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

- End-of-list. This kind of entry marks the end of the range list.

Each range list consists of a sequence of zero or more bounded range or base address entries, terminated by an end-of-list entry.

A range list containing only an end-of-list entry describes an empty scope (which contains no instructions).

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

A bounded range entry whose beginning and ending address offsets are equal (including zero) indicates an empty range and may be ignored.

Each range list entry begins with a single byte identifying the kind of that entry, followed by zero or more operands depending on the kind.

In the descriptions that follow, the term address index means the index of an address in the `.debug_addr` section. This index is relative to the value of the DW_AT_addr_base attribute of the associated compilation unit. The address given by this kind of operand is *not* relative to the compilation unit base address.

The following entry kinds are defined for use in both split or non-split units:

1. **DW_RLE_end_of_list**
   An end-of-list entry contains no further data.

   A series of this kind of entry may be used for padding or alignment purposes.

2. **DW_RLE_base_addressx**
   A base address entry has one unsigned LEB128 operand. The operand value is an address index (into the `.debug_addr` section) that indicates the applicable base address used by following DW_RLE_offset_pair entries.

3. **DW_RLE_startx_endx**
   This is a form of bounded range entry that has two unsigned LEB128 operands. The operand values are address indices (into the `.debug_addr` section) that indicate the starting and ending addresses, respectively, that define the address range.

4. **DW_RLE_startx_length**
   This is a form of bounded location description that has two unsigned ULEB operands. The first value is an address index (into the `.debug_addr` section) that indicates the beginning of the address range. The second value is the length of the range.

5. **DW_RLE_offset_pair**
   This is a form of bounded range entry that has two unsigned LEB128 operands. The values of these operands are the starting and ending offsets, respectively, relative to the applicable base address, that define the address range.

The following kinds of range entry may be used only in non-split units:

6. **DW_RLE_base_address**
   A base address entry has one target address operand. This operand is the same size as used in DW_FORM_addr. This address is used as the base address when interpreting offsets in subsequent location list entries of kind DW_RLE_offset_pair.

7. **DW_RLE_start_end**
   This is a form of bounded range entry that has two target address operands. Each operand is the same size as used in DW_FORM_addr. These indicate the starting and ending addresses, respectively, that define the address range for which the following location is valid.

8. **DW_RLE_start_length**
   This is a form of bounded range entry that has one target address operand value and an unsigned LEB128 integer length operand value. The address is the beginning address of the range over which the location description is valid, and the length is the number of bytes in that range.

2.18 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, lexical blocks, try/catch blocks, and the like, may have a DW_AT_entry_pc attribute to indicate the entry address which is the address of the instruction where execution begins within that range of addresses. If the value of the DW_AT_entry_pc attribute is of class address that address is the entry address; or, if it is of class constant, the value is an unsigned integer offset which, when added to the base address of the function, gives the entry address.

If no DW_AT_entry_pc attribute is present, then the entry address is assumed to be the same as the base address of the containing scope.

2.19 Static and Dynamic Values of Attributes

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 debugging information entry. This entry may:
  - describe a constant which is the attribute value,
  - describe a variable which contains the attribute value, or
  - contain a DW_AT_location attribute whose value is a DWARF expression which computes the attribute value (for example, a DW_TAG_dwarf_procedure entry).
- For an exprloc, the value is interpreted as a DWARF expression; evaluation of the expression yields the value of the attribute.

2.20 Entity Descriptions

Some debugging information entries may describe entities in the program that are artificial, or which otherwise have a "name" that is not a valid identifier in the programming language. This attribute provides a means for the producer to indicate the purpose or usage of the containing debugging infor

Generally, any debugging 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 providing a description of the entity.

It is expected that a debugger will display these descriptions as part of displaying other properties of an entity.

2.21 Byte and Bit Sizes

Many debugging information entries allow either a DW_AT_byte_size attribute or a DW_AT_bit_size attribute, whose integer constant value (see Section 2.19) specifies an amount of storage. The value of the DW_AT_byte_size attribute is interpreted in bytes and the value of the DW_AT_bit_size attribute is interpreted in bits. The DW_AT_string_length_byte_size and DW_AT_string_length_bit_size attributes are similar.

In addition, the integer constant value of a DW_AT_byte_stride attribute is interpreted in bytes and the integer constant value of a DW_AT_bit_stride attribute is interpreted in bits.

2.22 Linkage Names

Some language implementations, notably C++ and similar languages, make use of implementation-defined names within object files that are different from the identifier names (see Section 2.15 on page 50) of entities as they appear in the source. Such names, sometimes known as mangled names, are used in various ways, such as: to encode additional information about an entity, to distinguish multiple entities that have the same name, and so on. When an entity has an associated distinct linkage name it may sometimes be useful for a producer to include this name in the DWARF description of the program to facilitate consumer access to and use of object file information about an entity and/or information that is encoded in the linkage name itself.

A debugging information entry may have a DW_AT_linkage_name attribute whose value is a null-terminated string containing the object file linkage name associated with the corresponding entity.

2.23 Template Parameters

In C++, a template is a generic definition of a class, function, member function, or typedef (alias). A template has formal parameters that can be types or constant values; the class, function, member function, or typedef is instantiated differently for each distinct combination of type or value actual parameters. DWARF does not represent the generic template definition, but does represent each instantiation.

A debugging information entry that represents a template instantiation will contain child entries describing the actual template parameters. The containing entry and each of its child entries reference a template parameter entry in any circumstance where the template definition referenced a formal template parameter.

A template type parameter is represented by a debugging information entry with the tag DW_TAG_template_type_parameter. A template value parameter is represented by a debugging information entry with the tag DW_TAG_template_value_parameter. The actual template parameter entries appear in the same order as the corresponding template formal parameter declarations in the source program.

A type or value parameter entry may have a DW_AT_name attribute, whose value is a null-terminated string containing the name of the corresponding formal parameter. The entry may also have a DW_AT_default_value attribute, which is a flag indicating that the value corresponds to the default argument for the template parameter.

A template type parameter entry has a DW_AT_type attribute describing the actual type by which the formal is replaced.

A template value parameter entry has a DW_AT_type attribute describing the type of the parameterized value. The entry also has an attribute giving the actual compile-time or run-time constant value of the value parameter for this instantiation. This can be a DW_AT_const_value attribute, whose value is the compile-time constant value as represented on the target architecture, or a DW_AT_location attribute, whose value is a single location description for the run-time constant address.

2.24 Alignment

A debugging information entry may have a DW_AT_alignment attribute whose value of class constant is a positive, non-zero, integer describing the alignment of the entity.

For example, an alignment attribute whose value is 8 indicates that the entity to which it applies occurs at an address that is a multiple of eight (not a multiple of 2^8 or 256).

Chapter 3

Program Scope Entries

This section describes debugging information entries that relate to different levels of program scope: compilation, module, subprogram, and so on. Except for separate type entries (see Section 3.1.4 on page 68), these entries may be thought of as ranges of text addresses within the program.

3.1 Unit Entries

A DWARF object file is an object file that contains one or more DWARF compilation units, of which there are these kinds:

- A full compilation unit describes a complete compilation, possibly in combination with related partial compilation units and/or type units.
- A partial compilation unit describes a part of a compilation (generally corresponding to an imported module) which is imported into one or more related full compilation units.
- A type unit is a specialized unit (similar to a compilation unit) that represents a type whose description may be usefully shared by multiple other units.

These first three kinds of compilation unit are sometimes called "conventional" compilation units–they are kinds of compilation units that were defined prior to DWARF Version 5. Conventional compilation units are part of the same object file as the compiled code and data (whether relocatable, executable, shared and so on). The word "conventional" is usually omitted in these names, unless needed to distinguish them from the similar split compilation units below.

- A skeleton compilation unit represents the DWARF debugging information for a compilation using a minimal description that identifies a separate split compilation unit that provides the remainder (and most) of the description.

A skeleton compilation acts as a minimal conventional full compilation (see above) that identifies and is paired with a corresponding split full compilation (as described below). Like the conventional compilation units, a skeleton compilation unit is part of the same object file as the compiled code and data.

- A split compilation unit describes a complete compilation, possibly in combination with related type compilation units. It corresponds to a specific skeleton compilation unit.

- A split type unit is a specialized compilation unit that represents a type whose description may be usefully shared by multiple other units.

Split compilation units and split type units may be contained in object files separate from those containing the program code and data. These object files are not processed by a linker; thus, split units do not depend on underlying object file relocations.

Either a full compilation unit or a partial compilation unit may be logically incorporated into another compilation unit using an imported unit entry (see Section 3.2.5 on page 74).

A partial compilation unit is not defined for use within a split object file.

In the remainder of this document, the word "compilation" in the phrase "compilation unit" is generally omitted, unless it is deemed needed for clarity or emphasis.

3.1.1 Full and Partial Compilation Unit Entries

A full 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 compilation, a single compilation unit with the tag DW_TAG_compile_unit represents a complete object file and the tag DW_TAG_partial_unit (as well as tag DW_TAG_type_unit) is not used. In a compilation employing the DWARF space compression and duplicate elimination techniques from Appendix E.1 on page 365, multiple compilation units using the tags DW_TAG_compile_unit, DW_TAG_partial_unit and/or DW_TAG_type_unit are used to represent portions of an object file.

A full 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 header 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 full compilation unit can also function in a manner similar to a partial compilation unit in some cases. See Appendix E on page 365 for discussion of related compression techniques.

A full or partial 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 3.2.5 on page 74).

A full or partial compilation unit entry 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 2.17 on page 51).

   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 2.6.2 on page 43) and range lists (see Section 2.17.3 on page 52).

2. A DW_AT_name attribute whose value is a null-terminated string containing the full or relative path name (relative to the value of the DW_AT_comp_dir attribute, see below) 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 Table 3.1 on the following page.

Table 3.1: Language names

Language name	**Meaning**
DW_LANG_Ada83 †	ISO Ada:1983
DW_LANG_Ada95 †	ISO Ada:1995
DW_LANG_BLISS	BLISS
DW_LANG_C	Non-standardized C, such as K&R
DW_LANG_C89	ISO C:1989
DW_LANG_C99	ISO C:1999
DW_LANG_C11	ISO C:2011
DW_LANG_C_plus_plus	ISO C++98
DW_LANG_C_plus_plus_03	ISO C++03
DW_LANG_C_plus_plus_11	ISO C++11
DW_LANG_C_plus_plus_14	ISO C++14
DW_LANG_Cobol74	ISO COBOL:1974
DW_LANG_Cobol85	ISO COBOL:1985
DW_LANG_D †	D
DW_LANG_Dylan †	Dylan
DW_LANG_Fortran77	ISO FORTRAN:1977
DW_LANG_Fortran90	ISO Fortran:1990
DW_LANG_Fortran95	ISO Fortran:1995
DW_LANG_Fortran03	ISO Fortran:2004
DW_LANG_Fortran08	ISO Fortran:2010
DW_LANG_Go †	Go
DW_LANG_Haskell †	Haskell
DW_LANG_Java	Java
DW_LANG_Julia †	Julia
DW_LANG_Modula2	ISO Modula-2:1996
DW_LANG_Modula3	Modula-3
DW_LANG_ObjC	Objective C
DW_LANG_ObjC_plus_plus	Objective C++
DW_LANG_OCaml †	OCaml
DW_LANG_OpenCL †	OpenCL

Continued on next page

Language name	**Meaning**
DW_LANG_Pascal83	ISO Pascal:1983
DW_LANG_PLI †	ANSI PL/I:1976
DW_LANG_Python †	Python
DW_LANG_RenderScript †	RenderScript Kernel Language
DW_LANG_Rust †	Rust
DW_LANG_Swift	Swift
DW_LANG_UPC	UPC (Unified Parallel C)
† *Support for these languages is limited*	

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 on page 148).

5. A DW_AT_macros 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_macro` section of the first byte of the macro information for this compilation unit (see Section 6.3 on page 165).

   The DW_AT_macros attribute is new in DWARF Version 5, and supersedes the DW_AT_macro_info attribute of earlier DWARF versions. While DW_AT_macros and DW_AT_macro_info attributes cannot both occur in the same compilation unit, both may be found in the set of units that make up an executable or shared object file. The two attributes have distinct encodings to facilitate such coexistence.

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.

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 will 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 Table 3.2.

Table 3.2: Identifier case codes

DW_ID_case_sensitive
DW_ID_up_case
DW_ID_down_case
DW_ID_case_insensitive

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.

A 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.

A 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.

A 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 case is not significant.

A debugger should ignore case when doing lookups.

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 on page 103).

   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, or filenames in the line number table) are represented using the UTF-8 representation.

11. A DW_AT_main_subprogram attribute, which is a flag, whose presence indicates that the compilation unit contains a subprogram that has been identified as the starting subprogram of the program. If more than one compilation unit contains this flag, any one of them may contain the starting function.

    Fortran has a PROGRAM statement which is used to specify and provide a user-specified name for the main subroutine of a program. C uses the name "main" to identify the main subprogram of a program. Some other languages provide similar or other means to identify the main subprogram of a program. The DW_AT_main_subprogram attribute may also be used to identify such subprograms (see Section 3.3.1 on page 75).

12. A DW_AT_entry_pc attribute whose value is the address of the first executable instruction of the unit (see Section 2.18 on page 55).

13. A DW_AT_str_offsets_base attribute, whose value is of class stroffsetsptr. This attribute points to the first string offset of the compilation unit's contribution to the `.debug_str_offsets` (or `.debug_str_offsets.dwo`) section. Indirect string references (using DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 or DW_FORM_strx4) within the compilation unit are interpreted as indices relative to this base.

14. A DW_AT_addr_base attribute, whose value is of class addrptr. This attribute points to the beginning of the compilation unit's contribution to the `.debug_addr` section. Indirect references (using DW_FORM_addrx, DW_FORM_addrx1, DW_FORM_addrx2, DW_FORM_addrx3, DW_FORM_addrx4, DW_OP_addrx, DW_OP_constx, DW_LLE_base_addressx, DW_LLE_startx_endx, DW_LLE_startx_length, DW_RLE_base_addressx, DW_RLE_startx_endx or DW_RLE_startx_length) within the compilation unit are interpreted as indices relative to this base.

15. A DW_AT_rnglists_base attribute, whose value is of class rnglistsptr. This attribute points to the beginning of the offsets table (immediately following the header) of the compilation unit's contribution to the `.debug_rnglists` section. References to range lists (using DW_FORM_rnglistx) within the compilation unit are interpreted relative to this base.

16. A DW_AT_loclists_base attribute, whose value is of class loclistsptr. This attribute points to the beginning of the offsets table (immediately following the header) of the compilation unit's contribution to the `.debug_loclists` section. References to location lists (using DW_FORM_loclistx) within the compilation unit are interpreted relative to this base.

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 Skeleton Compilation Unit Entries

When generating a split DWARF object file (see Section 7.3.2 on page 187), the compilation unit in the `.debug_info` section is a "skeleton" compilation unit with the tag DW_TAG_skeleton_unit, which contains a DW_AT_dwo_name attribute as well as a subset of the attributes of a full or partial compilation unit. In general, it contains those attributes that are necessary for the consumer to locate the object file where the split full compilation unit can be found, and for the consumer to interpret references to addresses in the program.

A skeleton compilation unit has no children.

A skeleton compilation unit has a DW_AT_dwo_name attribute:

1. A DW_AT_dwo_name attribute whose value is a null-terminated string containing the full or relative path name (relative to the value of the DW_AT_comp_dir attribute, see below) of the object file that contains the full compilation unit.

   The value in the `dwo_id` field of the unit header for this unit is the same as the value in the `dwo_id` field of the unit header of the corresponding full compilation unit (see Section 7.5.1 on page 199).

   The means of determining a compilation unit ID does not need to be similar or related to the means of determining a type unit signature. However, it should be suitable for detecting file version skew or other kinds of mismatched files and for looking up a full split unit in a DWARF package file (see Section 7.3.5 on page 190).

A skeleton compilation unit may have additional attributes, which are the same as for conventional compilation unit entries except as noted, from among the following:

2. Either a DW_AT_low_pc and DW_AT_high_pc pair of attributes or a DW_AT_ranges attribute.
3. A DW_AT_stmt_list attribute.
4. A DW_AT_comp_dir attribute.
5. A DW_AT_use_UTF8 attribute.

   This attribute applies to strings referred to by the skeleton compilation unit entry itself, and strings in the associated line number information. The representation for strings in the object file referenced by the DW_AT_dwo_name attribute is determined by the presence of a DW_AT_use_UTF8 attribute in the full compilation unit (see Section 3.1.3 on the following page).
6. A DW_AT_str_offsets_base attribute, for indirect strings references from the skeleton compilation unit.
7. A DW_AT_addr_base attribute.

All other attributes of a compilation unit entry (described in Section 3.1.1 on page 60) are placed in the split full compilation unit (see 3.1.3 on the following page). The attributes provided by the skeleton compilation unit entry do not need to be repeated in the full compilation unit entry.

The DW_AT_addr_base and DW_AT_str_offsets_base attributes provide context that may be necessary to interpret the contents of the corresponding split DWARF object file.

The DW_AT_base_types attribute is not defined for a skeleton compilation unit.

3.1.3 Split Full Compilation Unit Entries

A split full compilation unit is represented by a debugging information entry with tag DW_TAG_compile_unit. It is very similar to a conventional full compilation unit but is logically paired with a specific skeleton compilation unit while being physically separate.

A split full compilation unit may have the following attributes, which are the same as for conventional compilation unit entries except as noted:

1. A DW_AT_name attribute.
2. A DW_AT_language attribute.
3. A DW_AT_macros attribute. The value of this attribute is of class macptr, which is an offset relative to the `.debug_macro.dwo` section.
4. A DW_AT_producer attribute.
5. A DW_AT_identifier_case attribute.
6. A DW_AT_main_subprogram attribute.
7. A DW_AT_entry_pc attribute.
8. A DW_AT_use_UTF8 attribute.

The following attributes are not part of a split full compilation unit entry but instead are inherited (if present) from the corresponding skeleton compilation unit: DW_AT_low_pc, DW_AT_high_pc, DW_AT_ranges, DW_AT_stmt_list, DW_AT_comp_dir, DW_AT_str_offsets_base, DW_AT_addr_base and DW_AT_rnglists_base.

The DW_AT_base_types attribute is not defined for a split full compilation unit.

3.1.4 Type Unit Entries

An object file may contain any number of separate type unit entries, each representing a single complete type definition. Each type unit must be uniquely identified by an 8-byte signature, stored as part of the type unit, which can be used to reference the type definition from debugging information entries in other compilation units and type units.

Conventional and split type units are identical except for the sections in which they are represented (see 7.3.2 on page 187 for details). Moreover, the DW_AT_str_offsets_base attribute (see below) is not used in a split type unit.

A type unit is represented by a debugging information entry with the tag DW_TAG_type_unit. A type unit entry owns debugging information entries that represent the definition of a single type, plus additional debugging information entries that may be necessary to include as part of the definition of the type.

A type unit entry may have the following attributes:

1. A DW_AT_language attribute, whose constant value is an integer code indicating the source language used to define the type. The set of language names and their meanings are given in Table 3.1 on page 62.

2. A DW_AT_stmt_list attribute whose value of class lineptr points to the line number information for this type unit.

   Because type units do not describe any code, they do not actually need a line number table, but the line number headers contain a list of directories and file names that may be referenced by the DW_AT_decl_file attribute of the type or part of its description.

   In an object file with a conventional compilation unit entry, the type unit entries may refer to (share) the line number table used by the compilation unit. In a type unit located in a split compilation unit, the DW_AT_stmt_list attribute refers to a "specialized" line number table in the `.debug_line.dwo` section, which contains only the list of directories and file names.

   All type unit entries in a split DWARF object file may (but are not required to) refer to the same specialized line number table.

3. A DW_AT_use_UTF8 attribute, which is a flag whose presence indicates that all strings referred to by this type unit entry, its children, and its associated specialized line number table, are represented using the UTF-8 representation.

4. A DW_AT_str_offsets_base attribute, whose value is of class stroffsetsptr. This attribute points to the first string offset of the type unit's contribution to the `.debug_str_offsets` section. Indirect string references (using DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 or DW_FORM_strx4) within the type unit are interpreted as indices relative to this base.

A type unit entry for a given type T owns a debugging information entry that represents a defining declaration of type T. If the type is nested within enclosing types or namespaces, the debugging information entry for T is nested within debugging information entries describing its containers; otherwise, T is a direct child of the type unit entry.

A type unit entry may also own additional debugging information entries that represent declarations of additional types that are referenced by type T and have not themselves been placed in separate type units. Like T, if an additional type U is nested within enclosing types or namespaces, the debugging information entry for U is nested within entries describing its containers; otherwise, U is a direct child of the type unit entry.

The containing entries for types T and U are declarations, and the outermost containing entry for any given type T or U is a direct child of the type unit entry. The containing entries may be shared among the additional types and between T and the additional types.

Examples of these kinds of relationships are found in Section E.2.1 on page 377 and Section E.2.3 on page 387.

Types are not required to be placed in type units. In general, only large types such as structure, class, enumeration, and union types included from header files should be considered for separate type units. Base types and other small types are not usually worth the overhead of placement in separate type units. Types that are unlikely to be replicated, such as those defined in the main source file, are also better left in the main 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 Modula-2 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).

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.

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 2.17 on page 51). 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 2.18 on page 55).

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.

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 declarations occur in the namespace.

A namespace may have a DW_AT_export_symbols attribute which is a flag which indicates that all member names defined within the namespace may be referenced as if they were defined within the containing namespace.

This may be used to describe an inline namespace in C++.

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). C++ states that declarations in the unnamed namespace are implicitly available in the containing scope; a producer should make this effect explicit with the DW_AT_export_symbols attribute, or by using a DW_TAG_imported_module that is a sibling of the namespace entry and references it.

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.

For C++ namespace examples, see Appendix D.3 on page 313.

3.2.3 Imported (or Renamed) Declaration Entries

Some languages support the concept of importing into or making accessible in a given unit certain declarations that occur in a different module or scope. 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 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 occurs.

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 and a DW_AT_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 a DW_AT_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 module 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 renamed 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 a DW_AT_import attribute referring to the namespace entry of the appropriate extension of the namespace (which might be the original namespace entry) 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 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 and no owned child entries.

A use statement with an "only list" is represented by a series of individual imported declaration entries as described in Section 3.2.3 on page 72.

A Fortran use statement 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 X 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 on page 313 for an example.

3.2.5 Imported Unit Entries

The place where a normal or partial compilation unit is imported 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 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.

3.3 Subroutine and Entry Point Entries

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

DW_TAG_subprogram	A subroutine or function
DW_TAG_inlined_subroutine	A particular inlined instance of a subroutine or function
DW_TAG_entry_point	An 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. It may also have a DW_AT_linkage_name attribute as described in Section 2.22 on page 56.

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, which is a flag.

Additional attributes for functions that are members of a class or structure are described in Section 5.7.8 on page 120.

A subroutine entry may contain a DW_AT_main_subprogram attribute which is a flag whose presence indicates that the subroutine has been identified as the starting function of the program. If more than one subprogram contains this flag, any one of them may be the starting subroutine of the program.

See also Section 3.1 on page 59) regarding the related use of this attribute to indicate that a compilation unit contains the main subroutine of a program.

3.3.1.1 Calling Convention Information

A subroutine entry may contain a DW_AT_calling_convention attribute, whose value is an integer constant. The set of calling convention codes for subroutines is given in Table 3.3 on the next page.

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.

Table 3.3: Calling convention codes for subroutines

DW_CC_normal
DW_CC_program
DW_CC_nocall

Note that DW_CC_normal is also used as a calling convention code for certain types (see Table 5.5 on page 115).

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.

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.

The DW_CC_program value is intended to support Fortran main programs which in some implementations may not be callable or which must be invoked in a special way. It is not intended as a way of finding the entry address for the program.

3.3.1.2 Miscellaneous Subprogram Properties

In 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, which is a flag. The attribute indicates whether a subroutine entry point corresponds to a function declaration that includes parameter prototype information.

A subprogram entry may have a DW_AT_elemental attribute, which is a flag. The attribute indicates whether the subroutine or entry point was declared with the “elemental” keyword or property.

A subprogram entry may have a DW_AT_pure attribute, which is a flag. The attribute indicates whether the subroutine was declared with the “pure” keyword or property.

A subprogram entry may have a DW_AT_recursive attribute, which is a flag. The attribute indicates whether the subroutine or entry point was declared with the "recursive" keyword or property.

A subprogram entry may have a DW_AT_noreturn attribute, which is a flag. The attribute indicates whether the subprogram was declared with the "noreturn" keyword or property indicating that the subprogram can be called, but will never return to its caller.

The Fortran language allows the keywords `elemental`, `pure` and `recursive` to be included as part of the declaration of a subroutine; these attributes reflect that usage. These attributes are not relevant for languages that do not support similar keywords or syntax. In particular, the DW_AT_recursive attribute is neither needed nor appropriate in languages such as C where functions support recursion by default.

3.3.1.3 Call Site-Related Attributes

While subprogram attributes in the previous section provide information about the subprogram and its entry point(s) as a whole, the following attributes provide summary information about the calls that occur within a subprogram.

A subroutine entry may have DW_AT_call_all_tail_calls, DW_AT_call_all_calls and/or DW_AT_call_all_source_calls attributes, each of which is a flag. These flags indicate the completeness of the call site information provided by call site entries (see Section 3.4.1 on page 89) within the subprogram.

The DW_AT_call_all_tail_calls attribute indicates that every tail call that occurs in the code for the subprogram is described by a DW_TAG_call_site entry. (There may or may not be other non-tail calls to some of the same target subprograms.)

The DW_AT_call_all_calls attribute indicates that every non-inlined call (either a tail call or a normal call) that occurs in the code for the subprogram is described by a DW_TAG_call_site entry.

The DW_AT_call_all_source_calls attribute indicates that every call that occurs in the code for the subprogram, including every call inlined into it, is described by either a DW_TAG_call_site entry or a DW_TAG_inlined_subroutine entry; further, any call that is optimized out is nonetheless also described using a DW_TAG_call_site entry that has neither a DW_AT_call_pc nor DW_AT_call_return_pc attribute.

The DW_AT_call_all_source_calls attribute is intended for debugging information format consumers that analyze call graphs.

If the the DW_AT_call_all_source_calls attribute is present then the DW_AT_call_all_calls and DW_AT_call_all_tail_calls attributes are also implicitly present. Similarly, if the DW_AT_call_all_calls attribute is present then the DW_AT_call_all_tail_calls attribute is implicitly present.

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.

Debugging information entries for declarations of C++ member functions with an `auto` *return type specifier should use an unspecified type entry (see Section 5.2 on page 108). The debugging information entry for the corresponding definition should provide the deduced return type. This practice causes the description of the containing class to be consistent across compilation units, allowing the class declaration to be placed into a separate type unit if desired.*

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 2.17 on page 51).

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 2.18 on page 55).

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

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

A subroutine entry representing a subroutine declaration that is not also a definition does not have 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 for entries for declarations other than formal parameters. The formal parameter entries may be interspersed with other entries used by formal parameter entries, such as type entries.

The unspecified (sometimes called "varying") parameters of a subroutine 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 information entry for the common block being included (see Section 4.2 on page 100).

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 specified 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 describes the "frame base" for the subroutine or entry point. If the location description is a simple register location description, the given register contains the frame base address. If the location description is a DWARF expression, the result of evaluating that expression is the frame base address. Finally, for a location list, this interpretation applies to each location description contained in the list of location list entries.

The use of one of the DW_OP_reg<n> operations in this context is equivalent to using DW_OP_breg<n>(0) but more compact. However, these are not equivalent in general.

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

1. *In subprograms 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 ones involving the frame base.*
2. *It can be used in resolving "up-level" addressing within nested routines. (See also 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 obeys the following constraints:

1. It computes a value that does not change during the life of the subprogram, and
2. The computed value is 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 which it may validly throw.

If a subroutine explicitly declares that it may throw an exception of 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. 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 for calls with values of different types. DWARF does not represent the generic template definition, but does represent each instantiation.

A function template instantiation is represented by a debugging information entry with the tag DW_TAG_subprogram. With the following 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 and values. The exceptions are:

1. Template parameters are described and referenced as specified in Section 2.23 on page 57.
2. If the compiler has generated a separate 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 information entry representing that compilation unit is empty or omitted.
3. If the subprogram entry representing the template instantiation or any of its child entries contain declaration coordinate attributes, those attributes refer to the source for the template definition, not to any source generated artificially by the compiler for this instantiation.

3.3.8 Inlinable and Inlined 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 Table 3.4 on the next page.

In C++, a function or a constructor declared with `constexpr` *is implicitly declared inline. The abstract instance (see Section 3.3.8.1 on the following page) is represented by a debugging information entry with the tag DW_TAG_subprogram. Such an entry has a DW_AT_inline attribute whose value is DW_INL_inlined.*

Table 3.4: Inline codes

Name	Meaning
DW_INL_not_inlined	Not declared inline nor inlined by the compiler (equivalent to the absence of the containing DW_AT_inline attribute)
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

3.3.8.1 Abstract Instances

Any subroutine entry that contains a DW_AT_inline attribute whose value is other than DW_INL_not_inlined is known as an abstract instance root. Any debugging information entry that is owned (either directly or indirectly) by an abstract instance root is known as an abstract instance entry. 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.

Each abstract instance root is either part of a larger tree (which gives a context for the root) or uses DW_AT_specification to refer to the declaration in context.

For example, in C++ the context might be a namespace declaration or a class declaration.

Abstract instance trees are defined so that no entry is part of more than one abstract instance tree.

Attributes and children in an abstract instance are shared by all concrete instances (see Section 3.3.8.2 on the next page).

A debugging information entry that is a member of an abstract instance tree may not contain any attributes which describe aspects of the subroutine which vary between distinct inlined 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.7.3 on page 333 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 Instances

Each inline expansion of a subroutine is represented by a debugging information entry with the tag DW_TAG_inlined_subroutine. Each such entry is 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 2.17 following). An inlined subroutine entry may also contain a DW_AT_entry_pc attribute, representing the first executable instruction of the inline expansion (see Section 2.18 on page 55).

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 2.14 on page 50).

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.

An inlined subroutine entry may have a DW_AT_const_expr attribute, which is a flag whose presence indicates that the subroutine has been evaluated as a compile-time constant. Such an entry may also have a DW_AT_const_value attribute, whose value may be of any form that is appropriate for the representation of the subroutine's return value. The value of this attribute is the actual return value of the subroutine, represented as it would be on the target architecture.

In C++, if a function or a constructor declared with `constexpr` is called with constant expressions, then the corresponding concrete inlined instance has a DW_AT_const_expr attribute, as well as a DW_AT_const_value attribute whose value represents the actual return value of the concrete inlined instance.

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 inlined instance tree is nested within another concrete instance tree, the entries in the nested concrete inline instance tree are not considered to be entries in the outer concrete instance tree.

Concrete inlined instance trees are defined so that no entry is part of more than one concrete inlined 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 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 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 inlined 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 inlined subroutine and for which no concrete instance entry exists, the reference refers 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 inlined 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 inlined expansion. In that case, they will not have associated entries in the abstract instance tree, do 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 inlined expansions. In any expansion which deviates from that expectation, the entries can be described in its concrete inlined instance tree.

3.3.8.3 Out-of-Line Instances of Inlined Subroutines

Under some conditions, compilers may need to generate concrete executable instances of inlined subroutines other than at points where those subroutines are actually called. Such concrete instances of inlined 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 inlined 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 inlined subroutine has the same tag as does its associated (abstract) inlined subroutine entry (that is, tag DW_TAG_subprogram rather than 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 be owned by the same parent entry.

3.3.8.4 Nested Inlined 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-inlined subroutine nested within an inlined 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 inlined 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 on page 82, 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 inlined 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 on page 83 or 3.3.8.3 following , respectively, and without regard to the fact that it is within an outer concrete instance tree.

See Appendix D.7 on page 329 for discussion and examples.

3.3.9 Trampolines

A trampoline is a compiler-generated subroutine that serves as an intermediary in making a call to another subroutine. 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:

- 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` 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 existence 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.

3.4 Call Site Entries and Parameters

A call site entry describes a call from one subprogram to another in the source program. It provides information about the actual parameters of the call so that they may be more easily accessed by a debugger. When used together with call frame information (see Section 6.4 on page 171), call site entries can be useful for computing the value of an actual parameter passed by a caller, even when the location description for the callee's corresponding formal parameter does not provide a current location for the formal parameter.

The DWARF expression for computing the value of an actual parameter at a call site may refer to registers or memory locations. The expression assumes these contain the values they would have at the point where the call is executed. After the called subprogram has been entered, these registers and memory locations might have been modified. In order to recover the values that existed at the point of the call (to allow evaluation of the DWARF expression for the actual parameter), a debugger may virtually unwind the subprogram activation (see Section 6.4 on page 171). Any register or memory location that cannot be recovered is referred to as "clobbered by the call."

A source call can be compiled into different types of machine code:

- A *normal call* uses a call-like instruction which transfers control to the start of some subprogram and preserves the call site location for use by the callee.
- A *tail call* uses a jump-like instruction which transfers control to the start of some subprogram, but there is no call site location address to preserve (and thus none is available using the virtual unwind information).
- A *tail recursion call* is a call to the current subroutine which is compiled as a jump to the current subroutine.
- An *inline (or inlined) call* is a call to an inlined subprogram, where at least one instruction has the location of the inlined subprogram or any of its blocks or inlined subprograms.

There are also different types of "optimized out" calls:

- An *optimized out (normal) call* is a call that is in unreachable code that has not been emitted (such as, for example, the call to `foo` in `if (0) foo();`).
- An *optimized out inline call* is a call to an inlined subprogram which either did not expand to any instructions or only parts of instructions belong to it and for debug information purposes those instructions are given a location in the caller.

DW_TAG_call_site entries describe normal and tail calls but not tail recursion calls, while DW_TAG_inlined_subroutine entries describe inlined calls (see Section 3.3.8 on page 81). Call site entries cannot describe tail recursion or optimized out calls.

3.4.1 Call Site Entries

A call site is represented by a debugging information entry with the tag DW_TAG_call_site. The entry for a call site is owned by the innermost debugging information entry representing the scope within which the call is present in the source program.

A scope entry (for example, a lexical block) that would not otherwise be present in the debugging information of a subroutine need not be introduced solely to represent the immediately containing scope of a call.

The call site entry may have a DW_AT_call_return_pc attribute which is the return address after the call. The value of this attribute corresponds to the return address computed by call frame information in the called subprogram (see Section 7.24 on page 238).

On many architectures the return address is the address immediately following the call instruction, but on architectures with delay slots it might be an address after the delay slot of the call.

The call site entry may have a DW_AT_call_pc attribute which is the address of the call-like instruction for a normal call or the jump-like instruction for a tail call.

If the call site entry corresponds to a tail call, it has the DW_AT_call_tail_call attribute, which is a flag.

The call site entry may have a DW_AT_call_origin attribute which is a reference. For direct calls or jumps where the called subprogram is known it is a reference to the called subprogram's debugging information entry. For indirect calls it may be a reference to a DW_TAG_variable, DW_TAG_formal_parameter or DW_TAG_member entry representing the subroutine pointer that is called.

The call site may have a DW_AT_call_target attribute which is a DWARF expression. For indirect calls or jumps where it is unknown at compile time which subprogram will be called the expression computes the address of the subprogram that will be called.

The DWARF expression should not use register or memory locations that might be clobbered by the call.

The call site entry may have a DW_AT_call_target_clobbered attribute which is a DWARF expression. For indirect calls or jumps where the address is not computable without use of registers or memory locations that might be clobbered by the call the DW_AT_call_target_clobbered attribute is used instead of the DW_AT_call_target attribute.

The expression of a call target clobbered attribute may only be valid at the time the call or call-like transfer of control is executed.

The call site entry may have a DW_AT_type attribute referencing a debugging information entry for the type of the called function.

When DW_AT_call_origin is present, DW_AT_type is usually omitted.

The call site entry may 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 call statement or expression. 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 2.14 on page 50).

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

3.4.2 Call Site Parameters

The call site entry may own DW_TAG_call_site_parameter debugging information entries representing the parameters passed to the call. Call site parameter entries occur in the same order as the corresponding parameters in the source. Each such entry has a DW_AT_location attribute which is a location description. This location description describes where the parameter is passed (usually either some register, or a memory location expressible as the contents of the stack register plus some offset).

Each DW_TAG_call_site_parameter entry may have a DW_AT_call_value attribute which is a DWARF expression which when evaluated yields the value of the parameter at the time of the call.

If it is not possible to avoid registers or memory locations that might be clobbered by the call in the expression, then the DW_AT_call_value attribute should not be provided. The reason for the restriction is that the value of the parameter may be needed in the midst of the callee, where the call clobbered registers or memory might be already clobbered, and if the consumer is not assured by the producer it can safely use those values, the consumer can not safely use the values at all.

For parameters passed by reference, where the code passes a pointer to a location which contains the parameter, or for reference type parameters, the DW_TAG_call_site_parameter entry may also have a DW_AT_call_data_location attribute whose value is a location description and a DW_AT_call_data_value attribute whose value is a DWARF expression. The DW_AT_call_data_location attribute describes where the referenced value lives during the call. If it is just DW_OP_push_object_address, it may be left out. The DW_AT_call_data_value attribute describes the value in that location. The expression should not use registers or memory locations that might be clobbered by the call, as it might be evaluated after virtually unwinding from the called function back to the caller.

Each call site parameter entry may also have a DW_AT_call_parameter attribute which contains a reference to a DW_TAG_formal_parameter entry, DW_AT_type attribute referencing the type of the parameter or DW_AT_name attribute describing the parameter's name.

Examples using call site entries and related attributes are found in Appendix D.15 on page 353.

3.5 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 2.17 on page 51).

A lexical block entry may also have a DW_AT_entry_pc attribute whose value is the address of the first executable instruction of the lexical block (see Section 2.18 on page 55).

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.

This is not the same as a C or C++ label (see Section 3.6).

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.

3.6 Label Entries

A label is a way of identifying a source location. 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 is 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 address of the first executable instruction for the location 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.

3.7 With Statement Entries

Both Pascal and Modula-2 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 2.17 on page 51).

A with statement entry may also have a DW_AT_entry_pc attribute whose value is the address of the first executable instruction of the with statement (see Section 2.18 on page 55).

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.

3.8 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 2.17 on page 51).

A try or catch block entry may also have a DW_AT_entry_pc attribute whose value is the address of the first executable instruction of the try or catch block (see Section 2.18 on page 55).

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

The siblings immediately following a try block entry are its corresponding catch block entries.

3.9 Declarations with Reduced Scope

Any debugging information entry for a declaration (including objects, subprograms, types and modules) whose scope has an address range that is a subset of the address range for the lexical scope most closely enclosing the declared entity may have a DW_AT_start_scope attribute to specify that reduced range of addresses.

There are two cases:

1. If the address range for the scope of the entry includes all of addresses for the containing scope except for a contiguous sequence of bytes at the beginning of the address range for the containing scope, then the address is specified using a value of class constant.

    a) If the address range of the containing scope is contiguous, the value of this attribute is the offset in bytes of the beginning of the address range for the scope of the object from the low PC value of the debugging information entry that defines that containing scope.

    b) If the address range of the containing scope is non-contiguous (see 2.17.3 on page 52) the value of this attribute is the offset in bytes of the beginning of the address range for the scope of the entity from the beginning of the first range list entry for the containing scope that is not a base address entry, a default location entry or an end-of-list entry.

2. Otherwise, the set of addresses for the scope of the entity is specified using a value of class rnglistsptr. This value indicates the beginning of a range list (see Section 2.17.3 on page 52).

For example, the scope of a variable may begin somewhere in the midst 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.

Consider the following example C code:

```
float x = 99.99;
int myfunc()
{
    float f = x;
    float x = 88.99;
    return 0;
}
```

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`*.*

Due to optimization, the scope of an object may be non-contiguous and require use of a range list even when the containing scope is contiguous. Conversely, the scope of an object may not require its own range list even when the containing scope is non-contiguous.

(empty page)

Chapter 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 have 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.

   If a variable entry describes an anonymous object (for example an anonymous union), the name attribute is omitted or its value consists of a single zero byte.

2. A DW_AT_external attribute, which is a flag, if the name of a variable is visible outside of its enclosing compilation unit.

   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, which is a flag that indicates whether 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.

   If no location attribute is present in a variable entry representing the definition of a variable (that is, with no DW_AT_declaration attribute), or if the location attribute is present but has an empty location description (as described in Section 2.6 on page 38), 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 supersedes the location specified by the defining declaration but only within the scope of the variable entry; if no location is specified, then the location specified in the defining declaration applies.

   This can occur, for example, for a C or C++ external variable (one that is defined and allocated in another compilation unit) and whose location varies in the current unit due to optimization.

   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 also has the tag DW_TAG_variable 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. A DW_AT_variable_parameter attribute, which is a flag, if a formal parameter entry represents a parameter whose value in the calling function may be modified by the callee. The absence of this attribute implies that the parameter's value in the calling function cannot be modified by the callee.

8. A DW_AT_is_optional attribute, which is a flag, if a parameter entry represents an optional parameter.

9. A DW_AT_default_value attribute for a formal parameter entry. The value of this attribute may be a constant, or a reference to the debugging information entry for a variable, or a reference to a debugging information entry containing a DWARF procedure. If the attribute form is of class constant, that constant is interpreted as a value whose type is the same as the type of the formal parameter. If the attribute form is of class reference, and the referenced entry is for a variable, the default value of the parameter is the value of the referenced variable. If the reference value is 0, no default value has been specified. Otherwise, the attribute represents an implicit DW_OP_call_ref to the referenced debugging information entry, and the default value of the parameter is the value returned by that DWARF procedure, interpreted as a value of the type of the formal parameter.

   For a constant form there is no way to express the absence of a default value.

10. A DW_AT_const_value attribute for 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. (Note that such an entry does not have a location attribute.) The value of this attribute may be a string or any of the constant data or data block forms, as appropriate for the representation of the variable's value. The value 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. A DW_AT_endianity attribute, whose value is a constant that specifies the endianity of the object. The value of this attribute specifies an ABI-defined byte ordering for the value of the object. If omitted, the default endianity of data for the given type is assumed.

    The set of values and their meaning for this attribute is given in Table 4.1. These represent the default encoding formats as defined by the target architecture's ABI or processor definition. The exact definition of these formats may differ in subtle ways for different architectures.

Table 4.1: Endianity attribute values

Name	Meaning
DW_END_default	Default endian encoding (equivalent to the absence of a DW_AT_endianity attribute)
DW_END_big	Big-endian encoding
DW_END_little	Little-endian encoding

12. A DW_AT_const_expr attribute, constant expression attribute which is a flag, if a variable entry represents a C++ object declared with the `constexpr` specifier. This attribute indicates that the variable can be evaluated as a compile-time constant.

    In C++, a variable declared with `constexpr` is implicitly `const`. Such a variable has a DW_AT_type attribute whose value is a reference to a debugging information entry describing a `const` qualified type.

13. A DW_AT_linkage_name attribute for a variable or constant entry as described in Section 2.22 on page 56.

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. It may also have a DW_AT_linkage_name attribute as described in Section 2.22 on page 56.

A common block entry 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.

Fortran allows each declarer of a common block to independently define its contents; thus, common blocks are not types.

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.

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.

(empty page)

Chapter 5

Type Entries

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

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 may have a DW_AT_name attribute whose value is a null-terminated string containing 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 has a DW_AT_encoding attribute describing how the base type is encoded and is to be interpreted. The DW_AT_encoding attribute is described in Section 5.1.1 following.

A base type entry may have a DW_AT_endianity attribute as described in Section 4.1 on page 97. If omitted, the encoding assumes the representation that is the default for the target architecture.

A base type entry has a DW_AT_byte_size attribute or a DW_AT_bit_size attribute whose integer constant value (see Section 2.21 on page 56) is the amount of storage needed to hold a value of the type.

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

If the value of an object of the given type does not fully occupy the storage described by a byte size attribute, the base type entry may also have a DW_AT_bit_size and a DW_AT_data_bit_offset attribute, both of whose values are integer constant values (see Section 2.19 on page 55). The bit size attribute describes the actual size in bits used to represent values of the given type. The data bit offset attribute is the offset in bits from the beginning of the containing storage to the beginning of the value. Bits that are part of the offset are padding. If this attribute is omitted a default data bit offset of zero is assumed.

A DW_TAG_base_type entry may have additional attributes that augment certain of the base type encodings; these are described in the following section.

5.1.1 Base Type Encodings

A base type entry 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 of class constant. The set of values and their meanings for the DW_AT_encoding attribute is given in Table 5.1 on the next page.

In Table 5.1, encodings are shown in groups that have similar characteristics purely for presentation purposes. These groups are not part of this DWARF specification.

5.1.1.1 Simple Encodings

Types with simple encodings are widely supported in many programming languages and are not discussed further.

5.1.1.2 Character Encodings

DW_ATE_UTF specifies the Unicode string encoding (see the Universal Character Set standard, ISO/IEC 10646-1:1993).

For example, the C++ type char16_t is represented by a base type entry with a name attribute whose value is "char16_t", an encoding attribute whose value is DW_ATE_UTF and a byte size attribute whose value is 2.

DW_ATE_ASCII and DW_ATE_UCS specify encodings for the Fortran 2003 string kinds `ASCII` (ISO/IEC 646:1991) and `ISO_10646` (UCS-4 in ISO/IEC 10646:2000).

Table 5.1: Encoding attribute values

Name	Meaning
Simple encodings	
DW_ATE_boolean	true or false
DW_ATE_address	linear machine address[a]
DW_ATE_signed	signed binary integer
DW_ATE_signed_char	signed character
DW_ATE_unsigned	unsigned binary integer
DW_ATE_unsigned_char	unsigned character
Character encodings	
DW_ATE_ASCII	ISO/IEC 646:1991 character
DW_ATE_UCS	ISO/IEC 10646-1:1993 character (UCS-4)
DW_ATE_UTF	ISO/IEC 10646-1:1993 character
Scaled encodings	
DW_ATE_signed_fixed	signed fixed-point scaled integer
DW_ATE_unsigned_fixed	unsigned fixed-point scaled integer
Floating-point encodings	
DW_ATE_float	binary floating-point number
DW_ATE_complex_float	complex binary floating-point number
DW_ATE_imaginary_float	imaginary binary floating-point number
DW_ATE_decimal_float	IEEE 754R decimal floating-point number
Decimal string encodings	
DW_ATE_packed_decimal	packed decimal number
DW_ATE_numeric_string	numeric string
DW_ATE_edited	edited string

[a]For segmented addresses, see Section 2.12 on page 48

5.1.1.3 Scaled Encodings

The DW_ATE_signed_fixed and DW_ATE_unsigned_fixed entries describe signed and unsigned fixed-point binary data types, respectively.

The fixed binary type encodings have a DW_AT_digit_count attribute with the same interpretation as described for the DW_ATE_packed_decimal and DW_ATE_numeric_string base type encodings (see Section 5.1.1.5 on the next page).

For a data type with a decimal scale factor, the fixed binary type entry has a DW_AT_decimal_scale attribute with the same interpretation as described for the DW_ATE_packed_decimal and DW_ATE_numeric_string base types (see Section 5.1.1.5).

For a data type with a binary scale factor, the fixed binary type entry has a DW_AT_binary_scale attribute. The DW_AT_binary_scale attribute is an integer constant value that represents the exponent of the base two scale factor to be applied to an instance of the type. Zero scale puts the binary point immediately to the right of the least significant bit. Positive scale moves the binary point to the right and implies that additional zero bits on the right are not stored in an instance of the type. Negative scale moves the binary point to the left; if the absolute value of the scale is larger than the number of bits, this implies additional zero bits on the left are not stored in an instance of the type.

For a data type with a non-decimal and non-binary scale factor, the fixed binary type entry has a DW_AT_small attribute which references a DW_TAG_constant entry. The scale factor value is interpreted in accordance with the value defined by the DW_TAG_constant entry. The value represented is the product of the integer value in memory and the associated constant entry for the type.

The DW_AT_small attribute is defined with the Ada `small` attribute in mind.

5.1.1.4 Floating-Point Encodings

Types with binary floating-point encodings (DW_ATE_float, DW_ATE_complex_float and DW_ATE_imaginary_float) are supported in many programming languages and are not discussed further.

DW_ATE_decimal_float specifies floating-point representations that have a power-of-ten exponent, such as specified in IEEE 754R.

5.1.1.5 Decimal String Encodings

The DW_ATE_packed_decimal and DW_ATE_numeric_string base type encodings represent packed and unpacked decimal string numeric data types, respectively, either of which may be either signed or unsigned. These base types are used in combination with DW_AT_decimal_sign, DW_AT_digit_count and DW_AT_decimal_scale attributes.

A DW_AT_decimal_sign attribute is an integer constant that conveys the representation of the sign of the decimal type (see Table 5.2). Its integer constant value is interpreted to mean that the type has a leading overpunch, trailing overpunch, leading separate or trailing separate sign representation or, alternatively, no sign at all.

Table 5.2: Decimal sign attribute values

Name	Meaning
DW_DS_unsigned	Unsigned
DW_DS_leading_overpunch	Sign is encoded in the most significant digit in a target-dependent manner
DW_DS_trailing_overpunch	Sign is encoded in the least significant digit in a target-dependent manner
DW_DS_leading_separate	Decimal type: Sign is a "+" or "-" character to the left of the most significant digit.
DW_DS_trailing_separate	Decimal type: Sign is a "+" or "-" character to the right of the least significant digit.
	Packed decimal type: Least significant nibble contains a target-dependent value indicating positive or negative.

The DW_AT_decimal_scale attribute is an integer constant value that represents the exponent of the base ten scale factor to be applied to an instance of the type. A scale of zero puts the decimal point immediately to the right of the least significant digit. Positive scale moves the decimal point to the right and implies that additional zero digits on the right are not stored in an instance of the type. Negative scale moves the decimal point to the left; if the absolute value of the scale is larger than the digit count, this implies additional zero digits on the left are not stored in an instance of the type.

The DW_AT_digit_count attribute is an integer constant value that represents the number of digits in an instance of the type.

The DW_ATE_edited base type is used to represent an edited numeric or alphanumeric data type. It is used in combination with a DW_AT_picture_string attribute whose value is a null-terminated string containing the target-dependent picture string associated with the type.

If the edited base type entry describes an edited numeric data type, the edited type entry has a DW_AT_digit_count and a DW_AT_decimal_scale attribute. These attributes have the same interpretation as described for the DW_ATE_packed_decimal and DW_ATE_numeric_string base types. If the edited type entry describes an edited alphanumeric data type, the edited type entry does not have these attributes.

The presence or absence of the DW_AT_digit_count and DW_AT_decimal_scale attributes allows a debugger to easily distinguish edited numeric from edited alphanumeric, although in principle the digit count and scale are derivable by interpreting the picture string.

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.

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 on the following page and Section 5.4 on page 110, 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).

C++ permits using the `auto` return type specifier for the return type of a member function declaration. The actual return type is deduced based on the definition of the function, so it may not be known when the function is declared. The language implementation can provide an unspecified type entry with the name `auto` which can be referenced by the return type attribute of a function declaration entry. When the function is later defined, the DW_TAG_subprogram entry for the definition includes a reference to the actual return type.

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 Table 5.3.

Table 5.3: Type modifier tags

Name	Meaning
DW_TAG_atomic_type	atomic qualified type (for example, in C)
DW_TAG_const_type	const qualified type (for example in C, C++)
DW_TAG_immutable_type	immutable type (for example, in D)
DW_TAG_packed_type	packed type (for example in Ada, Pascal)
DW_TAG_pointer_type	pointer to an object of the type being modified
DW_TAG_reference_type	reference to (lvalue of) an object of the type being modified
DW_TAG_restrict_type	restrict qualified type
DW_TAG_rvalue_reference_type	rvalue reference to an object of the type being modified (for example, in C++)
DW_TAG_shared_type	shared qualified type (for example, in UPC)
DW_TAG_volatile_type	volatile qualified type (for example, in C, C++)

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 name of the modified type.

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 (using DW_TAG_pointer_type, DW_TAG_reference_type or DW_TAG_rvalue_reference_type) may have a DW_AT_address_class attribute to describe how objects having the given pointer or reference type are dereferenced.

A modified type entry describing a UPC shared qualified type (using DW_TAG_shared_type) may have a DW_AT_count attribute whose value is a constant expressing the (explicit or implied) blocksize specified for the type in the source. If no count attribute is present, then the "infinite" blocksize is assumed.

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

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

This represents a volatile pointer to a constant character. It 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)
```

On the other hand

```
    volatile unsigned char * const restrict p;
```

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)
```

Figure 5.1: Type modifier examples

When multiple type modifiers are chained together to modify a base or user-defined type, the tree ordering reflects the semantics of the applicable language rather than the textual order in the source presentation.

Examples of modified types are shown in Figure 5.1.

5.4 Typedef Entries

A named type that is defined in terms of another type definition 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.

The typedef entry may also contain a DW_AT_type attribute whose value is a reference to the type named by the typedef. 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.

Depending on the language, a named type that is defined in terms of another type may be called a type alias, a subtype, a constrained type and other terms. A type name declared with no defining details may be termed an incomplete, forward or hidden type. While the DWARF DW_TAG_typedef entry was originally inspired by the like named construct in C and C++, it is broadly suitable for similar constructs (by whatever source syntax) in other languages.

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.

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 Table 5.4 following. 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.

Table 5.4: Array ordering

DW_ORD_col_major
DW_ORD_row_major

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 either a DW_AT_byte_stride or a DW_AT_bit_stride attribute, whose value (see Section 2.19 on page 55) is the size of each element of the array.

The array type entry may have either a DW_AT_byte_size or a DW_AT_bit_size attribute (see Section 2.21 on page 56), whose value is the amount of storage needed to hold 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 (that is, leftmost dimension first, next to leftmost second, and so on).

In languages that have no concept of a "multidimensional array" (for example, C), an array of arrays may be represented by a debugging information entry for a multidimensional array.

Alternatively, for an array with dynamic rank the array dimensions are described by a debugging information entry with the tag DW_TAG_generic_subrange. This entry has the same attributes as a DW_TAG_subrange_type entry; however, there is just one DW_TAG_generic_subrange entry and it describes all of the dimensions of the array. If DW_TAG_generic_subrange is used, the number of dimensions must be specified using a DW_AT_rank attribute. See also Section 5.18.3 on page 134.

Other attributes especially applicable to arrays are DW_AT_allocated, DW_AT_associated and DW_AT_data_location, which are described in Section 5.18 on page 132. For relevant examples, see also Appendix D.2.1 on page 292.

5.6 Coarray Type Entries

In Fortran, a "coarray" is an array whose elements are located in different processes rather than in the memory of one process. The individual elements of a coarray can be scalars or arrays. Similar to arrays, coarrays have "codimensions" that are indexed using a "coindex" or multiple "coindices".

A coarray type is represented by a debugging information entry with the tag DW_TAG_coarray_type. If a name has been given to the coarray type in the source, then the corresponding coarray type entry has a DW_AT_name attribute whose value is a null-terminated string containing the array type name.

A coarray entry has one or more DW_TAG_subrange_type child entries, one for each codimension. It also has a DW_AT_type attribute describing the type of each element of the coarray.

In a coarray application, the run-time number of processes in the application is part of the coindex calculation. It is represented in the Fortran source by a coindex which is declared with a "" as the upper bound. To express this concept in DWARF, the DW_TAG_subrange_type child entry for that index has only a lower bound and no upper bound.*

How coarray elements are located and how coindices are converted to process specifications is implementation-defined.

5.7 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.

5.7.1 Structure, Union and Class Type Entries

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.

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.

A structure, union, or class type may have a DW_AT_export_symbols attribute which indicates that all member names defined within the structure, union, or class may be referenced as if they were defined within the containing structure, union, or class.

This may be used to describe anonymous structures, unions and classes in C or C++.

A structure type, union type or class type entry may have either a DW_AT_byte_size or a DW_AT_bit_size attribute (see Section 2.21 on page 56), whose value is the amount of storage needed to hold an instance of the structure, union or class type, including any padding.

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 the complete declaration of a type has been placed in a separate type unit (see Section 3.1.4 on page 68), an incomplete declaration of that type in the compilation unit may provide the unique 8-byte signature of the type using a DW_AT_signature attribute.

If a structure, union or class entry represents the definition of a structure, union or class member corresponding to a prior incomplete structure, union or class, the entry may have a DW_AT_specification attribute whose value is a reference to the debugging information entry representing that incomplete declaration.

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 the structure, union or class they represent if such information is already provided in the declaration.

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 has a DW_AT_specification attribute referencing the debugging information entry owned by the body of the structure, union or class 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.1 on page 365).

A structure type, union type or class type entry may have a DW_AT_calling_convention attribute, whose value indicates whether a value of the type is passed by reference or passed by value. The set of calling convention codes for use with types is given in Table 5.5 following.

Table 5.5: Calling convention codes for types

DW_CC_normal
DW_CC_pass_by_value
DW_CC_pass_by_reference

If this attribute is not present, or its value is DW_CC_normal, the convention to be used for an object of the given type is assumed to be unspecified.

Note that DW_CC_normal is also used as a calling convention code for certain subprograms (see Table 3.3 on page 76).

If unspecified, a consumer may be able to deduce the calling convention based on knowledge of the type and the ABI.

5.7.2 Interface Type Entries

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.

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.

5.7.3 Derived or Extended Structures, Classes and Interfaces

In C++, a class (or struct) 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 or struct 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 instance 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 instance of the inherited type. Otherwise, the value must be a location description. In this latter case, the beginning address of the instance of the derived class is pushed on the expression stack before the location description is evaluated and the result of the evaluation is the location of the instance 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 5.7.6 following).

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 a struct, union or 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 description.

5.7.4 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, 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.

5.7.5 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.

5.7.6 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. If the member entry describes an anonymous union, the name attribute is omitted or the value of the attribute 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 member of a class and public access is assumed for an member of a structure, union, or interface.

A data member entry may have a DW_AT_mutable attribute, which is a flag. This attribute indicates whether the data member was declared with the mutable storage class specifier.

The beginning of a data member is described relative to the beginning of the object in which it is immediately contained. In general, the beginning is characterized by both an address and a bit offset within the byte at that address. When the storage for an entity includes all of the bits in the beginning byte, the beginning bit offset is defined to be zero.

The member entry corresponding to a data member that is defined in a structure, union or class may have either a DW_AT_data_member_location attribute or a DW_AT_data_bit_offset attribute. If the beginning of the data member is the same as the beginning of the containing entity then neither attribute is required.

For a DW_AT_data_member_location attribute there are two cases:

1. If the value is an integer constant, it is the offset in bytes from the beginning of the containing entity. If the beginning of the containing entity has a non-zero bit offset then the beginning of the member entry has that same bit offset as well.

2. Otherwise, the value must be a location description. In this case, the beginning of the containing entity must be byte aligned. The beginning address is pushed on the DWARF stack before the location description is evaluated; the result of the evaluation is the base address of the member entry.

The 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 2.5.1.3 on page 29); DW_OP_push_object_address therefore is not needed at the beginning of a location description 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.

A DW_AT_data_member_location attribute that has the form of a location description is not valid for a data member contained in an entity that is not byte aligned because DWARF operations do not allow for manipulating or computing bit offsets.

For a DW_AT_data_bit_offset attribute, the value is an integer constant (see Section 2.19 on page 55) that specifies the number of bits from the beginning of the containing entity to the beginning of the data member. This value must be greater than or equal to zero, but is not limited to less than the number of bits per byte.

If the size of a data member is not the same as the size of the type given for the data member, the data member has either a DW_AT_byte_size or a DW_AT_bit_size attribute whose integer constant value (see Section 2.19 on page 55) is the amount of storage needed to hold the value of the data member.

For showing nested and packed records and arrays, see Appendix D.2.7 on page 309 and D.2.8 on page 311.

5.7.7 Class Variable Entries

A class variable (“static data member” in C++) is a variable shared by all instances of a class. It is represented by a debugging information entry with the tag DW_TAG_variable.

The class variable entry may contain the same attributes and follows the same rules as non-member global variable entries (see Section 4.1 on page 97).

A class variable 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.

5.7.8 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 on page 75).

In particular, if the member function entry is an instantiation of a member function template, it follows the same rules as function template instantiations (see Section 3.3.7 on page 81).

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.

If the member function entry describes an explicit member function, then that entry has a DW_AT_explicit 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 the member function entry describes a non-static member function, then that entry has a DW_AT_object_pointer attribute whose value is a reference to the formal parameter entry that corresponds to the object for which the function is called. The name attribute of that formal parameter is defined by the current language (for example, `this` for C++ or `self` for Objective C and some other languages). That parameter also has a DW_AT_artificial attribute whose value is true.

Conversely, if the member function entry describes a static member function, the entry does not have a DW_AT_object_pointer attribute.

In C++, non-static member functions can have const-volatile qualifiers, which affect the type of the first formal parameter (the "`this`"-pointer).

If the member function entry describes a non-static member function that has a const-volatile qualification, then the entry describes a non-static member function whose object formal parameter has a type that has an equivalent const-volatile qualification.

Beginning in C++11, non-static member functions can also have one of the ref-qualifiers, & and &&. These do not change the type of the "`this`"-pointer, but they do affect the types of object values on which the function can be invoked.

The member function entry may have an DW_AT_reference attribute to indicate a non-static member function that can only be called on lvalue objects, or the DW_AT_rvalue_reference attribute to indicate that it can only be called on prvalues and xvalues.

The lvalue, prvalue and xvalue concepts are defined in the C++11 and later standards.

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 a name attribute giving the name of the function member whose definition they represent. Similarly, such entries do not need to contain a return type attribute, unless the return type on the declaration was unspecified (for example, the declaration used the C++ `auto` return type specifier).

In C++, a member function may be declared as deleted. This prevents the compiler from generating a default implementation of a special member function such as a constructor or destructor, and can affect overload resolution when used on other member functions.

If the member function entry has been declared as deleted, then that entry has a DW_AT_deleted attribute.

In C++, a special member function may be declared as defaulted, which explicitly declares a default compiler-generated implementation of the function. The declaration may have different effects on the calling convention used for objects of its class, depending on whether the default declaration is made inside or outside the class.

If the member function has been declared as defaulted, then the entry has a DW_AT_defaulted attribute whose integer constant value indicates whether, and if so, how, that member is defaulted. The possible values and their meanings are shown in Table 5.6 following.

Table 5.6: Defaulted attribute names

Defaulted attribute name	**Meaning**
DW_DEFAULTED_no	Not declared default
DW_DEFAULTED_in_class	Defaulted within the class
DW_DEFAULTED_out_of_class	Defaulted outside of the class

An artificial member function (that is, a compiler-generated copy that does not appear in the source) does not have a DW_AT_defaulted attribute.

5.7.9 Class Template Instantiations

In C++ a class template is a generic definition of a class type that may be instantiated when an instance of the class is declared or defined. The generic description of the class may include parameterized types, parameterized compile-time constant values, and/or parameterized run-time constant addresses. 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 the following 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. Template parameters are described and referenced as specified in Section 2.23 on page 57.

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

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

5.7.10 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 the discriminant value selecting this variant. 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 discriminant values and discriminant 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 Table 5.7.

Table 5.7: Discriminant descriptor values

DW_DSC_label
DW_DSC_range

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.

5.8 Condition Entries

COBOL has the notion of a "level-88 condition" that associates a data item, called the conditional variable, with a set of one or more constant values and/or value ranges. Semantically, the condition is 'true' if the conditional variable's value matches any of the described constants, and the condition is 'false' otherwise.

The DW_TAG_condition debugging information entry describes a logical condition that tests whether a given data item's value matches one of a set of constant values. If a name has been given to the condition, the condition entry has a DW_AT_name attribute whose value is a null-terminated string giving the condition name.

The condition entry's parent entry describes the conditional variable; normally this will be a DW_TAG_variable, DW_TAG_member or DW_TAG_formal_parameter entry. If the parent entry has an array type, the condition can test any individual element, but not the array as a whole. The condition entry implicitly specifies a "comparison type" that is the type of an array element if the parent has an array type; otherwise it is the type of the parent entry.

The condition entry owns DW_TAG_constant and/or DW_TAG_subrange_type entries that describe the constant values associated with the condition. If any child entry has a DW_AT_type attribute, that attribute describes a type compatible with the comparison type (according to the source language); otherwise the child's type is the same as the comparison type.

For conditional variables with alphanumeric types, COBOL permits a source program to provide ranges of alphanumeric constants in the condition. Normally a subrange type entry does not describe ranges of strings; however, this can be represented using bounds attributes that are references to constant entries describing strings. A subrange type entry may refer to constant entries that are siblings of the subrange type entry.

5.9 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.

The enumeration type entry may have a DW_AT_type attribute which refers to the underlying data type used to implement the enumeration. The entry also may have a DW_AT_byte_size attribute or DW_AT_bit_size attribute, whose value (see Section 2.21 on page 56) is the amount of storage required to hold an instance of the enumeration. If no DW_AT_byte_size or DW_AT_bit_size attribute is present, the size for holding an instance of the enumeration is given by the size of the underlying data type.

If an enumeration type has type safe semantics such that

1. Enumerators are contained in the scope of the enumeration type, and/or
2. Enumerators are not implicitly converted to another type

then the enumeration type entry may have a DW_AT_enum_class attribute, which is a flag. In a language that offers only one kind of enumeration declaration, this attribute is not required.

In C or C++, the underlying type will be the appropriate integral type determined by the compiler from the properties of the enumeration literal values. A C++ type declaration written using enum class declares a strongly typed enumeration and is represented using DW_TAG_enumeration_type in combination with DW_AT_enum_class.

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. 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 either a DW_AT_byte_stride or DW_AT_bit_stride attribute which specifies the separation between successive elements along the dimension as described in Section 2.19 on page 55. The value of the DW_AT_bit_stride attribute is interpreted as bits and the value of the DW_AT_byte_stride attribute is interpreted as bytes.

5.10 Subroutine Type Entries

It is possible in C to declare pointers to subroutines that return a value of a specific type. In both 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.

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 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, which is a flag.

Each debugging information entry owned by a subroutine type entry corresponds to either a formal parameter or the sequence of unspecified parameters of the subprogram type:

1. A formal parameter of a parameter list (that has a specific type) is 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.

C++ const-volatile qualifiers are encoded as part of the type of the "`this`"-pointer. C++11 reference and rvalue-reference qualifiers are encoded using the DW_AT_reference and DW_AT_rvalue_reference attributes, respectively. See also Section 5.7.8 on page 120.

A subroutine type entry may have the DW_AT_reference or DW_AT_rvalue_reference attribute to indicate that it describes the type of a member function with reference or rvalue-reference semantics, respectively.

5.11 String Type Entries

A "string" is a sequence of characters that have specific semantics and operations that distinguish 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.

A string type entry may have a DW_AT_type attribute describing how each character is encoded and is to be interpreted. The value of this attribute is a reference to a DW_TAG_base_type base type entry. If the attribute is absent, then the character is encoded using the system default.

The Fortran 2003 language standard allows string types that are composed of different types of (same sized) characters. While there is no standard list of character kinds, the kinds `ASCII` (see DW_ATE_ASCII), `ISO_10646` (see DW_ATE_UCS) and `DEFAULT` are defined.

The string type entry may have a DW_AT_byte_size attribute or DW_AT_bit_size attribute, whose value (see Section 2.21 on page 56) is the amount of storage needed to hold a value of the string type.

The string type entry may also have a DW_AT_string_length attribute whose value is either a reference (see Section 2.19) yielding the length of the string or a location description yielding the location where the length of the string is stored in the program. If the DW_AT_string_length attribute is not present, the size of the string is assumed to be the amount of storage that is allocated for the string (as specified by the DW_AT_byte_size or DW_AT_bit_size attribute).

The string type entry may also have a DW_AT_string_length_byte_size or DW_AT_string_length_bit_size attribute, whose value (see Section 2.21 on page 56) is the size of the data to be retrieved from the location referenced by the DW_AT_string_length attribute. If no byte or bit 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.

Prior to DWARF Version 5, the meaning of a DW_AT_byte_size attribute depended on the presence of the DW_AT_string_length attribute:

- *If DW_AT_string_length was present, DW_AT_byte_size specified the size of the length data to be retrieved from the location specified by the DW_AT_string_length attribute.*
- *If DW_AT_string_length was not present, DW_AT_byte_size specified the amount of storage allocated for objects of the string type.*

In DWARF Version 5, DW_AT_byte_size always specifies the amount of storage allocated for objects of the string type.

5.12 Set Type 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.

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 either a DW_AT_byte_size attribute, or DW_AT_bit_size attribute whose value (see Section 2.21 on page 56) is the amount of storage needed to hold a value of the set type.

5.13 Subrange Type Entries

Several languages support the concept of a "subrange" type. Objects of the subrange type can represent only a contiguous subset (range) of values from the type on which the subrange is defined. Subrange types 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.

The tag DW_TAG_generic_subrange is used to describe arrays with a dynamic rank. See Section 5.5 on page 111.

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 or DW_AT_bit_size attribute, whose value (see Section 2.19 on page 55) is the amount of storage needed to hold a value of the subrange type.

The subrange entry may have a DW_AT_threads_scaled attribute, which is a flag. If present, this attribute indicates whether this subrange represents a UPC array bound which is scaled by the runtime `THREADS` value (the number of UPC threads in this execution of the program).

This allows the representation of a UPC shared array such as

```
int shared foo[34*THREADS][10][20];
```

The subrange entry may have the attributes DW_AT_lower_bound and DW_AT_upper_bound to specify, 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 2.19 on page 55.

If the lower bound value is missing, the value is assumed to be a language-dependent default constant as defined in Table 7.17 on page 230.

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 determined as follows:

1. If there is a lower bound attribute that references an object, the basis type is assumed to be the same as the type of that object.
2. Otherwise, if there is an upper bound or count attribute that references an object, the basis type is assumed to be the same as the type of that object.
3. Otherwise, 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 either a DW_AT_byte_stride or DW_AT_bit_stride attribute which specifies the separation between successive elements along the dimension as described in Section 2.21 on page 56.

Note that the stride can be negative.

5.14 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.

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 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.

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 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 description is used in conjunction with the location descriptions 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 DWARF expression stack before the DW_AT_use_location description 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 DWARF expression stack.*
2. *Push the base address of `object` onto the DWARF expression stack.*
3. *Evaluate the DW_AT_use_location description given in the type of `mbr_ptr`.*

5.15 File Type Entries

Some languages, such as Pascal, provide a 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.

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 or DW_AT_bit_size attribute, whose value (see Section 2.19 on page 55) is the amount of storage need to hold a value of the file type.

5.16 Dynamic Type Entries

Some languages such as Fortran 90, provide types whose values may be dynamically allocated or associated with a variable under explicit program control. However, unlike the pointer type in C or C++, the indirection involved in accessing the value of the variable is generally implicit, that is, not indicated as part of the program source.

A dynamic type entry is used to declare a dynamic type that is "just like" another non-dynamic type without needing to replicate the full description of that other type.

A dynamic type is represented by a debugging information entry with the tag DW_TAG_dynamic_type. If a name has been given to the dynamic type, then the dynamic type has a DW_AT_name attribute whose value is a null-terminated string containing the dynamic type name.

A dynamic type entry has a DW_AT_type attribute whose value is a reference to the type of the entities that are dynamically allocated.

A dynamic type entry also has a DW_AT_data_location, and may also have DW_AT_allocated and/or DW_AT_associated attributes as described in Section 5.18. A DW_AT_data_location, DW_AT_allocated or DW_AT_associated attribute may not occur on a dynamic type entry if the same kind of attribute already occurs on the type referenced by the DW_AT_type attribute.

5.17 Template Alias Entries

In C++, a template alias is a form of typedef that has template parameters. DWARF does not represent the template alias definition but does represent instantiations of the alias.

A type named using a template alias is represented by a debugging information entry with the tag DW_TAG_template_alias. The template alias entry has a DW_AT_name attribute whose value is a null-terminated string containing the name of the template alias. The template alias entry has child entries describing the template actual parameters (see Section 2.23 on page 57).

5.18 Dynamic Properties of Types

The DW_AT_data_location, DW_AT_allocated and DW_AT_associated attributes described in this section are motivated for use with DW_TAG_dynamic_type entries but can be used for any other type as well.

5.18.1 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 description. The result of evaluating this description yields the location 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 description 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 on page 292.

5.18.2 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 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 integer 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 integer value of the attribute (see below) indicates whether an object of the type is currently associated or not.

The value of these attributes is determined as described in Section 2.19 on page 55. A non-zero value is interpreted as allocated or associated, and zero is interpreted as not allocated or not associated.

For Fortran 90, if the DW_AT_associated attribute is present, the type has the POINTER property where either the parent variable is never associated with a dynamic object or the implementation does not track whether the associated object is static or dynamic. If the DW_AT_allocated attribute is present and the DW_AT_associated attribute is not, the type has the ALLOCATABLE property. 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 association 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 on page 292.

5.18.3 Array Rank

The Fortran language supports "assumed-rank arrays". The rank (the number of dimensions) of an assumed-rank array is unknown at compile time. The Fortran runtime stores the rank in an array descriptor.

The presence of the attribute indicates that an array's rank (number of dimensions) is dynamic, and therefore unknown at compile time. The value of the DW_AT_rank attribute is either an integer constant or a DWARF expression whose evaluation yields the dynamic rank.

The bounds of an array with dynamic rank are described using a DW_TAG_generic_subrange entry, which is the dynamic rank array equivalent of DW_TAG_subrange_type. The difference is that a DW_TAG_generic_subrange entry contains generic lower/upper bound and stride expressions that need to be evaluated for each dimension. Before any expression contained in a DW_TAG_generic_subrange can be evaluated, the dimension for which the expression is to be evaluated needs to be pushed onto the stack. The expression will use it to find the offset of the respective field in the array descriptor metadata.

A producer is free to choose any layout for the array descriptor. In particular, the upper and lower bounds and stride values do not need to be bundled into a structure or record, but could be laid end to end in the containing descriptor, pointed to by the descriptor, or even allocated independently of the descriptor.

Dimensions are enumerated 0 to $rank - 1$ in source program order.

For an example in Fortran 2008, see Section D.2.3 on page 301.

Chapter 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 a `.debug_info` section.

In the descriptions that follow, these 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 on page 184 and 7.4 on page 196.
- sbyte, ubyte, uhalf and uword, which are defined in Section 7.31 on page 245.

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 the debugger 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 required 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.

To find the debugging information associated with a subroutine, given an address, 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, given an address, 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 (including data objects, functions and types) by name or by address faster, a producer of DWARF information may provide two different types of tables containing information about the debugging information entries owned by a particular compilation unit entry in a more condensed format.

6.1.1 Lookup by Name

For lookup by name, a name index is maintained in a separate object file section named `.debug_names`.

The `.debug_names` section is new in DWARF Version 5, and supersedes the `.debug_pubnames` and `.debug_pubtypes` sections of earlier DWARF versions. While `.debug_names` and either `.debug_pubnames` and/or `.debug_pubtypes` sections cannot both occur in the same compilation unit, both may be found in the set of units that make up an executable or shared object.

The index consists primarily of two parts: a list of names, and a list of index entries. A name, such as a subprogram name, type name, or variable name, may have several defining declarations in the debugging information. In this case, the entry for that name in the list of names will refer to a sequence of index entries in the second part of the table, each corresponding to one defining declaration in the `.debug_info` section.

The name index may also contain an optional hash table for faster lookup.

A relocatable object file may contain a "per-CU" index, which provides an index to the names defined in that compilation unit.

An executable or shareable object file may contain either a collection of "per-CU" indexes, simply copied from each relocatable object file, or the linker may produce a "per-module" index by combining the per-CU indexes into a single index that covers the entire load module.

6.1.1.1 Contents of the Name Index

The name index must contain an entry for each debugging information entry that defines a named subprogram, label, variable, type, or namespace, subject to the following rules:

- All non-defining declarations (that is, debugging information entries with a DW_AT_declaration attribute) are excluded.
- DW_TAG_namespace debugging information entries without a DW_AT_name attribute are included with the name "`(anonymous namespace)`".
- All other debugging information entries without a DW_AT_name attribute are excluded.
- DW_TAG_subprogram, DW_TAG_inlined_subroutine, and DW_TAG_label debugging information entries without an address attribute (DW_AT_low_pc, DW_AT_high_pc, DW_AT_ranges, or DW_AT_entry_pc) are excluded.
- DW_TAG_variable debugging information entries with a DW_AT_location attribute that includes a DW_OP_addr or DW_OP_form_tls_address operator are included; otherwise, they are excluded.
- If a subprogram or inlined subroutine is included, and has a DW_AT_linkage_name attribute, there will be an additional index entry for the linkage name.

For the purposes of determining whether a debugging information entry has a particular attribute (such as DW_AT_name), if debugging information entry A has a DW_AT_specification or DW_AT_abstract_origin attribute pointing to another debugging information entry B, any attributes of B are considered to be part of A.

The intent of the above rules is to provide the consumer with some assurance that looking up an unqualified name in the index will yield all relevant debugging information entries that provide a defining declaration at global scope for that name.

A producer may choose to implement additional rules for what names are placed in the index, and may communicate those rules to a cooperating consumer via an augmentation string, described below.

6.1.1.2 Structure of the Name Index

Logically, the name index can be viewed as a list of names, with a list of index entries for each name. Each index entry corresponds to a debugging information entry that matches the criteria given in the previous section. For example, if one compilation unit has a function named `fred` and another has a struct named `fred`, a lookup for "fred" will find the list containing those two index entries.

The index section contains eight individual parts, as illustrated in Figure 6.1 following.

1. A header, describing the layout of the section.
2. A list of compile units (CUs) referenced by this index.
3. A list of local type units (TUs) referenced by this index that are present in this object file.
4. A list of foreign type units (TUs) referenced by this index that are not present in this object file (that is, that have been placed in a split DWARF object file as described in 7.3.2 on page 187).
5. An optional hash lookup table.
6. The name table.
7. An abbreviations table, similar to the one used by the `.debug_info` section.
8. The entry pool, containing a list of index entries for each name in the name list.

The formats of the header and the hash lookup table are described in Section 6.1.1.4 on page 143.

The list of CUs and the list of local TUs are each an array of offsets, each of which is the offset of a compile unit or a type unit in the `.debug_info` section. For a per-CU index, there is a single CU entry, and there may be a TU entry for each type unit generated in the same translation unit as the single CU. For a per-module index, there will be one CU entry for each compile unit in the module, and one TU entry for each unique type unit in the module. Each list is indexed starting at 0.

The list of foreign TUs is an array of 64-bit (DW_FORM_ref_sig8) type signatures, representing types referenced by the index whose definitions have been placed in a different object file (that is, a split DWARF object). This list may be empty. The foreign TU list immediately follows the local TU list and they both use the same index, so that if there are N local TU entries, the index for the first foreign TU is N.

part 1 of 2

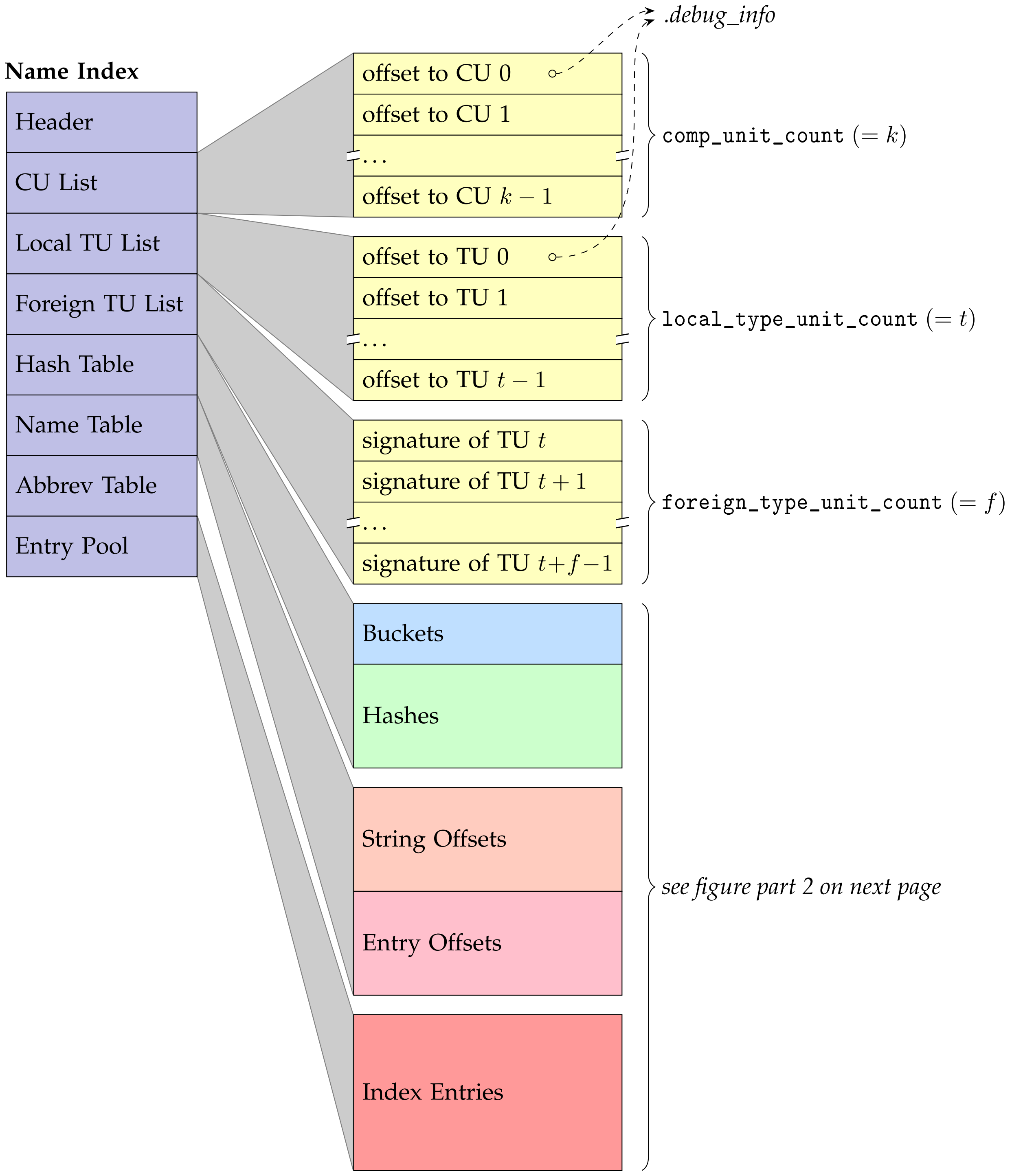
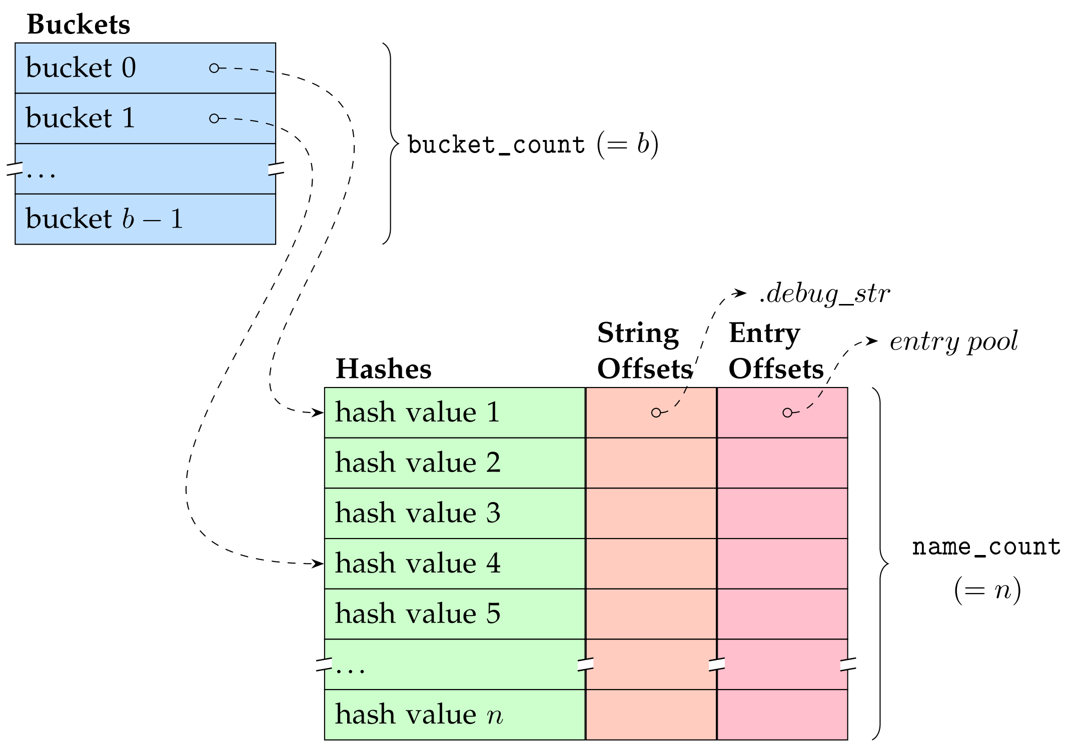


Figure 6.1: Name Index Layout

part 2 of 2



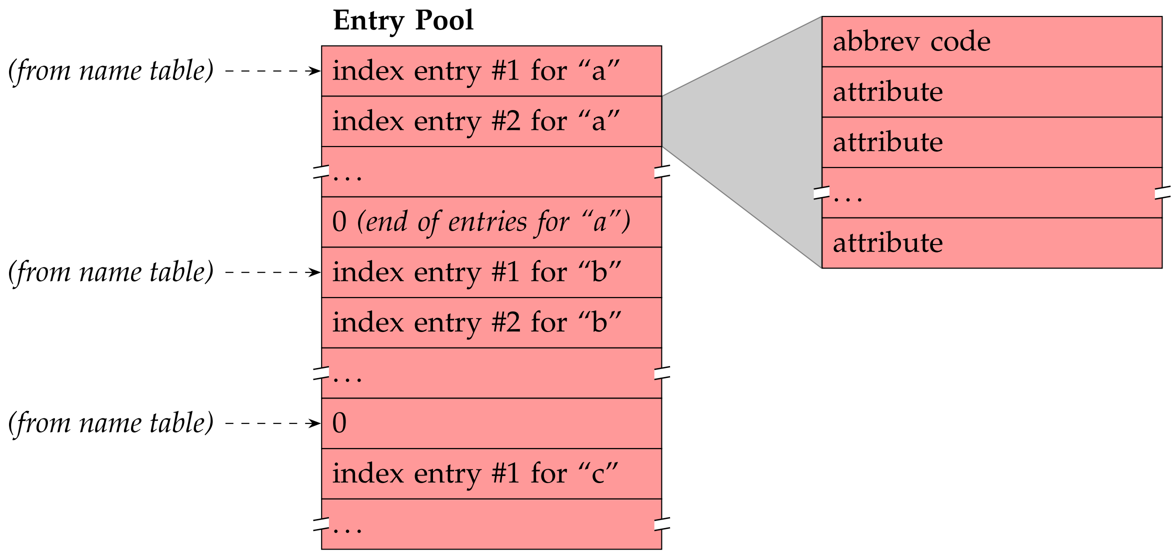


Figure 6.1: Name Index Layout *(concluded)*

The name table is logically a table with a row for each unique name in the index, and two columns. The first column contains a reference to the name, as a string. The second column contains the offset within the entry pool of the list of index entries for the name.

The abbreviations table describes the formats of the entries in the entry pool. Like the DWARF abbreviations table in the `.debug_abbrev` section, it defines one or more abbreviation codes. Each abbreviation code provides a DWARF tag value followed by a list of pairs that defines an attribute and form code used by entries with that abbreviation code.

The entry pool contains all the index entries, grouped by name. The second column of the name list points to the first index entry for the name, and all the index entries for that name are placed one after the other.

Each index entry begins with an unsigned LEB128 abbreviation code. The abbreviation list for that code provides the DWARF tag value for the entry as well as the set of attributes provided by the entry and their forms.

The standard attributes are:

- Compilation Unit (CU), a reference to an entry in the list of CUs. In a per-CU index, index entries without this attribute implicitly refer to the single CU.
- Type Unit (TU), a reference to an entry in the list of local or foreign TUs.
- Debugging information entry offset within the CU or TU.
- Parent debugging information entry, a reference to the index entry for the parent. This is represented as the offset of the entry relative to the start of the entry pool.
- Type hash, an 8-byte hash of the type declaration.

It is possible that an indexed debugging information entry has a parent that is not indexed (for example, if its parent does not have a name attribute). In such a case, a parent attribute may point to a nameless index entry (that is, one that cannot be reached from any entry in the name table), or it may point to the nearest ancestor that does have an index entry.

A producer may define additional vendor-specific attributes, and a consumer will be able to ignore and skip over any attributes it is not prepared to handle.

When an index entry refers to a foreign type unit, it may have attributes for both CU and (foreign) TU. For such entries, the CU attribute gives the consumer a reference to the CU that may be used to locate a split DWARF object file that contains the type unit.

The type hash attribute, not to be confused with the type signature for a TU, may be provided for type entries whose declarations are not in a type unit, for the convenience of link-time or post-link utilities that wish to de-duplicate type declarations across compilation units. The type hash, however, is computed by the same method as specified for type signatures.

The last entry for each name is followed by a zero byte that terminates the list. There may be gaps between the lists.

6.1.1.3 Per-CU versus Per-Module Indexes

In a per-CU index, the CU list may have only a single entry, and index entries may omit the CU attribute. (Cross-module or link-time optimization, however, may produce an object file with several compile units in one object. A compiler in this case may produce a separate index for each CU, or a combined index for all CUs. In the latter case, index entries will require the CU attribute.) Most name table entries may have only a single index entry for each, but sometimes a name may be used in more than one context and will require multiple index entries, each pointing to a different debugging information entry.

When linking object files containing per-CU indexes, the linker may choose to concatenate the indexes as ordinary sections, or it may choose to combine the input indexes into a single per-module index.

A per-module index will contain a number of CUs, and each index entry contains a CU attribute or a TU attribute to identify which CU or TU contains the debugging information entry being indexed. When a given name is used in multiple CUs or TUs, it will typically have a series of index entries pointing to each CU or TU where it is declared. For example, an index entry for a C++ namespace needs to list each occurrence, since each CU may contribute additional names to the namespace, and the consumer needs to find them all. On the other hand, some index entries do not need to list more than one definition; for example, with the one-definition rule in C++, duplicate entries for a function may be omitted, since the consumer only needs to find one declaration. Likewise, a per-module index needs to list only a single copy of a type declaration contained in a type unit.

For the benefit of link-time or post-link utilities that consume per-CU indexes and produce a per-module index, the per-CU index entries provide the tag encoding for the original debugging information entry, and may provide a type hash for certain types that may benefit from de-duplication. For example, the standard declaration of the typedef `uint32_t` is likely to occur in many CUs, but a combined per-module index needs to retain only one; a user declaration of a typedef `mytype` may refer to a different type at each occurrence, and a combined per-module index retains each unique declaration of that type.

6.1.1.4 Data Representation of the Name Index

The name index is placed in a section named `.debug_names`, and consists of the eight parts described in the following sections.

6.1.1.4.1 Section Header

The section header contains the following fields:

1. `unit_length` (initial length)
   The length of this contribution to the name index section, not including the length field itself.
2. `version` (uhalf)
   A version number (see Section 7.19 on page 234). This number is specific to the name index table and is independent of the DWARF version number.
3. *padding* (uhalf)
   Reserved to DWARF (must be zero).
4. `comp_unit_count` (uword)
   The number of CUs in the CU list.
5. `local_type_unit_count` (uword)
   The number of TUs in the local TU list.
6. `foreign_type_unit_count` (uword)
   The number of TUs in the foreign TU list.
7. `bucket_count` (uword)
   The number of hash buckets in the hash lookup table. If there is no hash lookup table, this field contains 0.
8. `name_count` (uword)
   The number of unique names in the index.
9. `abbrev_table_size` (uword)
   The size in bytes of the abbreviations table.

10. `augmentation_string_size` (uword)
    The size in bytes of the augmentation string. This value is rounded up to a multiple of 4.

11. `augmentation_string` (sequence of UTF-8 characters)
    A vendor-specific augmentation string, which provides additional information about the contents of this index. If provided, the string begins with a 4-character vendor ID. The remainder of the string is meant to be read by a cooperating consumer, and its contents and interpretation are not specified here. The string is padded with null characters to a multiple of four bytes in length.

    The presence of an unrecognised augmentation string does not make it impossible for a consumer to process data in the `.debug_names` section. The augmentation string only provides hints to the consumer regarding the completeness of the set of names in the index.

6.1.1.4.2 List of CUs

The list of CUs immediately follows the header. Each entry in the list is an offset of the corresponding compilation unit in the `.debug_info` section. In the DWARF-32 format, a section offset is 4 bytes, while in the DWARF-64 format, a section offset is 8 bytes.

The total number of entries in the list is given by `comp_unit_count`. There must be at least one CU.

6.1.1.4.3 List of Local TUs

The list of local TUs immediately follows the list of CUs. Each entry in the list is an offset of the corresponding type unit in the `.debug_info` section. In the DWARF-32 format, a section offset is 4 bytes, while in the DWARF-64 format, a section offset is 8 bytes.

The total number of entries in the list is given by `local_type_unit_count`. This list may be empty.

6.1.1.4.4 List of Foreign TUs

The list of foreign TUs immediately follows the list of local TUs. Each entry in the list is a 8-byte type signature (as described by DW_FORM_ref_sig8).

The number of entries in the list is given by `foreign_type_unit_count`. This list may be empty.

6.1.1.4.5 Hash Lookup Table

The optional hash lookup table immediately follows the list of type signatures.

The hash lookup table is actually two separate arrays: an array of buckets, followed immediately by an array of hashes. The number of entries in the buckets array is given by `bucket_count`, and the number of entries in the hashes array is given by `name_count`. Each array contains 4-byte unsigned integers.

Symbols are entered into the hash table by first computing a hash value from the symbol name. The hash is computed using the "DJB" hash function described in Section 7.33 on page 250. Given a hash value for the symbol, the symbol is entered into a bucket whose index is the hash value modulo `bucket_count`. The buckets array is indexed starting at 0.

For the purposes of the hash computation, each symbol name should be folded according to the simple case folding algorithm defined in the "Caseless Matching" subsection of Section 5.18 ("Case Mappings") of the Unicode Standard, Version 9.0.0. The original symbol name, as it appears in the source code, should be stored in the name table.name index!case folding

Thus, two symbols that differ only by case will hash to the same slot, but the consumer will be able to distinguish the names when appropriate.

The simple case folding algorithm is further described in the CaseFolding.txt file distributed with the Unicode Character Database. That file defines four classes of mappings: Common (C), Simple (S), Full (F), and Turkish (T). The hash computation specified here uses the C + S mappings only, which do not affect the total length of the string, with the addition that Turkish upper case dotted 'İ' and lower case dotless 'ı' are folded to the Latin lower case 'i'.

Each bucket contains the index of an entry in the hashes array. The hashes array is indexed starting at 1, and an empty bucket is represented by the value 0.

The hashes array contains a sequence of the full hash values for each symbol. All symbols that have the same index into the bucket list follow one another in the hashes array, and the indexed entry in the bucket list refers to the first symbol. When searching for a symbol, the search starts at the index given by the bucket, and continues either until a matching symbol is found or until a hash value from a different bucket is found. If two different symbol names produce the same hash value, that hash value will occur twice in the hashes array. Thus, if a matching hash value is found, but the name does not match, the search continues visiting subsequent entries in the hashes table.

When a matching hash value is found in the hashes array, the index of that entry in the hashes array is used to find the corresponding entry in the name table.

6.1.1.4.6 Name Table

The name table immediately follows the hash lookup table. It consists of two arrays: an array of string offsets, followed immediately by an array of entry offsets. The items in both arrays are section offsets: 4-byte unsigned integers for the DWARF-32 format or 8-byte unsigned integers for the DWARF-64 format. The string offsets in the first array refer to names in the `.debug_str` (or `.debug_str.dwo`) section. The entry offsets in the second array refer to index entries, and are relative to the start of the entry pool area.

These two arrays are indexed starting at 1, and correspond one-to-one with each other. The length of each array is given by `name_count`.

If there is a hash lookup table, the hashes array corresponds on a one-to-one basis with the string offsets array and with the entry offsets array.

If there is no hash lookup table, there is no ordering requirement for the name table.

6.1.1.4.7 Abbreviations Table

The abbreviations table immediately follows the name table. This table consists of a series of abbreviation declarations. Its size is given by `abbrev_table_size`.

Each abbreviation declaration defines the tag and other attributes for a particular form of index entry. Each declaration starts with an unsigned LEB128 number representing the abbreviation code itself. It is this code that appears at the beginning of an index entry. The abbreviation code must not be 0.

The abbreviation code is followed by another unsigned LEB128 number that encodes the tag of the debugging information entry corresponding to the index entry.

Following the tag encoding is a series of attribute specifications. Each attribute consists of two parts: an unsigned LEB128 number that represents the index attribute, and another unsigned LEB128 number that represents the attribute's form (as described in Section 7.5.4 on page 207). The series of attribute specifications ends with an entry containing 0 for the attribute and 0 for the form.

The index attributes and their meanings are listed in Table 6.1 on the next page.

The abbreviations table ends with an entry consisting of a single 0 byte for the abbreviation code. The size of the table given by `abbrev_table_size` may include optional padding following the terminating 0 byte.

Table 6.1: Index attribute encodings

Attribute name	Meaning
DW_IDX_compile_unit	Index of CU
DW_IDX_type_unit	Index of TU (local or foreign)
DW_IDX_die_offset	Offset of DIE within CU or TU
DW_IDX_parent	Index of name table entry for parent
DW_IDX_type_hash	Hash of type declaration

6.1.1.4.8 Entry Pool

The entry pool immediately follows the abbreviations table. Each entry in the entry offsets array in the name table (see Section 6.1.1.4.6) points to an offset in the entry pool, where a series of index entries for that name is located.

Each index entry in the series begins with an abbreviation code, and is followed by the attributes described by the abbreviation declaration for that code. The last index entry in the series is followed by a terminating entry whose abbreviation code is 0.

Gaps are not allowed between entries in a series (that is, the entries for a single name must all be contiguous), but there may be gaps between series.

For example, a producer/consumer combination may find it useful to maintain alignment.

The size of the entry pool is the remaining size of the contribution to the index section, as defined by the `unit_length` header field.

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 this contribution to the address lookup section, not including the length field itself.

2. `version` (uhalf)
   A version number (see Section 7.21 on page 235). 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 of an address in bytes on the target architecture. For segmented addressing, this is the size of the offset portion of the address.
5. `segment_selector_size` (ubyte)
   The size of a segment selector in bytes 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 triple consisting of a segment selector, the beginning address within that segment 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 three zeroes. When the `segment_selector_size` value is zero in the header, the segment selector is omitted so that each descriptor is just a pair, including the terminating entry. 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 needs to know how to associate locations in the source files with the corresponding machine instruction addresses in the executable or the shared object files used by that executable object file. Such an association makes it possible for the debugger user to specify machine instruction addresses in terms of source locations. This is 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.

Line number information generated for a compilation unit is represented in the `.debug_line` section of an object file, and optionally also in the `.debug_line_str` section, and is referenced by a corresponding compilation unit debugging information entry (see Section 3.1.1 on page 60) 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, source column and discriminator 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. To the line number information a consumer must "run" the state machine to generate the matrix for each compilation unit of interest. The concept of an encoded matrix also leaves room for expansion. In the future, columns can be added to the matrix to encode other things that are related to individual instruction addresses.

6.2.1 Definitions

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

state machine	The hypothetical machine used by a consumer of the line number information to expand the byte-coded instruction stream into a matrix of line number information.
line number program	A series of byte-coded line number information instructions representing one compilation unit.
basic block	A sequence of instructions where only the first instruction may be a branch target and only the last instruction may transfer control. A subprogram invocation is defined to be an exit from a basic block. *A basic block does not necessarily correspond to a specific source code construct.*
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 a number of registers as shown in Table 6.3 following.

Table 6.3: State machine registers

Register name	**Meaning**
`address`	The program-counter value corresponding to a machine instruction generated by the compiler.
`op_index`	An unsigned integer representing the index of an operation within a VLIW instruction. The index of the first operation is 0. For non-VLIW architectures, this register will always be 0.

Continued on next page

Register name	**Meaning**
`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. `end_sequence` terminates a sequence of lines; therefore other information in the same row is not meaningful.
`prologue_end`	A boolean indicating that the current address is one (of possibly many) where execution should be suspended for a breakpoint at the entry of a function.
`epilogue_begin`	A boolean indicating that the current address is one (of possibly many) where execution should be suspended for a breakpoint just prior to the exit of a function.

Continued on next page

Register name	Meaning
`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.*
`discriminator`	An unsigned integer identifying the block to which the current instruction belongs. Discriminator values are assigned arbitrarily by the DWARF producer and serve to distinguish among multiple blocks that may all be associated with the same source file, line, and column. Where only one block exists for a given source position, the discriminator value is be zero.

The `address` and `op_index` registers, taken together, form an operation pointer that can reference any individual operation within the instruction stream.

At the beginning of each sequence within a line number program, the state of the registers is as show in Table 6.4 on the following page.

The `isa` value 0 specifies that the instruction set is the architecturally determined default instruction set. This may be fixed by the ABI, or it may be specified by other means, for example, by the object file description.

6.2.3 Line Number Program Instructions

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

1. special opcodes
   These have a ubyte opcode field and no operands.

   Most of the instructions in a line number program are special opcodes.

Table 6.4: Line number program initial state

`address`	0
`op_index`	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`	0
`discriminator`	0

2. 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 Section 6.2.5.2 on page 162). 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.

3. 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 length field itself (see Section 7.2.2 on page 184).

2. `version` (uhalf)
   A version number (see Section 7.22 on page 236). This number is specific to the line number information and is independent of the DWARF version number.

3. `address_size` (ubyte)
   A 1-byte unsigned integer containing the size in bytes of an address (or offset portion of an address for segmented addressing) on the target system.

   The `address_size` field is new in DWARF Version 5. It is needed to support the common practice of stripping all but the line number sections (`.debug_line` and `.debug_line_str`) from an executable.

4. `segment_selector_size` (ubyte)
   A 1-byte unsigned integer containing the size in bytes of a segment selector on the target system.

   The `segment_selector_size` field is new in DWARF Version 5. It is needed in combination with the `address_size` field to accurately characterize the address representation on the target system.

5. `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 on page 196).

6. `minimum_instruction_length` (ubyte)
   The size in bytes of the smallest target machine instruction. Line number program opcodes that alter the `address` and `op_index` registers use this and `maximum_operations_per_instruction` in their calculations.

7. `maximum_operations_per_instruction` (ubyte)
   The maximum number of individual operations that may be encoded in an instruction. Line number program opcodes that alter the `address` and `op_index` registers use this and `minimum_instruction_length` in their calculations.

   For non-VLIW architectures, this field is 1, the `op_index` register is always 0, and the operation pointer is simply the `address` register.

8. `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.

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

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

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

    Opcode base is typically one greater than the highest-numbered standard opcode defined for the specified version of the line number information (12 in DWARF Versions 3, 4 and 5, and 9 in Version 2). If opcode_base is less than the typical value, then standard opcode numbers greater than or equal to the opcode base are not used in the line number table of this unit (and the codes are treated as special opcodes). If `opcode_base` is greater than the typical value, then the numbers between that of the highest standard opcode and the first special opcode (not inclusive) are used for vendor specific extensions.

12. `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.

The remaining fields provide information about the source files used in the compilation. These fields have been revised in DWARF Version 5 to support these goals:

- *To allow new alternative means for a consumer to check that a file it can access is the same version as that used in the compilation.*
- *To allow a producer to collect file name strings in a new section (`.debug_line_str`) that can be used to merge duplicate file name strings.*
- *To add the ability for producers to provide vendor-defined information that can be skipped by a consumer that is unprepared to process it.*

13. `directory_entry_format_count` (ubyte)
    A count of the number of entries that occur in the following `directory_entry_format` field.

14. `directory_entry_format` (sequence of ULEB128 pairs)
    A sequence of directory entry format descriptions. Each description consists of a pair of ULEB128 values:

    - A content type code (see Sections 6.2.4.1 on page 158 and 6.2.4.2 on page 159).
    - A form code using the attribute form codes

15. `directories_count` (ULEB128)
    A count of the number of entries that occur in the following directories field.

16. `directories` (sequence of directory names)
    A sequence of directory names and optional related information. Each entry is encoded as described by the `directory_entry_format` field.

    Entries in this sequence describe each path that was searched for included source files in this compilation, including the compilation directory of the compilation. (The paths include those directories specified by the user for the compiler to search and those the compiler searches without explicit direction.)

The first entry is the current directory of the compilation. Each additional path entry is either a full path name or is relative to the current directory of the compilation.

The line number program assigns a number (index) to each of the directory entries in order, beginning with 0.

Prior to DWARF Version 5, the current directory was not represented in the directories field and a directory index of 0 implicitly referred to that directory as found in the DW_AT_comp_dir attribute of the compilation unit debugging information entry. In DWARF Version 5, the current directory is explicitly present in the directories field. This is needed to support the common practice of stripping all but the line number sections (`.debug_line` and `.debug_line_str`) from an executable.

Note that if a `.debug_line_str` section is present, both the compilation unit debugging information entry and the line number header can share a single copy of the current directory name string.

17. `file_name_entry_format_count` (ubyte)
    A count of the number of file entry format entries that occur in the following `file_name_entry_format` field. If this field is zero, then the `file_names_count` field (see below) must also be zero.

18. `file_name_entry_format` (sequence of ULEB128 pairs)
    A sequence of file entry format descriptions. Each description consists of a pair of ULEB128 values:

    - A content type code (see below)
    - A form code using the attribute form codes

19. `file_names_count` (ULEB128)
    A count of the number of file name entries that occur in the following `file_names` field.

20. `file_names` (sequence of file name entries)
    A sequence of file names and optional related information. Each entry is encoded as described by the `file_name_entry_format` field.

    Entries in this sequence describe source files that contribute to the line number information for this compilation or is used in other contexts, such as in a declaration coordinate or a macro file inclusion.

    The first entry in the sequence is the primary source file whose file name exactly matches that given in the DW_AT_name attribute in the compilation unit debugging information entry.

The line number program references file names in this sequence beginning with 0, and uses those numbers instead of file names in the line number program that follows.

Prior to DWARF Version 5, the current compilation file name was not represented in the `file_names` field. In DWARF Version 5, the current compilation file name is explicitly present and has index 0. This is needed to support the common practice of stripping all but the line number sections (`.debug_line` and `.debug_line_str`) from an executable.

Note that if a `.debug_line_str` section is present, both the compilation unit debugging information entry and the line number header can share a single copy of the current file name string.

6.2.4.1 Standard Content Descriptions

DWARF-defined content type codes are used to indicate the type of information that is represented in one component of an include directory or file name description. The following type codes are defined.

1. DW_LNCT_path
   The component is a null-terminated path name string. If the associated form code is DW_FORM_string, then the string occurs immediately in the containing `directories` or `file_names` field. If the form code is DW_FORM_line_strp, DW_FORM_strp or DW_FORM_strp_sup, then the string is included in the `.debug_line_str`, `.debug_str` or supplementary string section, respectively, and its offset occurs immediately in the containing `directories` or `file_names` field.

   In the 32-bit DWARF format, the representation of a DW_FORM_line_strp value is a 4-byte unsigned offset; in the 64-bit DWARF format, it is an 8-byte unsigned offset (see Section 7.4 on page 196).

   Note that this use of DW_FORM_line_strp is similar to DW_FORM_strp but refers to the `.debug_line_str` section, not `.debug_str`. It is needed to support the common practice of stripping all but the line number sections (`.debug_line` and `.debug_line_str`) from an executable.

   In a `.debug_line.dwo` section, the forms DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 and DW_FORM_strx4 may also be used. These refer into the `.debug_str_offsets.dwo` section (and indirectly also the `.debug_str.dwo` section) because no `.debug_line_str_offsets.dwo` or `.debug_line_str.dwo` sections exist or are defined for use in split objects. (The form DW_FORM_string may also be used, but this precludes the benefits of string sharing.)

2. DW_LNCT_directory_index
   The unsigned directory index represents an entry in the directories field of the header. The index is 0 if the file was found in the current directory of the compilation (hence, the first directory in the directories field), 1 if it was found in the second directory in the directories field, and so on.

   This content code is always paired with one of DW_FORM_data1, DW_FORM_data2 or DW_FORM_udata.

   The optimal form for a producer to use (which results in the minimum size for the set of `include_index` fields) depends not only on the number of directories in the directories field, but potentially on the order in which those directories are listed and the number of times each is used in the `file_names` field.

3. DW_LNCT_timestamp
   DW_LNCT_timestamp indicates that the value is the implementation-defined time of last modification of the file, or 0 if not available. It is always paired with one of the forms DW_FORM_udata, DW_FORM_data4, DW_FORM_data8 or DW_FORM_block.

4. DW_LNCT_size
   DW_LNCT_size indicates that the value is the unsigned size of the file in bytes, or 0 if not available. It is paired with one of the forms DW_FORM_udata, DW_FORM_data1, DW_FORM_data2, DW_FORM_data4 or DW_FORM_data8.

5. DW_LNCT_MD5
   DW_LNCT_MD5 indicates that the value is a 16-byte MD5 digest of the file contents. It is paired with form DW_FORM_data16.

An example that uses this line number header format is found in Appendix D.5.1 on page 321.

6.2.4.2 Vendor-defined Content Descriptions

Vendor-defined content descriptions may be defined using content type codes in the range DW_LNCT_lo_user to DW_LNCT_hi_user. Each such code may be combined with one or more forms from the set: DW_FORM_block, DW_FORM_block1, DW_FORM_block2, DW_FORM_block4, DW_FORM_data1, DW_FORM_data2, DW_FORM_data4, DW_FORM_data8, DW_FORM_data16, DW_FORM_flag, DW_FORM_line_strp, DW_FORM_sdata, DW_FORM_sec_offset, DW_FORM_string, DW_FORM_strp, DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3, DW_FORM_strx4 and DW_FORM_udata.

If a consumer encounters a vendor-defined content type that it does not understand, it should skip the content data as though it were not present.

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 and operation pointers 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. Modify the operation pointer by incrementing the `address` and `op_index` registers as described below.
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."
7. Set the `discriminator` register to 0.

All of the special opcodes do those same seven things; they differ from one another only in what values they add to the `line`, `address` and `op_index` 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 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`, `address` and `op_index` 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 operation advance represents the number of operations to skip when advancing the operation pointer.

The special opcode is then calculated using the following formula:

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

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

When `maximum_operations_per_instruction` is 1, the operation advance is simply the address increment divided by the `minimum_instruction_length`.

To decode a special opcode, subtract the `opcode_base` from the opcode itself to give the *adjusted opcode*. The *operation advance* is the result of the adjusted opcode divided by the `line_range`. The new `address` and `op_index` values are given by

```
adjusted opcode = opcode - opcode_base
operation advance = adjusted opcode / line_range

new address = address +
  minimum_instruction_length *
    ((op_index + operation advance) / maximum_operations_per_instruction)

new op_index =
  (op_index + operation advance) % maximum_operations_per_instruction
```

When the `maximum_operations_per_instruction` field is 1, `op_index` is always 0 and these calculations simplify to those given for addresses in DWARF Version 3 and earlier.

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)
```

See Appendix D.5.2 on page 321 for an example.

6.2.5.2 Standard Opcodes

The standard opcodes, their 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 `discriminator` register to 0, and 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 as the operation advance and modifies the `address` and `op_index` registers as specified in Section 6.2.5.1 on page 160.

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 advances the `address` and `op_index` registers by the increments corresponding to special opcode 255.

   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 and sets the `op_index` register to 0. This is the only standard opcode whose operand is **not** a variable length number. It also does **not** multiply the operand by the `minimum_instruction_length` field of the header.

   Some assemblers may not be able 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

The extended opcodes are as follows:

1. **DW_LNE_end_sequence**
   The DW_LNE_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 on page 150). 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 of an address on the target machine. It sets the `address` register to the value given by the relocatable address and sets the `op_index` register to 0.

   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_set_discriminator**
   The DW_LNE_set_discriminator opcode takes a single parameter, an unsigned LEB128 integer. It sets the `discriminator` register to the new value.

The DW_LNE_define_file operation defined in earlier versions of DWARF is deprecated in DWARF Version 5.

Appendix D.5.3 on page 323 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 3.1.1 on page 60, the macro information for a given compilation unit is represented in the `.debug_macro` section of an object file.

The `.debug_macro` section is new in DWARF Version 5, and supersedes the `.debug_macinfo` section of earlier DWARF versions. While `.debug_macro` and `.debug_macinfo` sections cannot both occur in the same compilation unit, both may be found in the set of units that make up an executable or shared object file.

The representation of debugging information in the `.debug_macinfo` section is specified in earlier versions of the DWARF standard. Note that the `.debug_macinfo` section does not contain any headers and does not support sharing of strings or sharing of repeated macro sequences.

The macro information for each compilation unit consists of one or more macro units. Each macro unit starts with a header and is followed by a series of macro information entries or file inclusion entries. Each entry consists of an opcode followed by zero or more operands. Each macro unit ends with an entry containing an opcode of 0.

In all macro information entries, the line number of the entry is encoded as an unsigned LEB128 integer.

6.3.1 Macro Information Header

The macro information header contains the following fields:

1. `version` (uhalf)
   A version number (see Section 7.23 on page 237). This number is specific to the macro information and is independent of the DWARF version number.

2. `flags` (ubyte)
   The bits of the `flags` field are interpreted as a set of flags, some of which may indicate that additional fields follow.

   The following flags, beginning with the least significant bit, are defined:

   - `offset_size_flag`
     If the `offset_size_flag` is zero, the header is for a 32-bit DWARF format macro section and all offsets are 4 bytes long; if it is one, the header is for a 64-bit DWARF format macro section and all offsets are 8 bytes long.

   - `debug_line_offset_flag`
     If the `debug_line_offset_flag` is one, the `debug_line_offset` field (see below) is present. If zero, that field is omitted.

   - `opcode_operands_table_flag`
     If the `opcode_operands_table_flag` is one, the `opcode_operands_table` field (see below) is present. If zero, that field is omitted.

   All other flags are reserved by DWARF.

3. `debug_line_offset`
   An offset in the `.debug_line` section of the beginning of the line number information in the containing compilation, encoded as a 4-byte offset for a 32-bit DWARF format macro section and an 8-byte offset for a 64-bit DWARF format macro section.

4. `opcode_operands_table`
   An `opcode_operands_table` describing the operands of the macro information entry opcodes.

   The macro information entries defined in this standard may, but need not, be described in the table, while other user-defined entry opcodes used in the section are described there. Vendor extension entry opcodes are allocated in the range from DW_MACRO_lo_user to DW_MACRO_hi_user. Other unassigned codes are reserved for future DWARF standards.

The table starts with a 1-byte `count` of the defined opcodes, followed by an entry for each of those opcodes. Each entry starts with a 1-byte unsigned opcode number, followed by unsigned LEB128 encoded number of operands and for each operand there is a single unsigned byte describing the form in which the operand is encoded. The allowed forms are: DW_FORM_block, DW_FORM_block1, DW_FORM_block2, DW_FORM_block4, DW_FORM_data1, DW_FORM_data2, DW_FORM_data4, DW_FORM_data8, DW_FORM_data16, DW_FORM_flag, DW_FORM_line_strp, DW_FORM_sdata, DW_FORM_sec_offset, DW_FORM_string, DW_FORM_strp, DW_FORM_strp_sup, DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3, DW_FORM_strx4 and DW_FORM_udata.

6.3.2 Macro Information Entries

All macro information entries within a `.debug_macro` section for a given compilation unit appear in the same order in which the directives were processed by the compiler (after taking into account the effect of the macro import directives).

The source file in which a macro information entry occurs can be derived by interpreting the sequence of entries from the beginning of the `.debug_macro` *section. DW_MACRO_start_file and DW_MACRO_end_file indicate changes in the containing file.*

6.3.2.1 Define and Undefine Entries

The define and undefine macro entries have multiple forms that use different representations of their two operands.

While described in pairs below, the forms of define and undefine entries may be freely intermixed.

1. **DW_MACRO_define, DW_MACRO_undef**
   A DW_MACRO_define or DW_MACRO_undef entry has two operands. The first operand encodes the source line number of the `#define` or `#undef` macro directive. The second operand is a null-terminated character string for the macro being defined or undefined.

   The contents of the operands are described below (see Sections 6.3.2.2 and 6.3.2.3 following).

2. **DW_MACRO_define_strp, DW_MACRO_undef_strp**
   A DW_MACRO_define_strp or DW_MACRO_undef_strp entry has two operands. The first operand encodes the source line number of the `#define` or `#undef` macro directive. The second operand consists of an offset into a string table contained in the `.debug_str` section of the object file. The size of the operand is given in the header `offset_size_flag` field.

   The contents of the operands are described below (see Sections 6.3.2.2 and 6.3.2.3 following).

3. **DW_MACRO_define_strx, DW_MACRO_undef_strx**
   A DW_MACRO_define_strx or DW_MACRO_undef_strx entry has two operands. The first operand encodes the line number of the `#define` or `#undef` macro directive. The second operand identifies a string; it is represented using an unsigned LEB128 encoded value, which is interpreted as a zero-based index into an array of offsets in the `.debug_str_offsets` section.

   The contents of the operands are described below (see Sections 6.3.2.2 and 6.3.2.3 following).

4. **DW_MACRO_define_sup, DW_MACRO_undef_sup**
   A DW_MACRO_define_sup or DW_MACRO_undef_sup entry has two operands. The first operand encodes the line number of the `#define` or `#undef` macro directive. The second operand identifies a string; it is represented as an offset into a string table contained in the `.debug_str` section of the supplementary object file. The size of the operand depends on the macro section header `offset_size_flag` field.

   The contents of the operands are described below (see Sections 6.3.2.2 and 6.3.2.3 following).

6.3.2.2 Macro Define String

In the case of a DW_MACRO_define, DW_MACRO_define_strp, DW_MACRO_define_strx or DW_MACRO_define_sup entry, the value of the second operand is the name of the macro symbol that is 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 immediately by its definition string.

In the case of a function-like macro definition, no whitespace characters appear between the name of the defined macro and the following left parenthesis. Formal parameters are separated by a comma without any whitespace. Exactly one space character separates the right parenthesis that terminates the formal parameter list and the following definition string.

In the case of a "normal" (that is, non-function-like) macro definition, exactly one space character separates the name of the defined macro from the following definition text.

6.3.2.3 Macro Undefine String

In the case of a DW_MACRO_undef, DW_MACRO_undef_strp, DW_MACRO_undef_strx or DW_MACRO_undef_sup entry, the value of the second string is the name of the pre-processor symbol that is undefined at the indicated source line.

6.3.2.4 Entries for Command Line Options

A DWARF producer generates a define or undefine entry for each pre-processor symbol which is defined or undefined by some means other than such a directive within the compiled source text. In particular, pre-processor symbol definitions and undefinitions which occur as a result of command line options (when invoking the compiler) are represented by their own define and undefine entries.

All such define and undefine entries representing compilation options appear before the first DW_MACRO_start_file entry for that compilation unit (see Section 6.3.3 following) and encode the value 0 in their line number operands.

6.3.3 File Inclusion Entries

6.3.3.1 Source Include Directives

The following directives describe a source file inclusion directive (`#include` in C/C++) and the ending of an included file.

1. **DW_MACRO_start_file**
   A DW_MACRO_start_file entry has two operands. The first operand encodes the line number of the source line on which the `#include` macro directive occurs. The second operand encodes a source file name index.

   The source file name index is the file number in the line number information table for the compilation unit.

   If a DW_MACRO_start_file entry is present, the header contains a reference to the `.debug_line` section of the compilation.

2. **DW_MACRO_end_file**
   A DW_MACRO_end_file entry has no operands. The presence of the entry marks the end of the current source file inclusion.

When providing macro information in an object file, a producer generates DW_MACRO_start_file and DW_MACRO_end_file entries for the source file submitted to the compiler for compilation. This DW_MACRO_start_file entry has the value 0 in its line number operand and references the file entry in the line number information table for the primary source file.

6.3.3.2 Importation of Macro Units

The import entries make it possible to replicate macro units. The first form supports replication within the current compilation and the second form supports replication across separate executable or shared object files.

Import entries do not reflect the source program and, in fact, are not necessary at all. However, they do provide a mechanism that can be used to reduce redundancy in the macro information and thereby to save space.

1. **DW_MACRO_import**
   A DW_MACRO_import entry has one operand, an offset into another part of the `.debug_macro` section that is the beginning of a target macro unit. The size of the operand depends on the header `offset_size_flag` field. The DW_MACRO_import entry instructs the consumer to replicate the sequence of entries following the target macro header which begins at the given `.debug_macro` offset, up to, but excluding, the terminating entry with opcode `0`, as though it occurs in place of the import operation.

2. **DW_MACRO_import_sup**
   A DW_MACRO_import_sup entry has one operand, an offset from the start of the `.debug_macro` section in the supplementary object file. The size of the operand depends on the section header `offset_size_flag` field. Apart from the different location in which to find the macro unit, this entry type is equivalent to DW_MACRO_import.

   This entry type is aimed at sharing duplicate macro units between `.debug_macro` sections from different executable or shared object files.

   From within the `.debug_macro` section of the supplementary object file, DW_MACRO_define_strp and DW_MACRO_undef_strp entries refer to the `.debug_str` section of that same supplementary file; similarly, DW_MACRO_import entries refer to the `.debug_macro` section of that same supplementary file.

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 (for example, a breakpoint), or is a place where a subroutine made a call or was interrupted by an asynchronous event (for example, 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 virtually unwinds a stack in steps. Starting with the current activation it virtually 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 the actual state of the target process is unchanged.

The virtual unwind 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 subprograms save and restore registers during their lifetimes. This basis must be augmented on some machines with specific information that is defined by 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 object files, this is an object-relative offset.) The remaining columns contain virtual unwinding rules that are associated with the indicated location.

The CFA column defines the rule which computes the Canonical Frame Address 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 labelled 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 defined 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 recoverable 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.
val_offset(N)	The previous value of this register is the value 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 (see Section 2.5 on page 26).
val_expression(E)	The previous value of this register is the value produced by executing the DWARF expression E (see Section 2.5 on page 26).
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 a multiple of the address size relative to the start of the section 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 (see Section 7.2.2 on page 184). The size of the `length` field plus the value of `length` must be an integral multiple of the address size.

2. `CIE_id` (4 or 8 bytes, see Section 7.4 on page 196)
   A constant that is used to distinguish CIEs from FDEs.

3. `version` (ubyte)
   A version number (see Section 7.24 on page 238). This number is specific to the call frame information and is independent of the DWARF version number.

4. `augmentation` (sequence of UTF-8 characters)
   A null-terminated UTF-8 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.

   The augmentation string allows users to indicate that there is additional target-specific information in the CIE or FDE which is needed to virtually unwind a stack frame. For example, this might be information about dynamically allocated data which needs to be freed on exit from the routine.

   Because the `.debug_frame` section is useful independently of any `.debug_info` section, the augmentation string always uses UTF-8 encoding.

5. `address_size` (ubyte)
   The size of a target address in this CIE and any FDEs that use it, in bytes. If a compilation unit exists for this frame, its address size must match the address size here.

6. `segment_selector_size` (ubyte)
   The size of a segment selector in this CIE and any FDEs that use it, in bytes.

7. `code_alignment_factor` (unsigned LEB128)
   A constant that is factored out of all advance location instructions (see Section 6.4.2.1 on page 177). The resulting value is (*operand* * `code_alignment_factor`).

8. `data_alignment_factor` (signed LEB128)
   A constant that is factored out of certain offset instructions (see Sections 6.4.2.2 on page 177 and 6.4.2.3 on page 179). The resulting value is (*operand* * `data_alignment_factor`).

9. `return_address_register` (unsigned LEB128)
   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.

10. `initial_instructions` (array of ubyte)
    A sequence of rules that are interpreted to create the initial setting of each column in the table.

    The default rule for all columns before interpretation of the initial instructions is the undefined rule. However, an ABI authoring body or a compilation system authoring body may specify an alternate default value for any or all columns.

11. `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 (see Section 7.2.2 on page 184). The size of the `length` field plus the value of length must be an integral multiple of the address size.

2. `CIE_pointer` (4 or 8 bytes, see Section 7.4 on page 196)
   A constant offset into the `.debug_frame` section that denotes the CIE that is associated with this FDE.

3. `initial_location` (segment selector and target address)
   The address of the first location associated with this table entry. If the `segment_selector_size` field of this FDE's CIE is non-zero, the initial location is preceded by a segment selector of the given length.

4. `address_range` (target address)
   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 in Section 6.4.2.

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 7.24 on page 238). The instructions are defined in the following sections.

Some call frame instructions have operands that are encoded as DWARF expressions (see Section 2.5.1 on page 26). The following DWARF operators cannot be used in such operands:

- DW_OP_addrx, DW_OP_call2, DW_OP_call4, DW_OP_call_ref, DW_OP_const_type, DW_OP_constx, DW_OP_convert, DW_OP_deref_type, DW_OP_regval_type and DW_OP_reinterpret operators are not allowed in an operand of these instructions because the call frame information must not depend on other debug sections.
- DW_OP_push_object_address is not meaningful in an operand of these instructions because there is no object context to provide a value to push.
- DW_OP_call_frame_cfa is not meaningful in an operand of these instructions because its use would be circular.

Call frame instructions to which these restrictions apply include DW_CFA_def_cfa_expression, DW_CFA_expression and DW_CFA_val_expression.

6.4.2.1 Row Creation Instructions

1. **DW_CFA_set_loc**
   The DW_CFA_set_loc instruction takes a single operand that represents a target 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 is always greater than the current one. If the `segment_selector_size` field of this FDE's CIE is non-zero, the initial location is preceded by a segment selector of the given length.

2. **DW_CFA_advance_loc**
   The DW_CFA_advance_loc 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

3. **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.

4. **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.

5. **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.

6.4.2.2 CFA Definition Instructions

1. **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.

2. **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. The resulting offset is *factored_offset* * `data_alignment_factor`.

3. **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). This operation is valid only if the current CFA rule is defined to use a register and offset.

4. **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). This operation is valid only if the current CFA rule is defined to use a register and offset.

5. **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. The resulting offset is *factored_offset* * `data_alignment_factor`. This operation is valid only if the current CFA rule is defined to use a register and offset.

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

   See Section 6.4.2 on page 176 regarding restrictions on the DWARF expression operators that can be used.

6.4.2.3 Register Rule Instructions

1. **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."

2. **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."

3. **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 where the value of N is *factored offset* * `data_alignment_factor`.

4. **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.

5. **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 and factored. The resulting offset is *factored_offset* * `data_alignment_factor`.

6. **DW_CFA_val_offset**
   The DW_CFA_val_offset instruction takes two unsigned LEB128 operands representing a register number and a factored offset. The required action is to change the rule for the register indicated by the register number to be a val_offset(N) rule where the value of N is *factored_offset* * `data_alignment_factor`.

7. **DW_CFA_val_offset_sf**
   The DW_CFA_val_offset_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_val_offset except that the second operand is signed and factored. The resulting offset is *factored_offset* * `data_alignment_factor`.

8. **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 register(R) where R is the second register.

9. **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 change the rule for the register indicated by the register number to be an expression(E) rule where E is the DWARF expression. That is, the DWARF expression computes the address. The value of the CFA is pushed on the DWARF evaluation stack prior to execution of the DWARF expression.

   See Section 6.4.2 on page 176 regarding restrictions on the DWARF expression operators that can be used.

10. **DW_CFA_val_expression**
    The DW_CFA_val_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 change the rule for the register indicated by the register number to be a val_expression(E) rule where E is the DWARF expression. That is, the DWARF expression computes the value of the given register. The value of the CFA is pushed on the DWARF evaluation stack prior to execution of the DWARF expression.

    See Section 6.4.2 on page 176 regarding restrictions on the DWARF expression operators that can be used.

11. **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.

12. **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.

6.4.2.4 Row State Instructions

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.

1. **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.

2. **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.

6.4.2.5 Padding Instruction

1. **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.

6.4.3 Call Frame Instruction Usage

To determine the virtual unwind rule set for a given location (L1), search 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. Set L2 to the value of the `initial_location` field from the FDE header.*

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 is encountered, then compute a new location value (L2). If L1 $\geq$ 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. Note that the FDE is ill-formed if L2 is less than L1.*

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

For an example, see Appendix D.6 on page 325*.*

6.4.4 Call Frame Calling Address

When virtually unwinding frames, consumers frequently wish to obtain the address of the instruction which called a subroutine. This information is not always provided. Typically, however, one of the registers in the virtual unwind table is the Return Address.

If a Return Address register is defined in the virtual unwind table, and its rule is undefined (for example, by DW_CFA_undefined), then there is no return address and no call address, and the virtual unwind of stack activations is complete.

In most cases the return address is in the same context as the calling address, but that need not be the case, especially if the producer knows in some way the call never will return. The context of the 'return address' might be on a different line, in a different lexical block*, or past the end of the calling subroutine. If a consumer were to assume that it was in the same context as the calling address, the virtual unwind might fail.*

For architectures with constant-length instructions where the return address immediately follows the call instruction, a simple solution is to subtract the length of an instruction from the return address to obtain the calling instruction. For architectures with variable-length instructions (for example, x86), this is not possible. However, subtracting 1 from the return address, although not guaranteed to provide the exact calling address, generally will produce an address within the same context as the calling address, and that usually is sufficient.

Chapter 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_AT, DW_ATE, DW_CC, DW_CFA, DW_END, DW_IDX, DW_LANG, DW_LNCT, DW_LNE, DW_MACRO, DW_OP, DW_TAG, DW_UT) followed by _lo_user or _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.

For example, for debugging information entry tags, the special labels are DW_TAG_lo_user and DW_TAG_hi_user.

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 must be followed:

1. New attributes are 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 do not alter the semantics of previously existing attributes.
3. The semantics of any new tags do not conflict with the semantics of previously existing tags.
4. New forms of attribute value are not added.

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, because they do not represent valid encodings for the given type and do not appear in DWARF debugging information.

7.2.2 Initial Length Values

An initial length field is one of the fields that occur at the beginning of those DWARF sections that have a header (`.debug_aranges`, `.debug_info`, `.debug_line`, `.debug_loclists`, `.debug_names` and `.debug_rnglists`) 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 Version 2; such values must not be interpreted as a length field. The use of one such value, 0xffffffff, is defined in Section 7.4 on page 196); the use of the other values is reserved for possible future extensions.

7.3 Relocatable, Split, Executable, Shared, Package and Supplementary Object Files

7.3.1 Relocatable Object Files

A DWARF producer (for example, a compiler) typically generates its debugging information as part of a relocatable object file. Relocatable object files are then combined by a linker to form an executable file. During the linking process, the linker resolves (binds) symbolic references between the various object files, and relocates the contents of each object file into a combined virtual address space.

The DWARF debugging information is placed in several sections (see Appendix B on page 273), and requires an object file format capable of representing these separate sections. There are symbolic references between these sections, and also between the debugging information sections and the other sections that contain the text and data of the program itself. Many of these references require relocation, and the producer must emit the relocation information appropriate to the object file format and the target processor architecture. These references include the following:

- The compilation unit header (see Section 7.5.1 on page 199) in the `.debug_info` section contains a reference to the `.debug_abbrev` table. This reference requires a relocation so that after linking, it refers to that contribution to the combined `.debug_abbrev` section in the executable file.
- Debugging information entries may have attributes with the form DW_FORM_addr (see Section 7.5.4 on page 207). These attributes represent locations within the virtual address space of the program, and require relocation.
- A DWARF expression may contain a DW_OP_addr (see Section 2.5.1.1 on page 26) which contains a location within the virtual address space of the program, and require relocation.

- Debugging information entries may have attributes with the form DW_FORM_sec_offset (see Section 7.5.4 on page 207). These attributes refer to debugging information in other debugging information sections within the object file, and must be relocated during the linking process.
- Debugging information entries may have attributes with the form DW_FORM_ref_addr (see Section 7.5.4 on page 207). These attributes refer to debugging information entries that may be outside the current compilation unit. These values require both symbolic binding and relocation.
- Debugging information entries may have attributes with the form DW_FORM_strp (see Section 7.5.4 on page 207). These attributes refer to strings in the `.debug_str` section. These values require relocation.
- Entries in the `.debug_addr` and `.debug_aranges` sections may contain references to locations within the virtual address space of the program, and thus require relocation.
- Entries in the `.debug_loclists` and `.debug_rnglists` sections may contain references to locations within the virtual address space of the program depending on whether certain kinds of location or range list entries are used, and thus require relocation.
- In the `.debug_line` section, the operand of the DW_LNE_set_address opcode is a reference to a location within the virtual address space of the program, and requires relocation.
- The `.debug_str_offsets` section contains a list of string offsets, each of which is an offset of a string in the `.debug_str` section. Each of these offsets requires relocation. Depending on the implementation, these relocations may be implicit (that is, the producer may not need to emit any explicit relocation information for these offsets).
- The `debug_info_offset` field in the `.debug_aranges` header and the list of compilation units following the `.debug_names` header contain references to the `.debug_info` section. These references require relocation so that after linking they refer to the correct contribution in the combined `.debug_info` section in the executable file.
- Frame descriptor entries in the `.debug_frame` section (see Section 6.4.1 on page 172) contain an `initial_location` field value within the virtual address space of the program and require relocation.

Note that operands of classes constant and flag do not require relocation. Attribute operands that use forms DW_FORM_string, DW_FORM_ref1, DW_FORM_ref2, DW_FORM_ref4, DW_FORM_ref8, or DW_FORM_ref_udata also do not need relocation.

7.3.2 Split DWARF Object Files

A DWARF producer may partition the debugging information such that the majority of the debugging information can remain in individual object files without being processed by the linker.

This reduces link time by reducing the amount of information the linker must process.

7.3.2.1 First Partition (with Skeleton Unit)

The first partition contains debugging information that must still be processed by the linker, and includes the following:

- The line number tables, frame tables, and accelerated access tables, in the usual sections: `.debug_line`, `.debug_line_str`, `.debug_frame`, `.debug_names` and `.debug_aranges`, respectively.
- An address table, in the `.debug_addr` section. This table contains all addresses and constants that require link-time relocation, and items in the table can be referenced indirectly from the debugging information via the DW_FORM_addrx, DW_FORM_addrx1, DW_FORM_addrx2, DW_FORM_addrx3 and DW_FORM_addrx4 forms, by the DW_OP_addrx and DW_OP_constx operators, and by certain of the `DW_LLE_*` location list and `DW_RLE_*` range list entries.
- A skeleton compilation unit, as described in Section 3.1.2 on page 66, in the `.debug_info` section.
- An abbreviations table for the skeleton compilation unit, in the `.debug_abbrev` section used by the `.debug_info` section.
- A string table, in the `.debug_str` section. The string table is necessary only if the skeleton compilation unit uses one of the indirect string forms (DW_FORM_strp, DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 or DW_FORM_strx4).

- A string offsets table, in the `.debug_str_offsets` section for strings in the `.debug_str` section. The string offsets table is necessary only if the skeleton compilation unit uses one of the indexed string forms (DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3, DW_FORM_strx4).

The attributes contained in the skeleton compilation unit can be used by a DWARF consumer to find the DWARF object file that contains the second partition.

7.3.2.2 Second Partition (Unlinked or in a `.dwo` File)

The second partition contains the debugging information that does not need to be processed by the linker. These sections may be left in the object files and ignored by the linker (that is, not combined and copied to the executable object file), or they may be placed by the producer in a separate DWARF object file. This partition includes the following:

- The full compilation unit, in the `.debug_info.dwo` section.

  Attributes contained in the full compilation unit may refer to machine addresses indirectly using one of the DW_FORM_addrx, DW_FORM_addrx1, DW_FORM_addrx2, DW_FORM_addrx3 or DW_FORM_addrx4 forms, which access the table of addresses specified by the DW_AT_addr_base attribute in the associated skeleton unit. Location descriptions may similarly do so using the DW_OP_addrx and DW_OP_constx operations.

- Separate type units, in the `.debug_info.dwo` section.

- Abbreviations table(s) for the compilation unit and type units, in the `.debug_abbrev.dwo` section used by the `.debug_info.dwo` section.

- Location lists, in the `.debug_loclists.dwo` section.

- Range lists, in the `.debug_rnglists.dwo` section.

- A specialized line number table (for the type units), in the `.debug_line.dwo` section.

  This table contains only the directory and filename lists needed to interpret DW_AT_decl_file attributes in the debugging information entries.

- Macro information, in the `.debug_macro.dwo` section.

- A string table, in the `.debug_str.dwo` section.

- A string offsets table, in the `.debug_str_offsets.dwo` section for the strings in the `.debug_str.dwo` section.

Except where noted otherwise, all references in this document to a debugging information section (for example, `.debug_info`), apply also to the corresponding split DWARF section (for example, `.debug_info.dwo`).

Split DWARF object files do not get linked with any other files, therefore references between sections must not make use of normal object file relocation information. As a result, symbolic references within or between sections are not possible.

7.3.3 Executable Objects

The relocated addresses in the debugging information for an executable object are virtual addresses.

The sections containing the debugging information are typically not loaded as part of the memory image of the program (in ELF terminology, the sections are not "allocatable" and are not part of a loadable segment). Therefore, the debugging information sections described in this document are typically linked as if they were each to be loaded at virtual address 0, and references within the debugging information always implicitly indicate which section a particular offset refers to. (For example, a reference of form DW_FORM_sec_offset may refer to one of several sections, depending on the class allowed by a particular attribute of a debugging information entry, as shown in Table 7.5 on page 207.)

7.3.4 Shared Object Files

The relocated addresses in the debugging information for a shared object file are offsets relative to the start of the lowest region of memory loaded from that shared object file.

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

As with executable objects, the sections containing debugging information are typically not loaded as part of the memory image of the shared object, and are typically linked as if they were each to be loaded at virtual address 0.

7.3.5 DWARF Package Files

Using split DWARF object files allows the developer to compile, link, and debug an application quickly with less link-time overhead, but a more convenient format is needed for saving the debug information for later debugging of a deployed application. A DWARF package file can be used to collect the debugging information from the object (or separate DWARF object) files produced during the compilation of an application.

The package file is typically placed in the same directory as the application, and is given the same name with a "`.dwp`" extension.

A DWARF package file is itself an object file, using the same object file format (including byte order) as the corresponding application binary. It consists only of a file header, a section table, a number of DWARF debug information sections, and two index sections.

Each DWARF package file contains no more than one of each of the following sections, copied from a set of object or DWARF object files, and combined, section by section:

```
    .debug_info.dwo
    .debug_abbrev.dwo
    .debug_line.dwo
    .debug_loclists.dwo
    .debug_rnglists.dwo
    .debug_str_offsets.dwo
    .debug_str.dwo
    .debug_macro.dwo
```

The string table section in `.debug_str.dwo` contains all the strings referenced from DWARF attributes using any of the forms DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 or DW_FORM_strx4. Any attribute in a compilation unit or a type unit using this form refers to an entry in that unit's contribution to the `.debug_str_offsets.dwo` section, which in turn provides the offset of a string in the `.debug_str.dwo` section.

The DWARF package file also contains two index sections that provide a fast way to locate debug information by compilation unit ID for compilation units, or by type signature for type units:

```
    .debug_cu_index
    .debug_tu_index
```

7.3.5.1 The Compilation Unit (CU) Index Section

The `.debug_cu_index` section is a hashed lookup table that maps a compilation unit ID to a set of contributions in the various debug information sections. Each contribution is stored as an offset within its corresponding section and a size.

Each compilation unit set may contain contributions from the following sections:

```
    .debug_info.dwo (required)
    .debug_abbrev.dwo (required)
    .debug_line.dwo
    .debug_loclists.dwo
    .debug_rnglists.dwo
    .debug_str_offsets.dwo
    .debug_macro.dwo
```

Note that a compilation unit set is not able to represent `.debug_macinfo` information from DWARF Version 4 or earlier formats.

7.3.5.2 The Type Unit (TU) Index Section

The `.debug_tu_index` section is a hashed lookup table that maps a type signature to a set of offsets in the various debug information sections. Each contribution is stored as an offset within its corresponding section and a size.

Each type unit set may contain contributions from the following sections:

```
    .debug_info.dwo (required)
    .debug_abbrev.dwo (required)
    .debug_line.dwo
    .debug_str_offsets.dwo
```

7.3.5.3 Format of the CU and TU Index Sections

Both index sections have the same format, and serve to map an 8-byte signature to a set of contributions to the debug sections. Each index section begins with a header, followed by a hash table of signatures, a parallel table of indexes, a table of offsets, and a table of sizes. The index sections are aligned at 8-byte boundaries in the DWARF package file.

The index section header contains the following fields:

1. `version` (uhalf)
   A version number. This number is specific to the CU and TU index information and is independent of the DWARF version number.

   The version number is 5.

2. *padding* (uhalf)
   Reserved to DWARF (must be zero).

3. `section_count` (uword)
   The number of entries in the table of section counts that follows. For brevity, the contents of this field is referred to as N below.

4. `unit_count` (uword)
   The number of compilation units or type units in the index. For brevity, the contents of this field is referred to as U below.

5. `slot_count` (uword)
   The number of slots in the hash table. For brevity, the contents of this field is referred to as S below.

We assume that U and S do not exceed 2^{32}.

The size of the hash table, S, must be 2^k such that: $2^k \;>\; 3 * U/2$

The hash table begins at offset 16 in the section, and consists of an array of S 8-byte slots. Each slot contains a 64-bit signature.

The parallel table of indices begins immediately after the hash table (at offset $16 + 8 * S$ from the beginning of the section), and consists of an array of S 4-byte slots, corresponding 1-1 with slots in the hash table. Each entry in the parallel table contains a row index into the tables of offsets and sizes.

Unused slots in the hash table have 0 in both the hash table entry and the parallel table entry. While 0 is a valid hash value, the row index in a used slot will always be non-zero.

Given an 8-byte compilation unit ID or type signature X, an entry in the hash table is located as follows:

1. Define $REP(X)$ to be the value of X interpreted as an unsigned 64-bit integer in the target byte order.

2. Calculate a primary hash $H = REP(X)\ \&\ MASK(k)$, where $MASK(k)$ is a mask with the low-order k bits all set to 1.

3. Calculate a secondary hash $H' = (((REP(X) >> 32)\ \&\ MASK(k))\ |\ 1)$.

4. If the hash table entry at index H matches the signature, use that entry. If the hash table entry at index H is unused (all zeroes), terminate the search: the signature is not present in the table.
5. Let $H = (H + H')\ modulo\ S$. Repeat at Step 4.

Because $S > U$, and H' and S are relatively prime, the search is guaranteed to stop at an unused slot or find the match.

The table of offsets begins immediately following the parallel table (at offset $16 + 12 * S$ from the beginning of the section). This table consists of a single header row containing N fields, each a 4-byte unsigned integer, followed by U data rows, each also containing N fields of 4-byte unsigned integers. The fields in the header row provide a section identifier referring to a debug section; the available section identifiers are shown in Table 7.1 following. Each data row corresponds to a specific CU or TU in the package file. In the data rows, each field provides an offset to the debug section whose identifier appears in the corresponding field of the header row. The data rows are indexed starting at 1.

Not all sections listed in the table need be included.

Table 7.1: DWARF package file section identifier encodings

Section identifier	**Value**	**Section**
DW_SECT_INFO	1	`.debug_info.dwo`
Reserved	2	
DW_SECT_ABBREV	3	`.debug_abbrev.dwo`
DW_SECT_LINE	4	`.debug_line.dwo`
DW_SECT_LOCLISTS	5	`.debug_loclists.dwo`
DW_SECT_STR_OFFSETS	6	`.debug_str_offsets.dwo`
DW_SECT_MACRO	7	`.debug_macro.dwo`
DW_SECT_RNGLISTS	8	`.debug_rnglists.dwo`

The offsets provided by the CU and TU index sections are the base offsets for the contributions made by each CU or TU to the corresponding section in the package file. Each CU and TU header contains a `debug_abbrev_offset` field, used to find the abbreviations table for that CU or TU within the contribution to the `.debug_abbrev.dwo` section for that CU or TU, and are interpreted as relative to the base offset given in the index section. Likewise, offsets into `.debug_line.dwo` from DW_AT_stmt_list attributes are interpreted as relative to the base offset for `.debug_line.dwo`, and offsets into other debug sections obtained from DWARF attributes are also interpreted as relative to the corresponding base offset.

The table of sizes begins immediately following the table of offsets, and provides the sizes of the contributions made by each CU or TU to the corresponding section in the package file. This table consists of U data rows, each with N fields of 4-byte unsigned integers. Each data row corresponds to the same CU or TU as the corresponding data row in the table of offsets described above. Within each data row, the N fields also correspond one-to-one with the fields in the corresponding data row of the table of offsets. Each field provides the size of the contribution made by a CU or TU to the corresponding section in the package file.

For an example, see Figure F.10 on page 412.

7.3.6 DWARF Supplementary Object Files

A supplementary object file permits a post-link utility to analyze executable and shared object files and collect duplicate debugging information into a single file that can be referenced by each of the original files. This is in contrast to split DWARF object files, which allow the compiler to split the debugging information between multiple files in order to reduce link time and executable size.

A DWARF supplementary object file is itself an object file, using the same object file format, byte order, and size as the corresponding application executables or shared libraries. It consists only of a file header, section table, and a number of DWARF debug information sections. Both the supplementary object file and all the executable or shared object files that reference entries or strings in that file must contain a `.debug_sup` section that establishes the relationship.

The `.debug_sup` section contains:

1. `version` (uhalf)
   A 2-byte unsigned integer representing the version of the DWARF information for the compilation unit.

   The value in this field is 5.

2. `is_supplementary` (ubyte)
   A 1-byte unsigned integer, which contains the value 1 if it is in the supplementary object file that other executable or shared object files refer to, or 0 if it is an executable or shared object referring to a supplementary object file.

3. `sup_filename` (null terminated filename string)
   If `is_supplementary` is 0, this contains either an absolute filename for the supplementary object file, or a filename relative to the object file containing the `.debug_sup` section. If `is_supplementary` is 1, then `sup_filename` is not needed and must be an empty string (a single null byte).

4. `sup_checksum_len` (unsigned LEB128)
   Length of the following `sup_checksum` field; this value can be 0 if no checksum is provided.

5. `sup_checksum` (array of ubyte)
   An implementation-defined integer constant value that provides unique identification of the supplementary file.

Debug information entries that refer to an executable's or shared object's addresses must *not* be moved to supplementary files (the addesses will likely not be the same). Similarly, entries referenced from within location descriptions or using loclistsptr form attributes must not be moved to a supplementary object file.

Executable or shared object file compilation units can use DW_TAG_imported_unit with an DW_AT_import attribute that uses DW_FORM_ref_sup4 or DW_FORM_ref_sup8 to import entries from the supplementary object file, other DW_FORM_ref_sup4 or DW_FORM_ref_sup8 attributes to refer directly to individual entries in the supplementary file, and DW_FORM_strp_sup form attributes to refer to strings that are used by debug information of multiple executables or shared object files. Within the supplementary object file's debugging sections, forms DW_FORM_ref_sup4, DW_FORM_ref_sup8 or DW_FORM_strp_sup are not used, and all reference forms referring to some other sections refer to the local sections in the supplementary object file.

In macro information, DW_MACRO_define_sup or DW_MACRO_undef_sup opcodes can refer to strings in the `.debug_str` section of the supplementary object file, or DW_MACRO_import_sup can refer to `.debug_macro` section entries. Within the `.debug_macro` section of a supplementary object file, DW_MACRO_define_strp and DW_MACRO_undef_strp opcodes refer to the local `.debug_str` section in that supplementary file, not the one in the executable or shared object file.

7.4 32-Bit and 64-Bit DWARF Formats

There are two closely-related DWARF 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 four bytes. 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 eight bytes. 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.

Except where noted otherwise, all references in this document to a debugging information section (for example, `.debug_info`), apply also to the corresponding split DWARF section (for example, `.debug_info.dwo`).

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 on page 184) is an unsigned 4-byte integer (which must be less than `0xfffffff0`); in the 64-bit DWARF format, an initial length field is 12 bytes in size, and has two parts:

    - The first four bytes have the value `0xffffffff`.

    - The following eight bytes contain the actual length represented as an unsigned 8-byte 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 4-byte unsigned integer values; in the 64-bit DWARF format, they are 8-byte 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_names`	entry in array of CUs or local TUs	offset in `.debug_info`

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 4-byte unsigned integer; for the 64-bit DWARF format, the value is an 8-byte unsigned integer.

Form	Role
DW_FORM_line_strp	offset in `.debug_line_str`
DW_FORM_ref_addr	offset in `.debug_info`
DW_FORM_sec_offset	offset in a section other than `.debug_info` or `.debug_str`
DW_FORM_strp	offset in `.debug_str`
DW_FORM_strp_sup	offset in `.debug_str` section of a supplementary object file
DW_OP_call_ref	offset in `.debug_info`

4. Within the body of the `.debug_line` section, certain forms of content description depend on the choice of DWARF format as follows: for the 32-bit DWARF format, the value is a 4-byte unsigned integer; for the 64-bit DWARF format, the value is a 8-byte unsigned integer.

Form	Role
DW_FORM_line_strp	offset in `.debug_line_str`

5. Within the body of the `.debug_names` sections, the representation of each entry in the array of compilation units (CUs) and the array of local type units (TUs), which represents an offset in the `.debug_info` section, depends on the DWARF format as follows: in the 32-bit DWARF format, each entry is a 4-byte unsigned integer; in the 64-bit DWARF format, it is a 8-byte unsigned integer.

6. In the body of the `.debug_str_offsets` sections, the size of entries in the body depend on the DWARF format as follows: in the 32-bit DWARF format, entries are 4-byte unsigned integer values; in the 64-bit DWARF format, they are 8-byte unsigned integers.

7. In the body of the `.debug_loclists` and `.debug_rnglists` sections, the offsets the follow the header depend on the DWARF format as follows: in the 32-bit DWARF format, offsets are 4-byte unsigned integer values; in the 64-bit DWARF format, they are 8-byte unsigned integers.

The 32-bit and 64-bit DWARF format conventions must *not* 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 not *use the 64-bit format* by default. *In most cases, the division of even very large applications into a number of executable and shared object files 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-bit format allows the unusual case to be handled as well. Even in this case, it is expected that only application supplied objects will need to 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 producer, a contribution is made to the `.debug_info` section of the object file. Each such contribution consists of a compilation unit header (see Section 7.5.1.1 on page 200) followed by a single DW_TAG_compile_unit or DW_TAG_partial_unit debugging information entry, together with its children.

For each type defined in a compilation unit, a separate contribution may also be made to the `.debug_info` section of the object file. Each such contribution consists of a type unit header (see Section 7.5.1.3 on page 202) followed by a DW_TAG_type_unit entry, together with its children.

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.

Multiple debugging information entries may share the same abbreviation table entry. Each compilation unit is associated with a particular abbreviation table, but multiple compilation units may share the same table.

7.5.1 Unit Headers

Unit headers contain a field, `unit_type`, whose value indicates the kind of compilation unit (see Section 3.1) that follows. The encodings for the unit type enumeration are shown in Table 7.2.

Table 7.2: Unit header unit type encodings

Unit header unit type encodings	**Value**
DW_UT_compile ‡	0x01
DW_UT_type ‡	0x02
DW_UT_partial ‡	0x03
DW_UT_skeleton ‡	0x04
DW_UT_split_compile ‡	0x05
DW_UT_split_type ‡	0x06
DW_UT_lo_user ‡	0x80
DW_UT_hi_user ‡	0xff
‡ *New in DWARF Version 5*	

All unit headers have the same initial three fields: `initial_length`, `version` and `unit_type`.

7.5.1.1 Full and Partial Compilation Unit Headers

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 `0xfffffff0`); 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 on page 196).

2. `version` (uhalf)
   A 2-byte unsigned integer representing the version of the DWARF information for the compilation unit.

   The value in this field is 5.

   See also Appendix G on page 415 for a summary of all version numbers that apply to DWARF sections.

3. `unit_type` (ubyte)
   A 1-byte unsigned integer identifying this unit as a compilation unit. The value of this field is DW_UT_compile for a (non-split) full compilation unit or DW_UT_partial for a (non-split) partial compilation unit (see Section 3.1.1 on page 60).

   See Section 7.5.1.2 regarding a split full compilation unit.

   This field is new in DWARF Version 5.

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.

5. `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 on page 196).

7.5.1.2 Skeleton and Split Compilation Unit Headers

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 `0xfffffff0`); 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 on page 196).

2. `version` (uhalf)
   A 2-byte unsigned integer representing the version of the DWARF information for the compilation unit.

   The value in this field is 5.

   See also Appendix G on page 415 for a summary of all version numbers that apply to DWARF sections.

3. `unit_type` (ubyte)
   A 1-byte unsigned integer identifying this unit as a compilation unit. The value of this field is DW_UT_skeleton for a skeleton compilation unit or DW_UT_split_compile for a split (full) compilation unit (see Section 3.1.2 on page 66).

   There is no split analog to the partial compilation unit.

   This field is new in DWARF Version 5.

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.

5. `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 on page 196).

6. `dwo_id` (unit ID)
   An 8-byte implementation-defined integer constant value, known as the compilation unit ID, that provides unique identification of a skeleton compilation unit and its associated split compilation unit in the object file named in the DW_AT_dwo_name attribute of the skeleton compilation.

7.5.1.3 Type Unit Headers

The header for the series of debugging information entries contributing to the description of a type that has been placed in its own type unit, within the `.debug_info` section, 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 type 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 `0xfffffff0`); 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 on page 196).

2. `version` (uhalf)
   A 2-byte unsigned integer representing the version of the DWARF information for the type unit.

   The value in this field is 5.

3. `unit_type` (ubyte)
   A 1-byte unsigned integer identifying this unit as a type unit. The value of this field is DW_UT_type for a non-split type unit (see Section 3.1.4 on page 68) or DW_UT_split_type for a split type unit.

   This field is new in DWARF Version 5.

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.

5. `debug_abbrev_offset` (section offset)
   A 4-byte or 8-byte unsigned offset into the `.debug_abbrev` section. This offset associates the type 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 on page 196).

6. `type_signature` (8-byte unsigned integer)
   A unique 8-byte signature (see Section 7.32 on page 245) of the type described in this type unit.

   An attribute that refers (using DW_FORM_ref_sig8) to the primary type contained in this type unit uses this value.

7. `type_offset` (section offset)
   A 4-byte or 8-byte unsigned offset relative to the beginning of the type unit header. This offset refers to the debugging information entry that describes the type. Because the type may be nested inside a namespace or other structures, and may contain references to other types that have not been placed in separate type units, it is not necessarily either the first or the only entry in the type 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 on page 196).

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 Table 7.3 on the following page.

Table 7.3: Tag encodings

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
Reserved	0x06
Reserved	0x07
DW_TAG_imported_declaration	0x08
Reserved	0x09
DW_TAG_label	0x0a
DW_TAG_lexical_block	0x0b
Reserved	0x0c
DW_TAG_member	0x0d
Reserved	0x0e
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
Reserved	0x14
DW_TAG_subroutine_type	0x15
DW_TAG_typedef	0x16
DW_TAG_union_type	0x17
DW_TAG_unspecified_parameters	0x18
DW_TAG_variant	0x19
DW_TAG_common_block	0x1a
DW_TAG_common_inclusion	0x1b
DW_TAG_inheritance	0x1c
DW_TAG_inlined_subroutine	0x1d
DW_TAG_module	0x1e

Continued on next page

Tag name	Value
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
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

Continued on next page

Tag name	Value
Reserved	0x3e[1]
DW_TAG_condition	0x3f
DW_TAG_shared_type	0x40
DW_TAG_type_unit	0x41
DW_TAG_rvalue_reference_type	0x42
DW_TAG_template_alias	0x43
DW_TAG_coarray_type ‡	0x44
DW_TAG_generic_subrange ‡	0x45
DW_TAG_dynamic_type ‡	0x46
DW_TAG_atomic_type ‡	0x47
DW_TAG_call_site ‡	0x48
DW_TAG_call_site_parameter ‡	0x49
DW_TAG_skeleton_unit ‡	0x4a
DW_TAG_immutable_type ‡	0x4b
DW_TAG_lo_user	0x4080
DW_TAG_hi_user	0xffff
‡ *New in DWARF Version 5*	

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 Table 7.4 on the next page (As mentioned in Section 2.3 on page 24, each chain of sibling entries is terminated by a null entry.)

[1]Code 0x3e is reserved to allow backward compatible support of the DW_TAG_mutable_type DIE that was defined (only) in DWARF Version 3.

Table 7.4: Child determination encodings

Children determination name	Value
DW_CHILDREN_no	0x00
DW_CHILDREN_yes	0x01

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 attribute form DW_FORM_implicit_const is another special case. For attributes with this form, the attribute specification contains a third part, which is a signed LEB128 number. The value of this number is used as the value of the attribute, and no value is stored in the `.debug_info` section.

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

See Appendix D.1.1 on page 287 for a depiction of the organization of the debugging information.

7.5.4 Attribute Encodings

The encodings for the attribute names are given in Table 7.5 following.

Table 7.5: Attribute encodings

Attribute name	Value	Classes
DW_AT_sibling	0x01	reference
DW_AT_location	0x02	exprloc, loclist
DW_AT_name	0x03	string
Reserved	0x04	*not applicable*
Reserved	0x05	*not applicable*

Continued on next page

Attribute name	**Value**	**Classes**
Reserved	0x06	*not applicable*
Reserved	0x07	*not applicable*
Reserved	0x08	*not applicable*
DW_AT_ordering	0x09	constant
Reserved	0x0a	*not applicable*
DW_AT_byte_size	0x0b	constant, exprloc, reference
Reserved	0x0c[2]	constant, exprloc, reference
DW_AT_bit_size	0x0d	constant, exprloc, reference
Reserved	0x0e	*not applicable*
Reserved	0x0f	*not applicable*
DW_AT_stmt_list	0x10	lineptr
DW_AT_low_pc	0x11	address
DW_AT_high_pc	0x12	address, constant
DW_AT_language	0x13	constant
Reserved	0x14	*not applicable*
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	exprloc, loclist, reference
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
DW_AT_default_value	0x1e	constant, reference, flag
Reserved	0x1f	*not applicable*
DW_AT_inline	0x20	constant
DW_AT_is_optional	0x21	flag
DW_AT_lower_bound	0x22	constant, exprloc, reference
Reserved	0x23	*not applicable*

Continued on next page

[2]Code 0x0c is reserved to allow backward compatible support of the DW_AT_bit_offset attribute which was defined in DWARF Version 3 and earlier.

Attribute name	**Value**	**Classes**
Reserved	0x24	*not applicable*
DW_AT_producer	0x25	string
Reserved	0x26	*not applicable*
DW_AT_prototyped	0x27	flag
Reserved	0x28	*not applicable*
Reserved	0x29	*not applicable*
DW_AT_return_addr	0x2a	exprloc, loclist
Reserved	0x2b	*not applicable*
DW_AT_start_scope	0x2c	constant, rnglist
Reserved	0x2d	*not applicable*
DW_AT_bit_stride	0x2e	constant, exprloc, reference
DW_AT_upper_bound	0x2f	constant, exprloc, reference
Reserved	0x30	*not applicable*
DW_AT_abstract_origin	0x31	reference
DW_AT_accessibility	0x32	constant
DW_AT_address_class	0x33	constant
DW_AT_artificial	0x34	flag
DW_AT_base_types	0x35	reference
DW_AT_calling_convention	0x36	constant
DW_AT_count	0x37	constant, exprloc, reference
DW_AT_data_member_location	0x38	constant, exprloc, loclist
DW_AT_decl_column	0x39	constant
DW_AT_decl_file	0x3a	constant
DW_AT_decl_line	0x3b	constant
DW_AT_declaration	0x3c	flag
DW_AT_discr_list	0x3d	block
DW_AT_encoding	0x3e	constant
DW_AT_external	0x3f	flag
DW_AT_frame_base	0x40	exprloc, loclist
DW_AT_friend	0x41	reference
DW_AT_identifier_case	0x42	constant

Continued on next page

Attribute name	**Value**	**Classes**
Reserved	0x43[3]	macptr
DW_AT_namelist_item	0x44	reference
DW_AT_priority	0x45	reference
DW_AT_segment	0x46	exprloc, loclist
DW_AT_specification	0x47	reference
DW_AT_static_link	0x48	exprloc, loclist
DW_AT_type	0x49	reference
DW_AT_use_location	0x4a	exprloc, loclist
DW_AT_variable_parameter	0x4b	flag
DW_AT_virtuality	0x4c	constant
DW_AT_vtable_elem_location	0x4d	exprloc, loclist
DW_AT_allocated	0x4e	constant, exprloc, reference
DW_AT_associated	0x4f	constant, exprloc, reference
DW_AT_data_location	0x50	exprloc
DW_AT_byte_stride	0x51	constant, exprloc, reference
DW_AT_entry_pc	0x52	address, constant
DW_AT_use_UTF8	0x53	flag
DW_AT_extension	0x54	reference
DW_AT_ranges	0x55	rnglist
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_binary_scale	0x5b	constant
DW_AT_decimal_scale	0x5c	constant
DW_AT_small	0x5d	reference
DW_AT_decimal_sign	0x5e	constant
DW_AT_digit_count	0x5f	constant
DW_AT_picture_string	0x60	string

Continued on next page

[3] Code 0x43 is reserved to allow backward compatible support of the DW_AT_macro_info attribute which was defined in DWARF Version 4 and earlier.

Attribute name	**Value**	**Classes**
DW_AT_mutable	0x61	flag
DW_AT_threads_scaled	0x62	flag
DW_AT_explicit	0x63	flag
DW_AT_object_pointer	0x64	reference
DW_AT_endianity	0x65	constant
DW_AT_elemental	0x66	flag
DW_AT_pure	0x67	flag
DW_AT_recursive	0x68	flag
DW_AT_signature	0x69	reference
DW_AT_main_subprogram	0x6a	flag
DW_AT_data_bit_offset	0x6b	constant
DW_AT_const_expr	0x6c	flag
DW_AT_enum_class	0x6d	flag
DW_AT_linkage_name	0x6e	string
DW_AT_string_length_bit_size ‡	0x6f	constant
DW_AT_string_length_byte_size ‡	0x70	constant
DW_AT_rank ‡	0x71	constant, exprloc
DW_AT_str_offsets_base ‡	0x72	stroffsetsptr
DW_AT_addr_base ‡	0x73	addrptr
DW_AT_rnglists_base ‡	0x74	rnglistsptr
Reserved	0x75	*Unused*
DW_AT_dwo_name ‡	0x76	string
DW_AT_reference ‡	0x77	flag
DW_AT_rvalue_reference ‡	0x78	flag
DW_AT_macros ‡	0x79	macptr
DW_AT_call_all_calls ‡	0x7a	flag
DW_AT_call_all_source_calls ‡	0x7b	flag
DW_AT_call_all_tail_calls ‡	0x7c	flag
DW_AT_call_return_pc ‡	0x7d	address
DW_AT_call_value ‡	0x7e	exprloc
DW_AT_call_origin ‡	0x7f	exprloc
DW_AT_call_parameter ‡	0x80	reference

Continued on next page

Attribute name	Value	Classes
DW_AT_call_pc ‡	0x81	address
DW_AT_call_tail_call ‡	0x82	flag
DW_AT_call_target ‡	0x83	exprloc
DW_AT_call_target_clobbered ‡	0x84	exprloc
DW_AT_call_data_location ‡	0x85	exprloc
DW_AT_call_data_value ‡	0x86	exprloc
DW_AT_noreturn ‡	0x87	flag
DW_AT_alignment ‡	0x88	constant
DW_AT_export_symbols ‡	0x89	flag
DW_AT_deleted ‡	0x8a	flag
DW_AT_defaulted ‡	0x8b	constant
DW_AT_loclists_base ‡	0x8c	loclistsptr
DW_AT_lo_user	0x2000	—
DW_AT_hi_user	0x3fff	—

‡ *New in DWARF Version 5*

7.5.5 Classes and Forms

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. The attribute form governs how the value of an attribute is encoded. The classes and the forms they include are listed below.

Form DW_FORM_sec_offset is a member of more than one class, namely addrptr, lineptr, loclist, loclistsptr, macptr, rnglist, rnglistsptr, and stroffsetsptr; as a result, it is not possible for an attribute to allow more than one of these classes. The list of classes allowed by the applicable attribute in Table 7.5 on page 207 determines the class of the form.

In the form descriptions that follow, some forms are said to depend in part on the value of an attribute of the associated compilation unit:

- In the case of a split DWARF object file, the associated compilation unit is the skeleton compilation unit corresponding to the containing unit.
- Otherwise, the associated compilation unit is the containing unit.

Each possible form belongs to one or more of the following classes (see Table 2.3 on page 23 for a summary of the purpose and general usage of each class):

- address
  Represented as either:
  - 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 7.5.1.1 on page 200). This address is relocatable in a relocatable object file and is relocated in an executable file or shared object file.
  - An indirect index into a table of addresses (as described in the previous bullet) in the `.debug_addr` section (DW_FORM_addrx, DW_FORM_addrx1, DW_FORM_addrx2, DW_FORM_addrx3 and DW_FORM_addrx4). The representation of a DW_FORM_addrx value is an unsigned LEB128 value, which is interpreted as a zero-based index into an array of addresses in the `.debug_addr` section. The representation of a DW_FORM_addrx1, DW_FORM_addrx2, DW_FORM_addrx3 or DW_FORM_addrx4 value is a 1-, 2-, 3- or 4-byte unsigned integer value, respectively, which is similarly interpreted. The index is relative to the value of the DW_AT_addr_base attribute of the associated compilation unit.
- addrptr
  This is an offset into the `.debug_addr` section (DW_FORM_sec_offset). It consists of an offset from the beginning of the `.debug_addr` section to the beginning of the list of machine addresses information for the referencing entity. It is relocatable in a relocatable object file, and relocated in an executable or shared object file. 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 on page 196).

  This class is new in DWARF Version 5.
- 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 eight forms of constants. There are fixed length constant data forms for one-, two-, four-, eight- and sixteen-byte values (respectively, DW_FORM_data1, DW_FORM_data2, DW_FORM_data4, DW_FORM_data8 and DW_FORM_data16). There are variable length constant data forms encoded using signed LEB128 numbers (DW_FORM_sdata) and unsigned LEB128 numbers (DW_FORM_udata). There is also an implicit constant (DW_FORM_implicit_const), whose value is provided as part of the abbreviation declaration.

  The data in DW_FORM_data1, DW_FORM_data2, DW_FORM_data4, DW_FORM_data8 and DW_FORM_data16 can be anything. Depending on context, it may be a signed integer, an unsigned integer, a floating-point constant, or anything else. A consumer must use context to know how to interpret the bits, which if they are target machine data (such as an integer or floating-point constant) will be in target machine byte order.

  If one of the DW_FORM_data<n>forms is used to represent a signed or unsigned integer, it can be hard for a consumer to discover the context necessary to determine which interpretation is intended. Producers are therefore strongly encouraged to use DW_FORM_sdata or DW_FORM_udata for signed and unsigned integers respectively, rather than DW_FORM_data<n>.

- exprloc
  This is an unsigned LEB128 length followed by the number of information bytes specified by the length (DW_FORM_exprloc). The information bytes contain a DWARF expression (see Section 2.5 on page 26) or location description (see Section 2.6 on page 38).

- flag
  A flag is represented explicitly as a single byte of data (DW_FORM_flag) or implicitly (DW_FORM_flag_present). In the first case, 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. In the second case, the attribute is implicitly indicated as present, and no value is encoded in the debugging information entry itself.

- lineptr
  This is an offset into the `.debug_line` or `.debug_line.dwo` section (DW_FORM_sec_offset). It consists of an 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 file. 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 on page 196).

- loclist
  This is represented as either:

  - An index into the `.debug_loclists` section (DW_FORM_loclistx). The unsigned ULEB operand identifies an offset location relative to the base of that section (the location of the first offset in the section, not the first byte of the section). The contents of that location is then added to the base to determine the location of the target list of entries.

  - An offset into the `.debug_loclists` section (DW_FORM_sec_offset). The operand consists of a byte offset from the beginning of the `.debug_loclists` section. It is relocatable in a relocatable object file, and relocated in an executable or shared object file. 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 on page 196).

  This class is new in DWARF Version 5.

- loclistsptr
  This is an offset into the `.debug_loclists` section (DW_FORM_sec_offset). The operand consists of a byte offset from the beginning of the `.debug_loclists` section. It is relocatable in a relocatable object file, and relocated in an executable or shared object file. 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 on page 196).

This class is new in DWARF Version 5.

- macptr
  This is an offset into the `.debug_macro` or `.debug_macro.dwo` section (DW_FORM_sec_offset). It consists of an offset from the beginning of the `.debug_macro` or `.debug_macro.dwo` section to the the header 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 file. 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 on page 196).

- rnglist
  This is represented as either:

  - An index into the `.debug_rnglists` section (DW_FORM_rnglistx). The unsigned ULEB operand identifies an offset location relative to the base of that section (the location of the first offset in the section, not the first byte of the section). The contents of that location is then added to the base to determine the location of the target range list of entries.

  - An offset into the `.debug_rnglists` section (DW_FORM_sec_offset). The operand consists of a byte offset from the beginning of the `.debug_rnglists` section. It is relocatable in a relocatable object file, and relocated in an executable or shared object file. 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 on page 196).

  This class is new in DWARF Version 5.

- rnglistsptr
  This is an offset into the `.debug_rnglists` section (DW_FORM_sec_offset). It consists of a byte offset from the beginning of the `.debug_rnglists` section. It is relocatable in a relocatable object file, and relocated in an executable or shared object file. 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 on page 196).

  This class is new in DWARF Version 5.

- reference
  There are four 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 unsigned 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 `.debug_info` section; 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 file. 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 file, or, for references within a supplementary object file, an offset from the beginning of the local `.debug_info` section; it is relocatable in a relocatable object file and frequently relocated in an executable or shared object file. 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 on page 196).

    A debugging information entry that may be referenced by another compilation unit using DW_FORM_ref_addr must have a global symbolic name.

    For a reference from one executable or shared object file to another, the reference is resolved by the debugger to identify the executable or shared object file and the offset into that file'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 third type of reference can identify any debugging information type entry that has been placed in its own type unit. This type of reference (DW_FORM_ref_sig8) is the 8-byte type signature (see Section 7.32 on page 245) that was computed for the type.

  - The fourth type of reference is a reference from within the `.debug_info` section of the executable or shared object file to a debugging information entry in the `.debug_info` section of a supplementary object file. This type of reference (DW_FORM_ref_sup4 or DW_FORM_ref_sup8) is a 4- or 8-byte offset (respectively) from the beginning of the `.debug_info` section in the supplementary object file.

    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, third and fourth type of reference allows for the sharing of information, such as types, across compilation units, while the fourth type further allows for sharing of information across compilation units from different executables or shared object files.

    A reference to any kind of compilation unit identifies the debugging information entry for that unit, not the preceding header.

- 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),

  - As an offset into a string table contained in the `.debug_str` section of the object file (DW_FORM_strp), the `.debug_line_str` section of the object file (DW_FORM_line_strp), or as an offset into a string table contained in the `.debug_str` section of a supplementary object file (DW_FORM_strp_sup). DW_FORM_strp_sup offsets from the `.debug_info` section of a supplementary object file refer to the local `.debug_str` section of that same file. In the 32-bit DWARF format, the representation of a DW_FORM_strp, DW_FORM_line_strp or DW_FORM_strp_sup value is a 4-byte unsigned offset; in the 64-bit DWARF format, it is an 8-byte unsigned offset (see Section 7.4 on page 196).

  - As an indirect offset into the string table using an index into a table of offsets contained in the `.debug_str_offsets` section of the object file (DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 and DW_FORM_strx4). The representation of a DW_FORM_strx value is an unsigned LEB128 value, which is interpreted as a zero-based index into an array of offsets in the `.debug_str_offsets` section. The representation of a DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 or

DW_FORM_strx4 value is a 1-, 2-, 3- or 4-byte unsigned integer value, respectively, which is similarly interpreted. The offset entries in the `.debug_str_offsets` section have the same representation as DW_FORM_strp values.

Any combination of these three forms may be used within a single compilation.

If the DW_AT_use_UTF8 attribute is specified for the compilation, partial, skeleton or type 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 Version 3 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.

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

- stroffsetsptr
  This is an offset into the `.debug_str_offsets` section (DW_FORM_sec_offset). It consists of an offset from the beginning of the `.debug_str_offsets` section to the beginning of the string offsets information for the referencing entity. It is relocatable in a relocatable object file, and relocated in an executable or shared object file. 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 on page 196).

  This class is new in DWARF Version 5.

In no case does an attribute use one of the classes addrptr, lineptr, loclistsptr, macptr, rnglistsptr or stroffsetsptr to point into either the `.debug_info` or `.debug_str` section.

7.5.6 Form Encodings

The form encodings are listed in Table 7.6 following.

Table 7.6: Attribute form encodings

Form name	Value	Classes
DW_FORM_addr	0x01	address
Reserved	0x02	
DW_FORM_block2	0x03	block
DW_FORM_block4	0x04	block
DW_FORM_data2	0x05	constant
DW_FORM_data4	0x06	constant
DW_FORM_data8	0x07	constant
DW_FORM_string	0x08	string
DW_FORM_block	0x09	block
DW_FORM_block1	0x0a	block
DW_FORM_data1	0x0b	constant
DW_FORM_flag	0x0c	flag
DW_FORM_sdata	0x0d	constant
DW_FORM_strp	0x0e	string
DW_FORM_udata	0x0f	constant
DW_FORM_ref_addr	0x10	reference
DW_FORM_ref1	0x11	reference
DW_FORM_ref2	0x12	reference
DW_FORM_ref4	0x13	reference
DW_FORM_ref8	0x14	reference
DW_FORM_ref_udata	0x15	reference
DW_FORM_indirect	0x16	(see Section 7.5.3 on page 203)
DW_FORM_sec_offset	0x17	addrptr, lineptr, loclist, loclistsptr, macptr, rnglist, rnglistsptr, stroffsetsptr
DW_FORM_exprloc	0x18	exprloc
DW_FORM_flag_present	0x19	flag
DW_FORM_strx ‡	0x1a	string
DW_FORM_addrx ‡	0x1b	address
DW_FORM_ref_sup4 ‡	0x1c	reference
DW_FORM_strp_sup ‡	0x1d	string

Continued on next page

Form name	Value	Classes
DW_FORM_data16 ‡	0x1e	constant
DW_FORM_line_strp ‡	0x1f	string
DW_FORM_ref_sig8	0x20	reference
DW_FORM_implicit_const ‡	0x21	constant
DW_FORM_loclistx ‡	0x22	loclist
DW_FORM_rnglistx ‡	0x23	rnglist
DW_FORM_ref_sup8 ‡	0x24	reference
DW_FORM_strx1 ‡	0x25	string
DW_FORM_strx2 ‡	0x26	string
DW_FORM_strx3 ‡	0x27	string
DW_FORM_strx4 ‡	0x28	string
DW_FORM_addrx1 ‡	0x29	address
DW_FORM_addrx2 ‡	0x2a	address
DW_FORM_addrx3 ‡	0x2b	address
DW_FORM_addrx4 ‡	0x2c	address
‡ *New in DWARF Version 5*		

7.6 Variable Length Data

Integers may be 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 byte 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.

Unsigned LEB128 (ULEB128) 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.

Table 7.7 gives some examples of unsigned LEB128 numbers. The 0x80 in each case is the high order bit of the byte, indicating that an additional byte follows.

The encoding for signed, two's complement LEB128 (SLEB128) 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 4-byte 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. Table 7.7 gives some examples of unsigned LEB128 numbers and Table 7.8 gives some examples of signed LEB128 numbers.

Appendix C on page 283 gives algorithms for encoding and decoding these forms.

Table 7.7: Examples of unsigned LEB128 encodings

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

Table 7.8: Examples of signed 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

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 sequence of operations. Each operation is a 1-byte code that identifies that operation, followed by zero or more bytes of additional data. The encodings for the operations are described in Table 7.9.

Table 7.9: DWARF operation encodings

Operation	**Code**	**No. of Operands**	**Notes**
Reserved	0x01	-	
Reserved	0x02	-	
DW_OP_addr	0x03	1	constant address (size is target specific)
Reserved	0x04	-	
Reserved	0x05	-	
DW_OP_deref	0x06	0	
Reserved	0x07	-	
DW_OP_const1u	0x08	1	1-byte constant
DW_OP_const1s	0x09	1	1-byte constant
DW_OP_const2u	0x0a	1	2-byte constant
DW_OP_const2s	0x0b	1	2-byte constant
DW_OP_const4u	0x0c	1	4-byte constant
DW_OP_const4s	0x0d	1	4-byte constant
DW_OP_const8u	0x0e	1	8-byte constant
DW_OP_const8s	0x0f	1	8-byte constant
DW_OP_constu	0x10	1	ULEB128 constant
DW_OP_consts	0x11	1	SLEB128 constant
DW_OP_dup	0x12	0	
DW_OP_drop	0x13	0	
DW_OP_over	0x14	0	
DW_OP_pick	0x15	1	1-byte stack index
DW_OP_swap	0x16	0	

Continued on next page

Operation	Code	No. of Operands	Notes
DW_OP_rot	0x17	0	
DW_OP_xderef	0x18	0	
DW_OP_abs	0x19	0	
DW_OP_and	0x1a	0	
DW_OP_div	0x1b	0	
DW_OP_minus	0x1c	0	
DW_OP_mod	0x1d	0	
DW_OP_mul	0x1e	0	
DW_OP_neg	0x1f	0	
DW_OP_not	0x20	0	
DW_OP_or	0x21	0	
DW_OP_plus	0x22	0	
DW_OP_plus_uconst	0x23	1	ULEB128 addend
DW_OP_shl	0x24	0	
DW_OP_shr	0x25	0	
DW_OP_shra	0x26	0	
DW_OP_xor	0x27	0	
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_skip	0x2f	1	signed 2-byte constant
DW_OP_lit0	0x30	0	
DW_OP_lit1	0x31	0	literals 0 .. 31 =
...			(DW_OP_lit0 + literal)
DW_OP_lit31	0x4f	0	

Continued on next page

Operation	Code	No. of Operands	Notes
DW_OP_reg0	0x50	0	
DW_OP_reg1	0x51	0	reg 0 .. 31 =
. . .			(DW_OP_reg0 + regnum)
DW_OP_reg31	0x6f	0	
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, SLEB128 offset
DW_OP_piece	0x93	1	ULEB128 size of piece
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_form_tls_address	0x9b	0	
DW_OP_call_frame_cfa	0x9c	0	
DW_OP_bit_piece	0x9d	2	ULEB128 size, ULEB128 offset
DW_OP_implicit_value	0x9e	2	ULEB128 size, block of that size
DW_OP_stack_value	0x9f	0	
DW_OP_implicit_pointer ‡	0xa0	2	4- or 8-byte offset of DIE, SLEB128 constant offset
DW_OP_addrx ‡	0xa1	1	ULEB128 indirect address
DW_OP_constx ‡	0xa2	1	ULEB128 indirect constant

Continued on next page

Operation	Code	No. of Operands	Notes
DW_OP_entry_value ‡	0xa3	2	ULEB128 size, block of that size
DW_OP_const_type ‡	0xa4	3	ULEB128 type entry offset, 1-byte size, constant value
DW_OP_regval_type ‡	0xa5	2	ULEB128 register number, ULEB128 constant offset
DW_OP_deref_type ‡	0xa6	2	1-byte size, ULEB128 type entry offset
DW_OP_xderef_type ‡	0xa7	2	1-byte size, ULEB128 type entry offset
DW_OP_convert ‡	0xa8	1	ULEB128 type entry offset
DW_OP_reinterpret ‡	0xa9	1	ULEB128 type entry offset
DW_OP_lo_user	0xe0		
DW_OP_hi_user	0xff		

‡ *New in DWARF Version 5*

7.7.2 Location Descriptions

A location description is used to compute the location 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 entry, a default location entry or an end-of-list entry.

Each entry begins with an unsigned 1-byte code that indicates the kind of entry that follows. The encodings for these constants are given in Table 7.10.

Table 7.10: Location list entry encoding values

Location list entry encoding name	**Value**
DW_LLE_end_of_list ‡	0x00
DW_LLE_base_addressx ‡	0x01
DW_LLE_startx_endx ‡	0x02
DW_LLE_startx_length ‡	0x03
DW_LLE_offset_pair ‡	0x04
DW_LLE_default_location ‡	0x05
DW_LLE_base_address ‡	0x06
DW_LLE_start_end ‡	0x07
DW_LLE_start_length ‡	0x08
‡New in DWARF Version 5	

7.8 Base Type Attribute Encodings

The encodings of the constants used in the DW_AT_encoding attribute are given in Table 7.11

Table 7.11: Base type encoding values

Base type encoding name	**Value**
DW_ATE_address	0x01
DW_ATE_boolean	0x02
DW_ATE_complex_float	0x03
DW_ATE_float	0x04
DW_ATE_signed	0x05
DW_ATE_signed_char	0x06
DW_ATE_unsigned	0x07
DW_ATE_unsigned_char	0x08
DW_ATE_imaginary_float	0x09
DW_ATE_packed_decimal	0x0a
DW_ATE_numeric_string	0x0b
DW_ATE_edited	0x0c
DW_ATE_signed_fixed	0x0d

Continued on next page

Base type encoding name	Value
DW_ATE_unsigned_fixed	0x0e
DW_ATE_decimal_float	0x0f
DW_ATE_UTF	0x10
DW_ATE_UCS ‡	0x11
DW_ATE_ASCII ‡	0x12
DW_ATE_lo_user	0x80
DW_ATE_hi_user	0xff
‡ *New in DWARF Version 5*	

The encodings of the constants used in the DW_AT_decimal_sign attribute are given in Table 7.12.

Table 7.12: Decimal sign encodings

Decimal sign code name	Value
DW_DS_unsigned	0x01
DW_DS_leading_overpunch	0x02
DW_DS_trailing_overpunch	0x03
DW_DS_leading_separate	0x04
DW_DS_trailing_separate	0x05

The encodings of the constants used in the DW_AT_endianity attribute are given in Table 7.13.

Table 7.13: Endianity encodings

Endian code name	Value
DW_END_default	0x00
DW_END_big	0x01
DW_END_little	0x02
DW_END_lo_user	0x40
DW_END_hi_user	0xff

7.9 Accessibility Codes

The encodings of the constants used in the DW_AT_accessibility attribute are given in Table 7.14.

Table 7.14: Accessibility encodings

Accessibility code name	Value
DW_ACCESS_public	0x01
DW_ACCESS_protected	0x02
DW_ACCESS_private	0x03

7.10 Visibility Codes

The encodings of the constants used in the DW_AT_visibility attribute are given in Table 7.15.

Table 7.15: Visibility encodings

Visibility code name	Value
DW_VIS_local	0x01
DW_VIS_exported	0x02
DW_VIS_qualified	0x03

7.11 Virtuality Codes

The encodings of the constants used in the DW_AT_virtuality attribute are given in Table 7.16.

Table 7.16: Virtuality encodings

Virtuality code name	Value
DW_VIRTUALITY_none	0x00
DW_VIRTUALITY_virtual	0x01
DW_VIRTUALITY_pure_virtual	0x02

The value DW_VIRTUALITY_none is equivalent to the absence of the DW_AT_virtuality attribute.

7.12 Source Languages

The encodings of the constants used in the DW_AT_language attribute are given in Table 7.17. Names marked with † and their associated values are reserved, but the languages they represent are not well supported. Table 7.17 also shows the default lower bound, if any, assumed for an omitted DW_AT_lower_bound attribute in the context of a DW_TAG_subrange_type debugging information entry for each defined language.

Table 7.17: Language encodings

Language name	**Value**	**Default Lower Bound**
DW_LANG_C89	0x0001	0
DW_LANG_C	0x0002	0
DW_LANG_Ada83 †	0x0003	1
DW_LANG_C_plus_plus	0x0004	0
DW_LANG_Cobol74 †	0x0005	1
DW_LANG_Cobol85 †	0x0006	1
DW_LANG_Fortran77	0x0007	1
DW_LANG_Fortran90	0x0008	1
DW_LANG_Pascal83	0x0009	1
DW_LANG_Modula2	0x000a	1
DW_LANG_Java	0x000b	0
DW_LANG_C99	0x000c	0
DW_LANG_Ada95 †	0x000d	1
DW_LANG_Fortran95	0x000e	1
DW_LANG_PLI †	0x000f	1
DW_LANG_ObjC	0x0010	0
DW_LANG_ObjC_plus_plus	0x0011	0
DW_LANG_UPC	0x0012	0
DW_LANG_D	0x0013	0
DW_LANG_Python †	0x0014	0
DW_LANG_OpenCL †‡	0x0015	0

Continued on next page

Language name	Value	Default Lower Bound
DW_LANG_Go †‡	0x0016	0
DW_LANG_Modula3 †‡	0x0017	1
DW_LANG_Haskell †‡	0x0018	0
DW_LANG_C_plus_plus_03 ‡	0x0019	0
DW_LANG_C_plus_plus_11 ‡	0x001a	0
DW_LANG_OCaml ‡	0x001b	0
DW_LANG_Rust ‡	0x001c	0
DW_LANG_C11 ‡	0x001d	0
DW_LANG_Swift ‡	0x001e	0
DW_LANG_Julia ‡	0x001f	1
DW_LANG_Dylan ‡	0x0020	0
DW_LANG_C_plus_plus_14 ‡	0x0021	0
DW_LANG_Fortran03 ‡	0x0022	1
DW_LANG_Fortran08 ‡	0x0023	1
DW_LANG_RenderScript ‡	0x0024	0
DW_LANG_BLISS ‡	0x0025	0
DW_LANG_lo_user	0x8000	
DW_LANG_hi_user	0xffff	

† *See text*

‡ *New in DWARF Version 5*

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 Table 7.18.

Table 7.18: Identifier case encodings

Identifier case name	Value
DW_ID_case_sensitive	0x00
DW_ID_up_case	0x01
DW_ID_down_case	0x02
DW_ID_case_insensitive	0x03

7.15 Calling Convention Encodings

The encodings of the constants used in the DW_AT_calling_convention attribute are given in Table 7.19.

Table 7.19: Calling convention encodings

Calling convention name	Value
DW_CC_normal	0x01
DW_CC_program	0x02
DW_CC_nocall	0x03
DW_CC_pass_by_reference ‡	0x04
DW_CC_pass_by_value ‡	0x05
DW_CC_lo_user	0x40
DW_CC_hi_user	0xff
‡ *New in DWARF Version 5*	

7.16 Inline Codes

The encodings of the constants used in the DW_AT_inline attribute are given in Table 7.20.

Table 7.20: Inline encodings

Inline code name	Value
DW_INL_not_inlined	0x00
DW_INL_inlined	0x01
DW_INL_declared_not_inlined	0x02
DW_INL_declared_inlined	0x03

7.17 Array Ordering

The encodings of the constants used in the DW_AT_ordering attribute are given in Table 7.21.

Table 7.21: Ordering encodings

Ordering name	Value
DW_ORD_row_major	0x00
DW_ORD_col_major	0x01

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 Table 7.22.

Table 7.22: Discriminant descriptor encodings

Descriptor name	Value
DW_DSC_label	0x00
DW_DSC_range	0x01

7.19 Name Index Table

The version number in the name index table header is 5.

The name index attributes and their encodings are listed in Table 7.23.

Table 7.23: Name index attribute encodings

Attribute name	Value	Form/Class
DW_IDX_compile_unit ‡	1	constant
DW_IDX_type_unit ‡	2	constant
DW_IDX_die_offset ‡	3	reference
DW_IDX_parent ‡	4	constant
DW_IDX_type_hash ‡	5	DW_FORM_data8
DW_IDX_lo_user ‡	0x2000	
DW_IDX_hi_user ‡	0x3fff	
‡ *New in DWARF Version 5*		

The abbreviations table ends with an entry consisting of a single 0 byte for the abbreviation code. The size of the table given by `abbrev_table_size` may include optional padding following the terminating 0 byte.

7.20 Defaulted Member Encodings

The encodings of the constants used in the DW_AT_defaulted attribute are given in Table 7.24 following.

Table 7.24: Defaulted attribute encodings

Defaulted name	Value
DW_DEFAULTED_no ‡	0x00
DW_DEFAULTED_in_class ‡	0x01
DW_DEFAULTED_out_of_class ‡	0x02
‡ *New in DWARF Version 5*	

7.21 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. `unit_length` (initial length)
   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 `0xfffffff0`); 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 on page 196).

2. version (uhalf)
   A 2-byte version identifier representing the version of the DWARF information for the address range table.

   This value in this field is 2.

3. debug_info_offset (section offset)
   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 on page 196).

4. `address_size` (ubyte)
   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. `segment_selector_size` (ubyte)
   A 1-byte unsigned integer containing the size in bytes of a segment selector on the target system.

This header is followed by a series of tuples. Each tuple consists of a segment, an address and a length. The segment selector size is given by the `segment_selector_size` field of the header; the address and length size are each given by the `address_size` field of the header. 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, the size of a segment selector plus twice the size of an address). The header is padded, if necessary, to that boundary. Each set of tuples is terminated by a 0 for the segment, a 0 for the address and 0 for the length. If the `segment_selector_size` field in the header is zero, the segment selectors are omitted from all tuples, including the terminating tuple.

7.22 Line Number Information

The version number in the line number program header is 5.

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 standard opcodes are given in Table 7.25.

Table 7.25: Line number standard opcode encodings

Opcode name	**Value**
DW_LNS_copy	0x01
DW_LNS_advance_pc	0x02
DW_LNS_advance_line	0x03
DW_LNS_set_file	0x04
DW_LNS_set_column	0x05
DW_LNS_negate_stmt	0x06
DW_LNS_set_basic_block	0x07
DW_LNS_const_add_pc	0x08
DW_LNS_fixed_advance_pc	0x09
DW_LNS_set_prologue_end	0x0a
DW_LNS_set_epilogue_begin	0x0b
DW_LNS_set_isa	0x0c

The encodings for the extended opcodes are given in Table 7.26.

Table 7.26: Line number extended opcode encodings

Opcode name	Value
DW_LNE_end_sequence	0x01
DW_LNE_set_address	0x02
Reserved	0x03[4]
DW_LNE_set_discriminator	0x04
DW_LNE_lo_user	0x80
DW_LNE_hi_user	0xff

The encodings for the line number header entry formats are given in Table 7.27.

Table 7.27: Line number header entry format encodings

Line number header entry format name	Value
DW_LNCT_path ‡	0x1
DW_LNCT_directory_index ‡	0x2
DW_LNCT_timestamp ‡	0x3
DW_LNCT_size ‡	0x4
DW_LNCT_MD5 ‡	0x5
DW_LNCT_lo_user ‡	0x2000
DW_LNCT_hi_user ‡	0x3fff
‡ *New in DWARF Version 5*	

7.23 Macro Information

The version number in the macro information header is 5.

The source line numbers and source file indices encoded in the macro information section are represented as unsigned LEB128 numbers.

[4] Code 0x03 is reserved to allow backward compatible support of the DW_LNE_define_file operation which was defined in DWARF Version 4 and earlier.

The macro information entry type is encoded as a single unsigned byte. The encodings are given in Table 7.28.

Table 7.28: Macro information entry type encodings

Macro information entry type name	**Value**
DW_MACRO_define ‡	0x01
DW_MACRO_undef ‡	0x02
DW_MACRO_start_file ‡	0x03
DW_MACRO_end_file ‡	0x04
DW_MACRO_define_strp ‡	0x05
DW_MACRO_undef_strp ‡	0x06
DW_MACRO_import ‡	0x07
DW_MACRO_define_sup ‡	0x08
DW_MACRO_undef_sup ‡	0x09
DW_MACRO_import_sup ‡	0x0a
DW_MACRO_define_strx ‡	0x0b
DW_MACRO_undef_strx ‡	0x0c
DW_MACRO_lo_user ‡	0xe0
DW_MACRO_hi_user ‡	0xff
‡ *New in DWARF Version 5*	

7.24 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 4.

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 $\gg$ 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 Table 7.29 on the following page.

Table 7.29: Call frame instruction encodings

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 offset
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	
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_val_offset	0	0x14	ULEB128	ULEB128
DW_CFA_val_offset_sf	0	0x15	ULEB128	SLEB128
DW_CFA_val_expression	0	0x16	ULEB128	BLOCK
DW_CFA_lo_user	0	0x1c		
DW_CFA_hi_user	0	0x3f		

7.25 Range List Entries for Non-contiguous Address Ranges

Each entry in a range list (see Section 2.17.3 on page 52) is either a range list entry, a base address selection entry, or an end-of-list entry.

Each entry begins with an unsigned 1-byte code that indicates the kind of entry that follows. The encodings for these constants are given in Table 7.30.

Table 7.30: Range list entry encoding values

Range list entry encoding name	**Value**
DW_RLE_end_of_list ‡	0x00
DW_RLE_base_addressx ‡	0x01
DW_RLE_startx_endx ‡	0x02
DW_RLE_startx_length ‡	0x03
DW_RLE_offset_pair ‡	0x04
DW_RLE_base_address ‡	0x05
DW_RLE_start_end ‡	0x06
DW_RLE_start_length ‡	0x07
‡New in DWARF Version 5	

For a range list to be specified, the base address of the corresponding compilation unit must be defined (see Section 3.1.1 on page 60).

7.26 String Offsets Table

Each set of entries in the string offsets table contained in the `.debug_str_offsets` or `.debug_str_offsets.dwo` section begins with a header containing:

1. `unit_length` (initial length)
   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 `0xfffffff0`); 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 on page 196).
2. `version` (uhalf)
   A 2-byte version identifier containing the value 5.

3. *padding* (uhalf)
   Reserved to DWARF (must be zero).

This header is followed by a series of string table offsets that have the same representation as DW_FORM_strp. For the 32-bit DWARF format, each offset is 4 bytes long; for the 64-bit DWARF format, each offset is 8 bytes long.

The DW_AT_str_offsets_base attribute points to the first entry following the header. The entries are indexed sequentially from this base entry, starting from 0.

7.27 Address Table

Each set of entries in the address table contained in the `.debug_addr` section begins with a header containing:

1. `unit_length` (initial length)
   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 `0xfffffff0`); 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 on page 196).

2. `version` (uhalf)
   A 2-byte version identifier containing the value 5.

3. `address_size` (ubyte)
   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.

4. `segment_selector_size` (ubyte)
   A 1-byte unsigned integer containing the size in bytes of a segment selector on the target system.

This header is followed by a series of segment/address pairs. The segment size is given by the `segment_selector_size` field of the header, and the address size is given by the `address_size` field of the header. If the `segment_selector_size` field in the header is zero, the entries consist only of an addresses.

The DW_AT_addr_base attribute points to the first entry following the header. The entries are indexed sequentially from this base entry, starting from 0.

7.28 Range List Table

Each `.debug_rnglists` and `.debug_rnglists.dwo` section begins with a header containing:

1. `unit_length` (initial length)
   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 `0xfffffff0`); 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 on page 196).

2. `version` (uhalf)
   A 2-byte version identifier containing the value 5.

3. `address_size` (ubyte)
   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.

4. `segment_selector_size` (ubyte)
   A 1-byte unsigned integer containing the size in bytes of a segment selector on the target system.

5. `offset_entry_count` (uword)
   A 4-byte count of the number of offsets that follow the header. This count may be zero.

Immediately following the header is an array of offsets. This array is followed by a series of range lists.

If the `offset_entry_count` is non-zero, there is one offset for each range list. The contents of the i^{th} offset is the offset (an unsigned integer) from the beginning of the offset array to the location of the i^{th} range list. In the 32-bit DWARF format, each offset is 4-bytes in size; in the 64-bit DWARF format, each offset is 8-bytes in size (see Section 7.4 on page 196).

If the `offset_entry_count` is zero, then DW_FORM_rnglistx cannot be used to access a range list; DW_FORM_sec_offset must be used instead. If the `offset_entry_count` is non-zero, then DW_FORM_rnglistx may be used to access a range list; this is necessary in split units and may be more compact than using DW_FORM_sec_offsetin non-split units.

Range lists are described in Section 2.17.3 on page 52.

The segment size is given by the `segment_selector_size` field of the header, and the address size is given by the `address_size` field of the header. If the `segment_selector_size` field in the header is zero, the segment selector is omitted from the range list entries.

The DW_AT_rnglists_base attribute points to the first offset following the header. The range lists are referenced by the index of the position of their corresponding offset in the array of offsets, which indirectly specifies the offset to the target list.

7.29 Location List Table

Each `.debug_loclists` or `.debug_loclists.dwo` section begins with a header containing:

1. `unit_length` (initial length)
   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 `0xfffffff0`); 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 on page 196).

2. `version` (uhalf)
   A 2-byte version identifier containing the value 5.

3. `address_size` (ubyte)
   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.

4. `segment_selector_size` (ubyte)
   A 1-byte unsigned integer containing the size in bytes of a segment selector on the target system.

5. `offset_entry_count` (uword)
   A 4-byte count of the number of offsets that follow the header. This count may be zero.

Immediately following the header is an array of offsets. This array is followed by a series of location lists.

If the `offset_entry_count` is non-zero, there is one offset for each location list. The contents of the i^{th} offset is the offset (an unsigned integer) from the beginning of the offset array to the location of the i^{th} location list. In the 32-bit DWARF format, each offset is 4-bytes in size; in the 64-bit DWARF format, each offset is 8-bytes in size (see Section 7.4 on page 196).

If the `offset_entry_count` is zero, then DW_FORM_loclistx cannot be used to access a location list; DW_FORM_sec_offset must be used instead. If the `offset_entry_count` is non-zero, then DW_FORM_loclistx may be used to access a location list; this is necessary in split units and may be more compact than using DW_FORM_sec_offsetin non-split units.

Location lists are described in Section 2.6.2 on page 43.

The segment size is given by the `segment_selector_size` field of the header, and the address size is given by the `address_size` field of the header. If the `segment_selector_size` field in the header is zero, the segment selector is omitted from location list entries.

The DW_AT_loclists_base attribute points to the first offset following the header. The location lists are referenced by the index of the position of their corresponding offset in the array of offsets, which indirectly specifies the offset to the target list.

7.30 Dependencies and Constraints

The debugging information in this format is intended to exist in sections of an object file, or an equivalent separate file or database, having names beginning with the prefix ".debug_" (see Appendix G on page 415 for a complete list of such names). Except as specifically specified, this information is not aligned on 2-, 4- or 8-byte boundaries. 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 4-byte section offset that occurs at an arbitrary alignment.
- For the 64-bit DWARF format and a target architecture with 32-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 a 4-byte address or 8-byte section offset that occurs at an arbitrary alignment.

It is expected that this will be required only for very large 32-bit programs or by those architectures which support a mix of 32-bit and 64-bit code and data within the same executable object.

- 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.

7.31 Integer Representation Names

The sizes of the integers used in the lookup by name, lookup by address, line number, call frame information and other sections are given in Table 7.31.

Table 7.31: Integer representation names

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

7.32 Type Signature Computation

A type signature is used by a DWARF consumer to resolve type references to the type definitions that are contained in type units (see Section 3.1.4 on page 68).

A type signature is computed only by a DWARF producer; a consumer need only compare two type signatures to check for equality.

The type signature for a type T0 is formed from the MD5[5] digest of a flattened description of the type. The flattened description of the type is a byte sequence derived from the DWARF encoding of the type as follows:

1. Start with an empty sequence S and a list V of visited types, where V is initialized to a list containing the type T0 as its single element. Elements in V are indexed from 1, so that V[1] is T0.

[5]MD5 Message Digest Algorithm, R.L. Rivest, RFC 1321, April 1992

2. If the debugging information entry represents a type that is nested inside another type or a namespace, append to S the type’s context as follows: For each surrounding type or namespace, beginning with the outermost such construct, append the letter ’C’, the DWARF tag of the construct, and the name (taken from the DW_AT_name attribute) of the type or namespace (including its trailing null byte).

3. Append to S the letter ’D’, followed by the DWARF tag of the debugging information entry.

4. For each of the attributes in Table 7.32 on the following page that are present in the debugging information entry, in the order listed, append to S a marker letter (see below), the DWARF attribute code, and the attribute value.

   Note that except for the initial DW_AT_name attribute, attributes are appended in order according to the alphabetical spelling of their identifier.

   If an implementation defines any vendor-specific attributes, any such attributes that are essential to the definition of the type are also included at the end of the above list, in their own alphabetical suborder.

   An attribute that refers to another type entry T is processed as follows:

   a) If T is in the list V at some V[x], use the letter ’R’ as the marker and use the unsigned LEB128 encoding of x as the attribute value.

   b) Otherwise, append type T to the list V, then use the letter ’T’ as the marker, process the type T recursively by performing Steps 2 through 7, and use the result as the attribute value.

   Other attribute values use the letter ’A’ as the marker, and the value consists of the form code (encoded as an unsigned LEB128 value) followed by the encoding of the value according to the form code. To ensure reproducibility of the signature, the set of forms used in the signature computation is limited to the following: DW_FORM_sdata, DW_FORM_flag, DW_FORM_string, DW_FORM_exprloc, and DW_FORM_block.

5. If the tag in Step 3 is one of DW_TAG_pointer_type, DW_TAG_reference_type, DW_TAG_rvalue_reference_type, DW_TAG_ptr_to_member_type, or DW_TAG_friend, and the referenced type (via the DW_AT_type or DW_AT_friend attribute) has a DW_AT_name attribute, append to S the letter ’N’, the DWARF attribute code (DW_AT_type or DW_AT_friend), the context of the type (according to the method in Step 2), the letter ’E’, and the name of the type. For DW_TAG_friend, if the referenced entry is a DW_TAG_subprogram, the context is omitted and the

Table 7.32: Attributes used in type signature computation

DW_AT_name	DW_AT_endianity
DW_AT_accessibility	DW_AT_enum_class
DW_AT_address_class	DW_AT_explicit
DW_AT_alignment	DW_AT_is_optional
DW_AT_allocated	DW_AT_location
DW_AT_artificial	DW_AT_lower_bound
DW_AT_associated	DW_AT_mutable
DW_AT_binary_scale	DW_AT_ordering
DW_AT_bit_size	DW_AT_picture_string
DW_AT_bit_stride	DW_AT_prototyped
DW_AT_byte_size	DW_AT_rank
DW_AT_byte_stride	DW_AT_reference
DW_AT_const_expr	DW_AT_rvalue_reference
DW_AT_const_value	DW_AT_small
DW_AT_containing_type	DW_AT_segment
DW_AT_count	DW_AT_string_length
DW_AT_data_bit_offset	DW_AT_string_length_bit_size
DW_AT_data_location	DW_AT_string_length_byte_size
DW_AT_data_member_location	DW_AT_threads_scaled
DW_AT_decimal_scale	DW_AT_upper_bound
DW_AT_decimal_sign	DW_AT_use_location
DW_AT_default_value	DW_AT_use_UTF8
DW_AT_digit_count	DW_AT_variable_parameter
DW_AT_discr	DW_AT_virtuality
DW_AT_discr_list	DW_AT_visibility
DW_AT_discr_value	DW_AT_vtable_elem_location
DW_AT_encoding	

name to be used is the ABI-specific name of the subprogram (for example, the mangled linker name).

6. If the tag in Step 3 is not one of DW_TAG_pointer_type, DW_TAG_reference_type, DW_TAG_rvalue_reference_type, DW_TAG_ptr_to_member_type, or DW_TAG_friend, but has a DW_AT_type attribute, or if the referenced type (via the DW_AT_type or DW_AT_friend attribute) does not have a DW_AT_name attribute, the attribute is processed according to the method in Step 4 for an attribute that refers to another type entry.

7. Visit each child C of the debugging information entry as follows: If C is a nested type entry or a member function entry, and has a DW_AT_name attribute, append to S the letter 'S', the tag of C, and its name; otherwise, process C recursively by performing Steps 3 through 7, appending the result to S. Following the last child (or if there are no children), append a zero byte.

For the purposes of this algorithm, if a debugging information entry S has a DW_AT_specification attribute that refers to another entry D (which has a DW_AT_declaration attribute), then S inherits the attributes and children of D, and S is processed as if those attributes and children were present in the entry S. Exception: if a particular attribute is found in both S and D, the attribute in S is used and the corresponding one in D is ignored.

DWARF tag and attribute codes are appended to the sequence as unsigned LEB128 values, using the values defined earlier in this chapter.

A grammar describing this computation may be found in Appendix E.2.2 on page 385.

An attribute that refers to another type entry is recursively processed or replaced with the name of the referent (in Step 4, 5 or 6). If neither treatment applies to an attribute that references another type entry, the entry that contains that attribute is not suitable for a separate type unit.

If a debugging information entry contains an attribute from the list above that would require an unsupported form, that entry is not suitable for a separate type unit.

A type is suitable for a separate type unit only if all of the type entries that it contains or refers to in Steps 6 and 7 are themselves suitable for a separate type unit.

Where the DWARF producer may reasonably choose two or more different forms for a given attribute, it should choose the simplest possible form in computing the signature. (For example, a constant value should be preferred to a location expression when possible.)

Once the string S has been formed from the DWARF encoding, an 16-byte MD5 digest is computed for the string and the last eight bytes are taken as the type signature.

The string S is intended to be a flattened representation of the type that uniquely identifies that type (that is, a different type is highly unlikely to produce the same string).

A debugging information entry is not be placed in a separate type unit if any of the following apply:

- *The entry has an attribute whose value is a location description, and the location description contains a reference to another debugging information entry (for example, a DW_OP_call_ref operator), as it is unlikely that the entry will remain identical across compilation units.*
- *The entry has an attribute whose value refers to a code location or a location list.*
- *The entry has an attribute whose value refers to another debugging information entry that does not represent a type.*

Certain attributes are not included in the type signature:

- *The DW_AT_declaration attribute is not included because it indicates that the debugging information entry represents an incomplete declaration, and incomplete declarations should not be placed in separate type units.*
- *The DW_AT_description attribute is not included because it does not provide any information unique to the defining declaration of the type.*
- *The DW_AT_decl_file, DW_AT_decl_line, and DW_AT_decl_column attributes are not included because they may vary from one source file to the next, and would prevent two otherwise identical type declarations from producing the same MD5 digest.*
- *The DW_AT_object_pointer attribute is not included because the information it provides is not necessary for the computation of a unique type signature.*

Nested types and some types referred to by a debugging information entry are encoded by name rather than by recursively encoding the type to allow for cases where a complete definition of the type might not be available in all compilation units.

If a type definition contains the definition of a member function, it cannot be moved as is into a type unit, because the member function contains attributes that are unique to that compilation unit. Such a type definition can be moved to a type unit by rewriting the debugging information entry tree, moving the member function declaration into a separate declaration tree, and replacing the function definition in the type with a non-defining declaration of the function (as if the function had been defined out of line).

An example that illustrates the computation of an MD5 digest may be found in Appendix E.2 on page 375.

7.33 Name Table Hash Function

The hash function used for hashing name strings in the accelerated access name index table (see Section 6.1 on page 135) is defined in C as shown in Figure 7.1 following.[6]

```
uint32_t /* must be a 32-bit integer type */
    hash(unsigned char *str)
    {
        uint32_t hash = 5381;
        int c;

        while (c = *str++)
            hash = hash * 33 + c;

        return hash;
    }
```

Figure 7.1: Name Table Hash Function Definition

[6] This hash function is sometimes known as the "Bernstein hash function" or the "DJB hash function" (see, for example, http://en.wikipedia.org/wiki/List_of_hash_functions or http://stackoverflow.com/questions/10696223/reason-for-5381-number-in-djb-hash-function).

Appendix A

Attributes by Tag Value (Informative)

The table 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 information entry. Therefore, the table may be taken as instructive, but cannot be considered definitive.

In the following table, the following special conventions apply:

1. The DECL pseudo-attribute stands for all three of the declaration coordinates DW_AT_decl_column, DW_AT_decl_file and DW_AT_decl_line.

2. The DW_AT_description attribute can be used on any debugging information entry that may have a DW_AT_name attribute. For simplicity, this attribute is not explicitly shown.

3. The DW_AT_sibling attribute can be used on any debugging information entry. For simplicity, this attribute is not explicitly shown.

4. The DW_AT_abstract_origin attribute can be used with almost any debugging information entry; the exceptions are mostly the compilation unit-like entries. For simplicity, this attribute is not explicitly shown.

Table A.1: Attributes by tag value

TAG name	Applicable attributes
DW_TAG_access_declaration	DECL
	DW_AT_accessibility
	DW_AT_name
DW_TAG_array_type	DECL
	DW_AT_accessibility
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_bit_size
	DW_AT_bit_stride
	DW_AT_byte_size
	DW_AT_data_location
	DW_AT_declaration
	DW_AT_name
	DW_AT_ordering
	DW_AT_rank
	DW_AT_specification
	DW_AT_start_scope
	DW_AT_type
	DW_AT_visibility
DW_TAG_atomic_type	DECL
	DW_AT_alignment
	DW_AT_name
	DW_AT_type

Continued on next page

TAG name	**Applicable attributes**
DW_TAG_base_type	DECL
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_binary_scale
	DW_AT_bit_size
	DW_AT_byte_size
	DW_AT_data_bit_offset
	DW_AT_data_location
	DW_AT_decimal_scale
	DW_AT_decimal_sign
	DW_AT_digit_count
	DW_AT_encoding
	DW_AT_endianity
	DW_AT_name
	DW_AT_picture_string
	DW_AT_small
DW_TAG_call_site	DW_AT_call_column
	DW_AT_call_file
	DW_AT_call_line
	DW_AT_call_origin
	DW_AT_call_pc
	DW_AT_call_return_pc
	DW_AT_call_tail_call
	DW_AT_call_target
	DW_AT_call_target_clobbered
	DW_AT_type
DW_TAG_call_site_parameter	DW_AT_call_data_location
	DW_AT_call_data_value
	DW_AT_call_parameter
	DW_AT_call_value
	DW_AT_location
	DW_AT_name
	DW_AT_type

Continued on next page

TAG name	**Applicable attributes**
DW_TAG_catch_block	DECL
	DW_AT_entry_pc
	DW_AT_high_pc
	DW_AT_low_pc
	DW_AT_ranges
	DW_AT_segment
DW_TAG_class_type	DECL
	DW_AT_accessibility
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_bit_size
	DW_AT_byte_size
	DW_AT_calling_convention
	DW_AT_data_location
	DW_AT_declaration
	DW_AT_export_symbols
	DW_AT_name
	DW_AT_signature
	DW_AT_specification
	DW_AT_start_scope
	DW_AT_visibility
DW_TAG_coarray_type	DECL
	DW_AT_alignment
	DW_AT_bit_size
	DW_AT_byte_size
	DW_AT_name
	DW_AT_type

Continued on next page

TAG name	Applicable attributes
DW_TAG_common_block	DECL
	DW_AT_declaration
	DW_AT_linkage_name
	DW_AT_location
	DW_AT_name
	DW_AT_segment
	DW_AT_visibility
DW_TAG_common_inclusion	DECL
	DW_AT_common_reference
	DW_AT_declaration
	DW_AT_visibility
DW_TAG_compile_unit	DW_AT_addr_base
	DW_AT_base_types
	DW_AT_comp_dir
	DW_AT_entry_pc
	DW_AT_identifier_case
	DW_AT_high_pc
	DW_AT_language
	DW_AT_low_pc
	DW_AT_macros
	DW_AT_main_subprogram
	DW_AT_name
	DW_AT_producer
	DW_AT_ranges
	DW_AT_rnglists_base
	DW_AT_segment
	DW_AT_stmt_list
	DW_AT_str_offsets_base
	DW_AT_use_UTF8
DW_TAG_condition	DECL
	DW_AT_name

Continued on next page

TAG name	**Applicable attributes**
DW_TAG_const_type	DECL
	DW_AT_alignment
	DW_AT_name
	DW_AT_type
DW_TAG_constant	DECL
	DW_AT_accessibility
	DW_AT_const_value
	DW_AT_declaration
	DW_AT_endianity
	DW_AT_external
	DW_AT_linkage_name
	DW_AT_name
	DW_AT_start_scope
	DW_AT_type
	DW_AT_visibility
DW_TAG_dwarf_procedure	DW_AT_location
DW_TAG_dynamic_type	DECL
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_data_location
	DW_AT_name
	DW_AT_type
DW_TAG_entry_point	DECL
	DW_AT_address_class
	DW_AT_frame_base
	DW_AT_linkage_name
	DW_AT_low_pc
	DW_AT_name
	DW_AT_return_addr
	DW_AT_segment
	DW_AT_static_link
	DW_AT_type

Continued on next page

TAG name	**Applicable attributes**
DW_TAG_enumeration_type	DECL
	DW_AT_accessibility
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_bit_size
	DW_AT_bit_stride
	DW_AT_byte_size
	DW_AT_byte_stride
	DW_AT_data_location
	DW_AT_declaration
	DW_AT_enum_class
	DW_AT_name
	DW_AT_signature
	DW_AT_specification
	DW_AT_start_scope
	DW_AT_type
	DW_AT_visibility
DW_TAG_enumerator	DECL
	DW_AT_const_value
	DW_AT_name
DW_TAG_file_type	DECL
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_bit_size
	DW_AT_byte_size
	DW_AT_data_location
	DW_AT_name
	DW_AT_start_scope
	DW_AT_type
	DW_AT_visibility

Continued on next page

TAG name	**Applicable attributes**
DW_TAG_formal_parameter	DECL
	DW_AT_artificial
	DW_AT_const_value
	DW_AT_default_value
	DW_AT_endianity
	DW_AT_is_optional
	DW_AT_location
	DW_AT_name
	DW_AT_segment
	DW_AT_type
	DW_AT_variable_parameter
DW_TAG_friend	DECL
	DW_AT_friend
DW_TAG_generic_subrange	DECL
	DW_AT_accessibility
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_bit_size
	DW_AT_bit_stride
	DW_AT_byte_size
	DW_AT_byte_stride
	DW_AT_count
	DW_AT_data_location
	DW_AT_declaration
	DW_AT_lower_bound
	DW_AT_name
	DW_AT_threads_scaled
	DW_AT_type
	DW_AT_upper_bound
	DW_AT_visibility
DW_TAG_immutable_type	DECL
	DW_AT_name
	DW_AT_type

Continued on next page

TAG name	Applicable attributes
DW_TAG_imported_declaration	DECL
	DW_AT_accessibility
	DW_AT_import
	DW_AT_name
	DW_AT_start_scope
DW_TAG_imported_module	DECL
	DW_AT_import
	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_type
	DW_AT_virtuality
DW_TAG_inlined_subroutine	DW_AT_call_column
	DW_AT_call_file
	DW_AT_call_line
	DW_AT_const_expr
	DW_AT_entry_pc
	DW_AT_high_pc
	DW_AT_low_pc
	DW_AT_ranges
	DW_AT_return_addr
	DW_AT_segment
	DW_AT_start_scope
	DW_AT_trampoline
DW_TAG_interface_type	DECL
	DW_AT_accessibility
	DW_AT_alignment
	DW_AT_name
	DW_AT_signature
	DW_AT_start_scope

Continued on next page

TAG name	Applicable attributes
DW_TAG_label	DECL
	DW_AT_low_pc
	DW_AT_name
	DW_AT_segment
	DW_AT_start_scope
DW_TAG_lexical_block	DECL
	DW_AT_entry_pc
	DW_AT_high_pc
	DW_AT_low_pc
	DW_AT_name
	DW_AT_ranges
	DW_AT_segment
DW_TAG_member	DECL
	DW_AT_accessibility
	DW_AT_artificial
	DW_AT_bit_size
	DW_AT_byte_size
	DW_AT_data_bit_offset
	DW_AT_data_member_location
	DW_AT_declaration
	DW_AT_mutable
	DW_AT_name
	DW_AT_type
	DW_AT_visibility

Continued on next page

TAG name	**Applicable attributes**
DW_TAG_module	DECL
	DW_AT_accessibility
	DW_AT_declaration
	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_specification
	DW_AT_visibility
DW_TAG_namelist	DECL
	DW_AT_accessibility
	DW_AT_declaration
	DW_AT_name
	DW_AT_visibility
DW_TAG_namelist_item	DECL
	DW_AT_namelist_item
DW_TAG_namespace	DECL
	DW_AT_export_symbols
	DW_AT_extension
	DW_AT_name
	DW_AT_start_scope
DW_TAG_packed_type	DECL
	DW_AT_alignment
	DW_AT_name
	DW_AT_type

Continued on next page

TAG name	Applicable attributes
DW_TAG_partial_unit	DW_AT_addr_base
	DW_AT_base_types
	DW_AT_comp_dir
	DW_AT_dwo_name
	DW_AT_entry_pc
	DW_AT_identifier_case
	DW_AT_high_pc
	DW_AT_language
	DW_AT_low_pc
	DW_AT_macros
	DW_AT_main_subprogram
	DW_AT_name
	DW_AT_producer
	DW_AT_ranges
	DW_AT_rnglists_base
	DW_AT_segment
	DW_AT_stmt_list
	DW_AT_str_offsets_base
	DW_AT_use_UTF8
DW_TAG_pointer_type	DECL
	DW_AT_address_class
	DW_AT_alignment
	DW_AT_bit_size
	DW_AT_byte_size
	DW_AT_name
	DW_AT_type

Continued on next page

TAG name	Applicable attributes
DW_TAG_ptr_to_member_type	DECL
	DW_AT_address_class
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_containing_type
	DW_AT_data_location
	DW_AT_declaration
	DW_AT_name
	DW_AT_type
	DW_AT_use_location
	DW_AT_visibility
DW_TAG_reference_type	DECL
	DW_AT_address_class
	DW_AT_alignment
	DW_AT_bit_size
	DW_AT_byte_size
	DW_AT_name
	DW_AT_type
DW_TAG_restrict_type	DECL
	DW_AT_alignment
	DW_AT_name
	DW_AT_type
DW_TAG_rvalue_reference_type	DECL
	DW_AT_address_class
	DW_AT_alignment
	DW_AT_bit_size
	DW_AT_byte_size
	DW_AT_name
	DW_AT_type

Continued on next page

TAG name	Applicable attributes
DW_TAG_set_type	DECL
	DW_AT_accessibility
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_bit_size
	DW_AT_byte_size
	DW_AT_data_location
	DW_AT_declaration
	DW_AT_name
	DW_AT_start_scope
	DW_AT_type
	DW_AT_visibility
DW_TAG_shared_type	DECL
	DW_AT_count
	DW_AT_alignment
	DW_AT_name
	DW_AT_type
DW_TAG_skeleton_unit	DW_AT_addr_base
	DW_AT_comp_dir
	DW_AT_dwo_name
	DW_AT_high_pc
	DW_AT_low_pc
	DW_AT_ranges
	DW_AT_rnglists_base
	DW_AT_stmt_list
	DW_AT_str_offsets_base
	DW_AT_use_UTF8

Continued on next page

TAG name	**Applicable attributes**
DW_TAG_string_type	DECL
	DW_AT_alignment
	DW_AT_accessibility
	DW_AT_allocated
	DW_AT_associated
	DW_AT_bit_size
	DW_AT_byte_size
	DW_AT_data_location
	DW_AT_declaration
	DW_AT_name
	DW_AT_start_scope
	DW_AT_string_length
	DW_AT_string_length_bit_size
	DW_AT_string_length_byte_size
	DW_AT_visibility
DW_TAG_structure_type	DECL
	DW_AT_accessibility
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_bit_size
	DW_AT_byte_size
	DW_AT_calling_convention
	DW_AT_data_location
	DW_AT_declaration
	DW_AT_export_symbols
	DW_AT_name
	DW_AT_signature
	DW_AT_specification
	DW_AT_start_scope
	DW_AT_visibility

Continued on next page

TAG name	Applicable attributes
DW_TAG_subprogram	DECL
	DW_AT_accessibility
	DW_AT_address_class
	DW_AT_alignment
	DW_AT_artificial
	DW_AT_calling_convention
	DW_AT_declaration
	DW_AT_defaulted
	DW_AT_deleted
	DW_AT_elemental
	DW_AT_entry_pc
	DW_AT_explicit
	DW_AT_external
	DW_AT_frame_base
	DW_AT_high_pc
	DW_AT_inline
	DW_AT_linkage_name
	DW_AT_low_pc
	DW_AT_main_subprogram
	DW_AT_name
	DW_AT_noreturn
	DW_AT_object_pointer
	DW_AT_prototyped
	DW_AT_pure
	DW_AT_ranges
	DW_AT_recursive
	DW_AT_reference
	DW_AT_return_addr
	DW_AT_rvalue_reference
	DW_AT_segment
	DW_AT_specification
	Additional attributes continue on next page
Continued on next page	

TAG name	**Applicable attributes**
DW_TAG_subprogram (cont.)	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_accessibility
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_bit_size
	DW_AT_bit_stride
	DW_AT_byte_size
	DW_AT_byte_stride
	DW_AT_count
	DW_AT_data_location
	DW_AT_declaration
	DW_AT_lower_bound
	DW_AT_name
	DW_AT_threads_scaled
	DW_AT_type
	DW_AT_upper_bound
	DW_AT_visibility

Continued on next page

TAG name	Applicable attributes
DW_TAG_subroutine_type	DECL
	DW_AT_accessibility
	DW_AT_address_class
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_data_location
	DW_AT_declaration
	DW_AT_name
	DW_AT_prototyped
	DW_AT_reference
	DW_AT_rvalue_reference
	DW_AT_start_scope
	DW_AT_type
	DW_AT_visibility
DW_TAG_template_alias	DECL
	DW_AT_accessibility
	DW_AT_allocated
	DW_AT_associated
	DW_AT_data_location
	DW_AT_declaration
	DW_AT_name
	DW_AT_signature
	DW_AT_start_scope
	DW_AT_type
	DW_AT_visibility
DW_TAG_template_type_parameter	DECL
	DW_AT_default_value
	DW_AT_name
	DW_AT_type

Continued on next page

TAG name	Applicable attributes
DW_TAG_template_value_parameter	DECL
	DW_AT_const_value
	DW_AT_default_value
	DW_AT_name
	DW_AT_type
DW_TAG_thrown_type	DECL
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_data_location
	DW_AT_name
	DW_AT_type
DW_TAG_try_block	DECL
	DW_AT_entry_pc
	DW_AT_high_pc
	DW_AT_low_pc
	DW_AT_ranges
	DW_AT_segment
DW_TAG_typedef	DECL
	DW_AT_accessibility
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_data_location
	DW_AT_declaration
	DW_AT_name
	DW_AT_start_scope
	DW_AT_type
	DW_AT_visibility
DW_TAG_type_unit	DW_AT_language
	DW_AT_stmt_list
	DW_AT_str_offsets_base
	DW_AT_use_UTF8

Continued on next page

TAG name	**Applicable attributes**
DW_TAG_union_type	DECL
	DW_AT_accessibility
	DW_AT_alignment
	DW_AT_allocated
	DW_AT_associated
	DW_AT_bit_size
	DW_AT_byte_size
	DW_AT_calling_convention
	DW_AT_data_location
	DW_AT_declaration
	DW_AT_export_symbols
	DW_AT_name
	DW_AT_signature
	DW_AT_specification
	DW_AT_start_scope
	DW_AT_visibility
DW_TAG_unspecified_parameters	DECL
	DW_AT_artificial
DW_TAG_unspecified_type	DECL
	DW_AT_name

Continued on next page

TAG name	Applicable attributes
DW_TAG_variable	DECL
	DW_AT_accessibility
	DW_AT_alignment
	DW_AT_artificial
	DW_AT_const_expr
	DW_AT_const_value
	DW_AT_declaration
	DW_AT_endianity
	DW_AT_external
	DW_AT_linkage_name
	DW_AT_location
	DW_AT_name
	DW_AT_segment
	DW_AT_specification
	DW_AT_start_scope
	DW_AT_type
	DW_AT_visibility
DW_TAG_variant	DECL
	DW_AT_accessibility
	DW_AT_declaration
	DW_AT_discr_list
	DW_AT_discr_value
DW_TAG_variant_part	DECL
	DW_AT_accessibility
	DW_AT_declaration
	DW_AT_discr
	DW_AT_type
DW_TAG_volatile_type	DECL
	DW_AT_name
	DW_AT_type

Continued on next page

TAG name	Applicable attributes
DW_TAG_with_stmt	DECL
	DW_AT_accessibility
	DW_AT_address_class
	DW_AT_declaration
	DW_AT_entry_pc
	DW_AT_high_pc
	DW_AT_location
	DW_AT_low_pc
	DW_AT_ranges
	DW_AT_segment
	DW_AT_type
	DW_AT_visibility

Appendix B

Debug Section Relationships (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 diagrams and associated notes on the following pages.

In the figures, a section is shown as a shaded oval with the name of the section inside. References from one section to another are shown by an arrow. In the first figure, the arrow is annotated with an unshaded box which contains an indication of the construct (such as an attribute or form) that encodes the reference. In the second figure, this box is left out for reasons of space in favor of a label annotation that is explained in the subsequent notes.

B.1 Normal DWARF Section Relationships

Figure B.1 following illustrates the DWARF section relations without split DWARF object files involved. Similarly, it does not show the relationships between the main debugging sections of an executable or sharable file and a related supplementary object file.

B.2 Split DWARF Section Relationships

Figure B.2 illustrates the DWARF section relationships for split DWARF object files. However, it does not show the relationships between the main debugging sections of an executable or shareable file and a related supplementary object file.

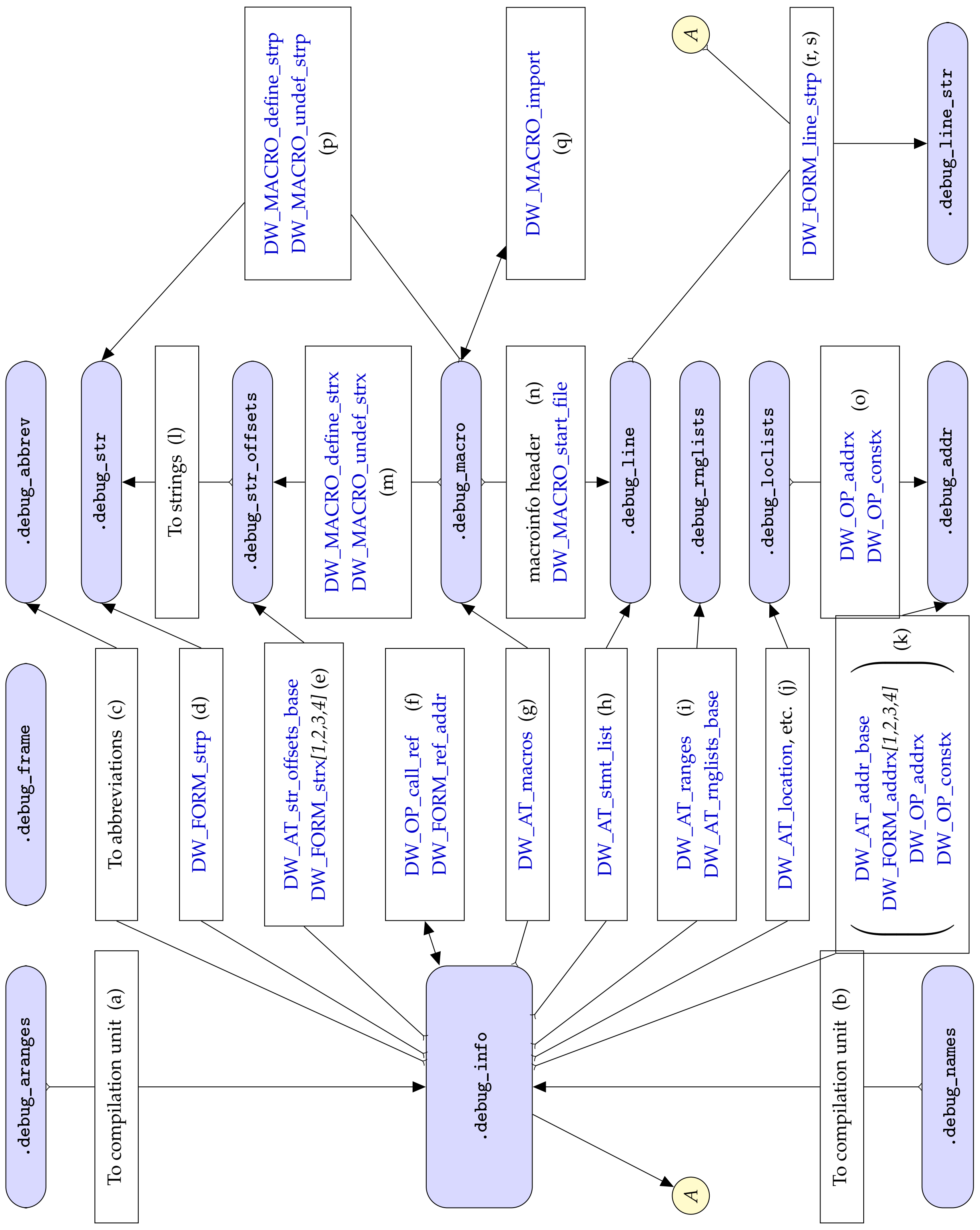


Figure B.1: Debug section relationships

Notes for Figure B.1

(a) `.debug_aranges` **to** `.debug_info`
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_names` **to** `.debug_info`
The list of compilation units following the header contains the offsets in the `.debug_info` section of the corresponding compilation unit headers (not the compilation unit entries).

(c) `.debug_info` **to** `.debug_abbrev`
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` **to** `.debug_str`
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_info` **to** `.debug_str_offsets`
The value of the DW_AT_str_offsets_base attribute in a DW_TAG_compile_unit, DW_TAG_type_unit or DW_TAG_partial_unit DIE is the offset in the `.debug_str_offsets` section of the string offsets table for that unit. In addition, attribute values of class string may have one of the forms DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 or DW_FORM_strx4, whose value is an index into the string offsets table.

(f) `.debug_info` **to** `.debug_info`
The operand of the DW_OP_call_ref DWARF expression operator is the offset of a debugging information entry in the `.debug_info` section of another compilation. Similarly for attribute operands that use DW_FORM_ref_addr.

(g) `.debug_info` **to** `.debug_macro`
An attribute value of class macptr (specifically form DW_FORM_sec_offset) is an offset within the `.debug_macro` section of the beginning of the macro information for the referencing unit.

(h) `.debug_info` **to** `.debug_line`
An attribute value of class lineptr (specifically form DW_FORM_sec_offset) is an offset in the `.debug_line` section of the beginning of the line number information for the referencing unit.

(i) `.debug_info` **to** `.debug_rnglists`
An attribute value of class rnglist (specifically form DW_FORM_rnglistx or DW_FORM_sec_offset) is an index or offset within the `.debug_rnglists` section of a range list.

(j) `.debug_info` **to** `.debug_loclists`
An attribute value of class loclist (specifically form DW_FORM_loclistx or DW_FORM_sec_offset) is an index or offset within the `.debug_loclists` section of a location list.

(k) `.debug_info` **to** `.debug_addr`
The value of the DW_AT_addr_base attribute in the DW_TAG_compile_unit or DW_TAG_partial_unit DIE is the offset in the `.debug_addr` section of the machine addresses for that unit. DW_FORM_addrx, DW_FORM_addrx1, DW_FORM_addrx2, DW_FORM_addrx3, DW_FORM_addrx4, DW_OP_addrx and DW_OP_constx contain indices relative to that offset.

(l) `.debug_str_offsets` **to** `.debug_str`
Entries in the string offsets table are offsets to the corresponding string text in the `.debug_str` section.

(m) `.debug_macro` **to** `.debug_str_offsets`
The second operand of a DW_MACRO_define_strx or DW_MACRO_undef_strx macro information entry is an index into the string offset table in the `.debug_str_offsets` section.

(n) `.debug_macro` **to** `.debug_line`
The second operand of DW_MACRO_start_file refers to a file entry in the `.debug_line` section relative to the start of that section given in the macro information header.

(o) `.debug_loclists` **to** `.debug_addr`
DW_OP_addrx and DW_OP_constx operators that occur in the `.debug_loclists` section refer indirectly to the `.debug_addr` section by way of the DW_AT_addr_base attribute in the associated `.debug_info` section.

(p) `.debug_macro` **to** `.debug_str`
The second operand of a DW_MACRO_define_strp or DW_MACRO_undef_strp macro information entry is an index into the string table in the `.debug_str` section.

(q) `.debug_macro` **to** `.debug_macro`
The operand of a DW_MACRO_import macro information entry is an offset into another part of the `.debug_macro` section to the header for the sequence to be replicated.

(r) `.debug_line` **to** `.debug_line_str`
The value of a DW_FORM_line_strp form refers to a string section specific to the line number table. This form can be used in a `.debug_line` section (as well as in a `.debug_info` section).

(s) `.debug_info` **to** `.debug_line_str`
The value of a DW_FORM_line_strp form refers to a string section specific to the line number table. This form can be used in a `.debug_info` section (as well as in a `.debug_line` section).[1]

[1] The circled (s) connects to the circled (s)′ via hyperspace (a wormhole).

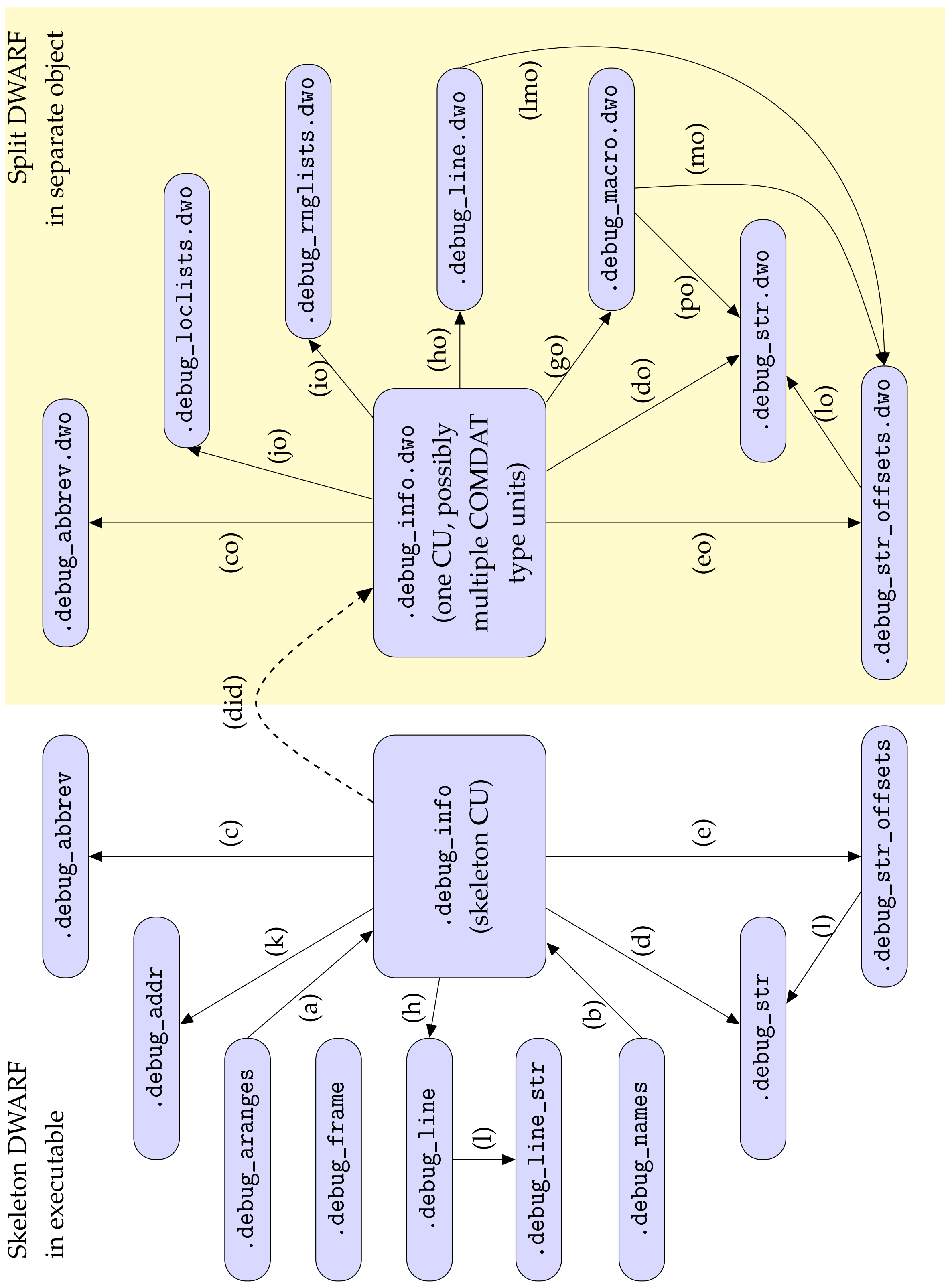


Figure B.2: Split DWARF section relationships

Notes for Figure B.2

(a) `.debug_aranges` **to** `.debug_info`
The `debug_info_offset` field in the header is the offset in the `.debug_info` section of the corresponding compilation unit header of the skeleton `.debug_info` section (not the compilation unit entry). The DW_AT_dwo_name attribute in the `.debug_info` skeleton connects the ranges to the full compilation unit in `.debug_info.dwo`.

(b) `.debug_names` **to** `.debug_info`
The `.debug_names` section offsets lists provide an offset for the skeleton compilation unit and eight byte signatures for the type units that appear only in the `.debug_info.dwo`. The DIE offsets for these compilation units and type units refer to the DIEs in the `.debug_info.dwo` section for the respective compilation unit and type units.

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

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

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

(did) `.debug_info` **to** `.debug_info.dwo`
The DW_AT_dwo_name attribute in a skeleton unit identifies the file containing the corresponding `.dwo` (split) data.

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

(e) `.debug_info` **to** `.debug_str_offsets`
Attribute values of class string may have one of the forms DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 or DW_FORM_strx4, whose value is an index into the `.debug_str_offsets` section for the corresponding string.

(eo) `.debug_info.dwo` **to** `.debug_str_offsets.dwo`
Attribute values of class string may have one of the forms DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 or DW_FORM_strx4, whose value is an index into the `.debug_str_offsets.dwo` section for the corresponding string.

(fo) `.debug_info.dwo` **to** `.debug_info.dwo`
The operand of the DW_OP_call_ref DWARF expression operator is the offset of a debugging information entry in the `.debug_info.dwo` section of another compilation unit. Similarly for attribute operands that use DW_FORM_ref_addr. See Section 2.5.1.5 on page 35.

(go) `.debug_info.dwo` **to** `.debug_macro.dwo`
An attribute of class macptr (specifically DW_AT_macros with form DW_FORM_sec_offset) is an offset within the `.debug_macro.dwo` section of the beginning of the macro information for the referencing unit.

(h) `.debug_info` **(skeleton) to** `.debug_line`
An attribute value of class lineptr (specifically DW_AT_stmt_list with form DW_FORM_sec_offset) is an offset within the `.debug_line` section of the beginning of the line number information for the referencing unit.

(ho) `.debug_info.dwo` **to** `.debug_line.dwo` **(skeleton)**
An attribute value of class lineptr (specifically DW_AT_stmt_list with form DW_FORM_sec_offset) is an offset within the `.debug_line.dwo` section of the beginning of the line number header information for the referencing unit (the line table details are not in `.debug_line.dwo` but the line header with its list of file names is present).

(io) `.debug_info.dwo` **to** `.debug_rnglists.dwo`
An attribute value of class rnglist (specifically DW_AT_ranges with form DW_FORM_rnglistx or DW_FORM_sec_offset) is an index or offset within the `.debug_rnglists.dwo` section of a range list. The format of `.debug_rnglists.dwo` location list entries is restricted to a subset of those in `.debug_rnglists`. See Section 2.17.3 on page 52 for details.

(jo) `.debug_info.dwo` **to** `.debug_loclists.dwo`
An attribute value of class loclist (specifically with form DW_FORM_loclistx or DW_FORM_sec_offset) is an index or offset within the `.debug_loclists.dwo` section of a location list. The format of `.debug_loclists.dwo` location list entries is restricted to a subset of those in `.debug_loclists`. See Section 2.6.2 on page 43 for details.

(k) `.debug_info` **to** `.debug_addr`
The value of the DW_AT_addr_base attribute in the DW_TAG_compile_unit, DW_TAG_partial_unit or DW_TAG_type_unit DIE is the offset in the `.debug_addr` section of the machine addresses for that unit. DW_FORM_addrx, DW_FORM_addrx1, DW_FORM_addrx2, DW_FORM_addrx3, DW_FORM_addrx4, DW_OP_addrx and DW_OP_constx contain indices relative to that offset.

(empty page)

Appendix C

Variable Length Data: Encoding/Decoding (Informative)

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

The encode and decode algorithms given here do not take account of C/C++ rules that mean that in `E1<<E2` the type of `E1` should be a sufficiently large unsigned type to hold the correct mathematical result. The decode algorithms do not take account of or protect from possibly invalid LEB values, such as values that are too large to fit in the target type or that lack a proper terminator byte. Implementation languages may have additional or different rules.

```
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 C.1: 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 second 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 C.2: 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 C.3: Algorithm to decode an unsigned LEB128 integer

```
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 second 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 C.4: Algorithm to decode a signed LEB128 integer

(empty page)

Appendix D

Examples (Informative)

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

D.1 General Description Examples

D.1.1 Compilation Units and Abbreviations Table Example

Figure D.1 on the next page 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.

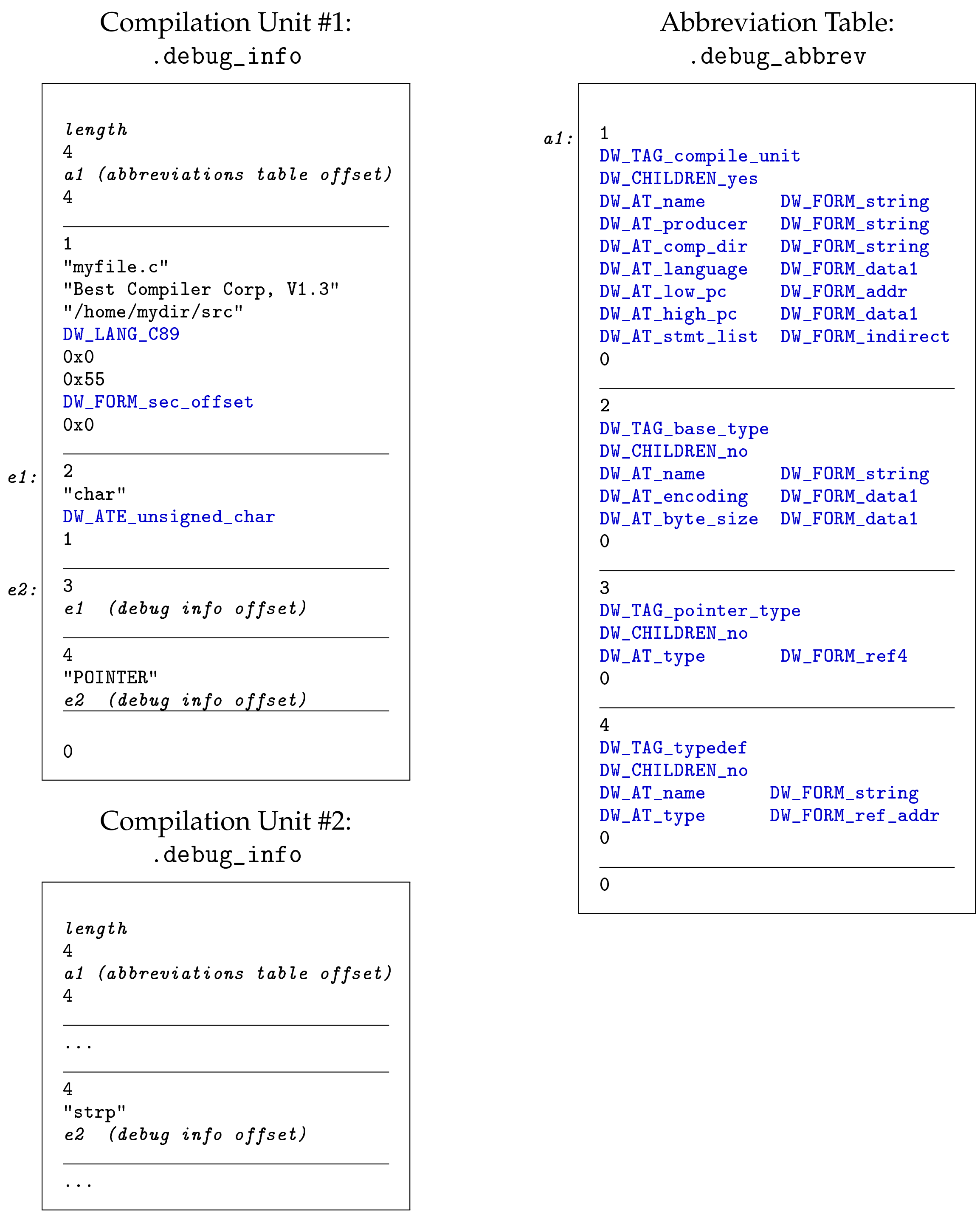
The figure corresponds to the following two trivial source files:

File myfile.c

```
typedef char* POINTER;
```

File myfile2.c

```
typedef char* strp;
```



Compilation Unit #1:
.debug_info

```
     length
     4
     a1 (abbreviations table offset)
     4
     ------------------------------
     1
     "myfile.c"
     "Best Compiler Corp, V1.3"
     "/home/mydir/src"
     DW_LANG_C89
     0x0
     0x55
     DW_FORM_sec_offset
     0x0
     ------------------------------
e1:  2
     "char"
     DW_ATE_unsigned_char
     1
     ------------------------------
e2:  3
     e1  (debug info offset)
     ------------------------------
     4
     "POINTER"
     e2  (debug info offset)
     ------------------------------
     0
```

Compilation Unit #2:
.debug_info

```
length
4
a1 (abbreviations table offset)
4
------------------------------
...
------------------------------
4
"strp"
e2  (debug info offset)
------------------------------
...
```

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_data1
     DW_AT_stmt_list  DW_FORM_indirect
     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
     ------------------------------
     3
     DW_TAG_pointer_type
     DW_CHILDREN_no
     DW_AT_type       DW_FORM_ref4
     0
     ------------------------------
     4
     DW_TAG_typedef
     DW_CHILDREN_no
     DW_AT_name      DW_FORM_string
     DW_AT_type      DW_FORM_ref_addr
     0
     ------------------------------
     0
```

Figure D.1: Compilation units and abbreviations table

D.1.2 DWARF Stack Operation Examples

The stack operations defined in Section 2.5.1.3 on page 29. 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

D.1.3 DWARF Location Description Examples

Following are examples of DWARF operations used to form location descriptions:

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 a DW_AT_frame_base value of "DW_OP_breg31 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.

DW_OP_reg0 DW_OP_piece 4 DW_OP_piece 4 DW_OP_fbreg -12 DW_OP_piece 4

> A twelve byte value whose first four bytes reside in register zero, whose middle four bytes are unavailable (perhaps due to optimization), and whose last four bytes are in memory, 12 bytes before the frame base.

DW_OP_breg1 0 DW_OP_breg2 0 DW_OP_plus DW_OP_stack_value

> Add the contents of r1 and r2 to compute a value. This value is the "contents" of an otherwise anonymous location.

DW_OP_lit1 DW_OP_stack_value DW_OP_piece 4 DW_OP_breg3 0 DW_OP_breg4 0 DW_OP_plus DW_OP_stack_value DW_OP_piece 4

> The object value is found in an anonymous (virtual) location whose value consists of two parts, given in memory address order: the 4 byte value 1 followed by the four byte value computed from the sum of the contents of r3 and r4.

DW_OP_entry_value 2 DW_OP_breg1 0

> The value register 1 contained upon entering the current subprogram is pushed on the stack.

DW_OP_entry_value 1 DW_OP_reg1

> Same as the previous example (push the value register 1 contained upon entering the current subprogram) but use the more compact register location description.

DW_OP_entry_value 2 DW_OP_breg1 0 DW_OP_stack_value

> The value register 1 contained upon entering the current subprogram is pushed on the stack. This value is the "contents" of an otherwise anonymous location.

DW_OP_entry_value 1 DW_OP_reg1 DW_OP_stack_value

> Same as the previous example (push the value register 1 contained upon entering the current subprogram) but use the more compact register location description.

DW_OP_entry_value 3 DW_OP_breg4 16 DW_OP_deref DW_OP_stack_value

> Add 16 to the value register 4 had upon entering the current subprogram to form an address and then push the value of the memory location at that address. This value is the "contents" of an otherwise anonymous location.

DW_OP_entry_value 1 DW_OP_reg5 DW_OP_plus_uconst 16

> The address of the memory location is calculated by adding 16 to the value contained in register 5 upon entering the current subprogram.
>
> *Note that unlike the previous DW_OP_entry_value examples, this one does not end with DW_OP_stack_value.*

DW_OP_reg0 DW_OP_bit_piece 1 31 DW_OP_bit_piece 7 0 DW_OP_reg1 DW_OP_piece 1

> A variable whose first bit resides in the 31st bit of register 0, whose next seven bits are undefined and whose second byte resides in register 1.

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 Simple Array Example

Consider the Fortran array source fragment in Figure D.2 following.

```
TYPE array_ptr
REAL :: myvar
REAL, DIMENSION (:), POINTER :: ap
END TYPE array_ptr
TYPE(array_ptr), ALLOCATABLE, DIMENSION(:) :: arrayvar
ALLOCATE(arrayvar(20))
DO I = 1, 20
    ALLOCATE(arrayvar(i)%ap(i+10))
END DO
```

Figure D.2: Fortran array example: source fragment

For allocatable and pointer arrays, it is essentially required by theFortran array 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 D.3. 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 D.3: Fortran array 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 attribute. (See the following Ada example for a case where the type for an object without a descriptor does have a DW_AT_data_location attribute. In that case the object doubles as its own descriptor.)

The Fortran derived type `array_ptr` can now be re-described in C-like terms that expose some of the representation as in

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

Similarly for variable `arrayvar`:

```
desc<1> arrayvar;
```

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 D.4 on page 296.

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

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

Interpretation of this expression proceeds as follows:

1. Lookup name `arrayvar`. 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 will skip that for this example.]

    c) the stride

For a), check for a DW_AT_data_location attribute. Since there is one, go execute the expression, whose result is the address needed. The object address used in this case is the object we are working on, namely the variable named `arrayvar`, 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 an expression, which again begins with DW_OP_push_object_address. This object is **still** `arrayvar`, from step 1, because we have not begun to actually perform any indexing yet.

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

part 1 of 2

```
! Description for type of 'ap'
!
1$: DW_TAG_array_type
        ! No name, default (Fortran) ordering, default stride
        DW_AT_type(reference to REAL)
        DW_AT_associated(expression=    ! Test 'ptr_assoc' flag
            DW_OP_push_object_address
            DW_OP_lit<n>                ! where n == offset(ptr_assoc)
            DW_OP_plus
            DW_OP_deref
            DW_OP_lit1                  ! mask for 'ptr_assoc' flag
            DW_OP_and)
        DW_AT_data_location(expression= ! Get raw data address
            DW_OP_push_object_address
            DW_OP_lit<n>                ! where n == offset(base)
            DW_OP_plus
            DW_OP_deref)                ! Type of index of array 'ap'
2$:     DW_TAG_subrange_type
            ! No name, default stride
            DW_AT_type(reference to INTEGER)
            DW_AT_lower_bound(expression=
                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(expression=
                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<n> where
            !       n == offset(desc, dims)          +
            !                (m-1)*sizeof(dims_str)  +
            !                 offset(dims_str, [lower|upper]_bound)
            !  That is, the expression does not get longer for each successive
            !  dimension (other than to express the larger offsets involved).
```

Figure D.4: Fortran array example: DWARF description

part 2 of 2

```
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 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$: DW_TAG_array_type
        ! No name, default (Fortran) ordering, default stride
        DW_AT_type(reference to 3$)
        DW_AT_allocated(expression=       ! 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(expression=   ! Get raw data address
            DW_OP_push_object_address
            DW_OP_lit<n>                  ! where n == offset(base)
            DW_OP_plus
            DW_OP_deref)
7$:     DW_TAG_subrange_type
            ! No name, default stride
            DW_AT_type(reference to INTEGER)
            DW_AT_lower_bound(expression=
                DW_OP_push_object_address
                DW_OP_lit<n>              ! where n == ...
                DW_OP_plus
                DW_OP_deref)
            DW_AT_upper_bound(expression=
                DW_OP_push_object_address
                DW_OP_lit<n>              ! where n == ...
                DW_OP_plus
                DW_OP_deref)
8$: DW_TAG_variable
        DW_AT_name("arrayvar")
        DW_AT_type(reference to 6$)
        DW_AT_location(expression=
            ...as appropriate...)       ! Assume static allocation
```

Figure D.4: Fortran array example: DWARF description *(concluded)*

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 `arrayvar`.

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 second 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 will skip that for this example]

   c) the stride

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 Fortran Coarray Examples

D.2.2.1 Fortran Scalar Coarray Example

The Fortran scalar coarray example in Figure D.5 on the next page can be described as illustrated in Figure D.6 on the following page.

```
        INTEGER x[*]
```

Figure D.5: Fortran scalar coarray: source fragment

```
10$:  DW_TAG_coarray_type
          DW_AT_type(reference to INTEGER)
          DW_TAG_subrange_type                ! Note omitted upper bound
          DW_AT_lower_bound(constant 1)       ! Can be omitted (default is 1)

11$:  DW_TAG_variable
          DW_AT_name("x")
          DW_AT_type(reference to coarray type at 10$)
```

Figure D.6: Fortran scalar coarray: DWARF description

D.2.2.2 Fortran Array Coarray Example

The Fortran (simple) array coarray example in Figure D.7 can be described as illustrated in Figure D.8.

```
        INTEGER x(10)[*]
```

Figure D.7: Fortran array coarray: source fragment

```
10$:  DW_TAG_array_type
          DW_AT_ordering(DW_ORD_col_major)
          DW_AT_type(reference to INTEGER)
11$:      DW_TAG_subrange_type
            ! DW_AT_lower_bound(constant 1)    ! Omitted (default is 1)
              DW_AT_upper_bound(constant 10)

12$:  DW_TAG_coarray_type
          DW_AT_type(reference to array type at 10$)
13$:      DW_TAG_subrange_type                ! Note omitted upper & lower bounds

14$:  DW_TAG_variable
          DW_AT_name("x")
          DW_AT_type(reference to coarray type at 12$)
```

Figure D.8: Fortran array coarray: DWARF description

D.2.2.3 Fortran Multidimensional Coarray Example

The Fortran multidimensional coarray of a multidimensional array example in Figure D.9 can be described as illustrated in Figure D.10 following.

```
        INTEGER x(10,11,12)[2,3,*]
```

Figure D.9: Fortran multidimensional coarray: source fragment

```
10$:  DW_TAG_array_type              ! Note omitted lower bounds (default to 1)
          DW_AT_ordering(DW_ORD_col_major)
          DW_AT_type(reference to INTEGER)
11$:      DW_TAG_subrange_type
              DW_AT_upper_bound(constant 10)
12$:      DW_TAG_subrange_type
              DW_AT_upper_bound(constant 11)
13$:      DW_TAG_subrange_type
              DW_AT_upper_bound(constant 12)

14$:  DW_TAG_coarray_type            ! Note omitted lower bounds (default to 1)
          DW_AT_type(reference to array_type at 10$)
15$:      DW_TAG_subrange_type
              DW_AT_upper_bound(constant 2)
16$:      DW_TAG_subrange_type
              DW_AT_upper_bound(constant 3)
17$:      DW_TAG_subrange_type       ! Note omitted upper (& lower) bound

18$:  DW_TAG_variable
          DW_AT_name("x")
          DW_AT_type(reference to coarray type at 14$)
```

Figure D.10: Fortran multidimensional coarray: DWARF description

D.2.3 Fortran 2008 Assumed-rank Array Example

Consider the example in Figure D.11, which shows an assumed-rank array in Fortran 2008 with supplement 29113:[1]

```
SUBROUTINE Foo(x)
  REAL :: x(..)

  ! x has n dimensions

END SUBROUTINE
```

Figure D.11: Declaration of a Fortran 2008 assumed-rank array

Let's assume the Fortran compiler used an array descriptor that (in C) looks like the one shown in Figure D.12.

```
struct array_descriptor {
  void *base_addr;
  int rank;
  struct dim dims[];
}

struct dim {
   int lower_bound;
   int upper_bound;
   int stride;
   int flags;
}
```

Figure D.12: One of many possible layouts for an array descriptor

The DWARF type for the array *x* can be described as shown in Figure D.13 on the following page.

The layout of the array descriptor is not specified by the Fortran standard unless the array is explicitly marked as C-interoperable. To get the bounds of an assumed-rank array, the expressions in the DW_TAG_generic_subrange entry need to be evaluated for each of the DW_AT_rank dimensions as shown by the pseudocode in Figure D.14 on page 303.

[1]Technical Specification ISO/IEC TS 29113:2012 *Further Interoperability of Fortran with C*

```
10$:  DW_TAG_array_type
          DW_AT_type(reference to real)
          DW_AT_rank(expression=
              DW_OP_push_object_address
              DW_OP_lit<n>                     ! offset of rank in descriptor
              DW_OP_plus
              DW_OP_deref)
          DW_AT_data_location(expression=
              DW_OP_push_object_address
              DW_OP_lit<n>                     ! offset of data in descriptor
              DW_OP_plus
              DW_OP_deref)
11$:     DW_TAG_generic_subrange
              DW_AT_type(reference to integer)
              DW_AT_lower_bound(expression=
              !   Looks up the lower bound of dimension i.
              !   Operation                    ! Stack effect
              !   (implicit)                   ! i
                  DW_OP_lit<n>                 ! i sizeof(dim)
                  DW_OP_mul                    ! dim[i]
                  DW_OP_lit<n>                 ! dim[i] offsetof(dim)
                  DW_OP_plus                   ! dim[i]+offset
                  DW_OP_push_object_address    ! dim[i]+offsetof(dim) objptr
                  DW_OP_plus                   ! objptr.dim[i]
                  DW_OP_lit<n>                 ! objptr.dim[i] offsetof(lb)
                  DW_OP_plus                   ! objptr.dim[i].lowerbound
                  DW_OP_deref)                 ! *objptr.dim[i].lowerbound
              DW_AT_upper_bound(expression=
              !   Looks up the upper bound of dimension i.
                  DW_OP_lit<n>                 ! sizeof(dim)
                  DW_OP_mul
                  DW_OP_lit<n>                 ! offsetof(dim)
                  DW_OP_plus
                  DW_OP_push_object_address
                  DW_OP_plus
                  DW_OP_lit<n>                 ! offset of upperbound in dim
                  DW_OP_plus
                  DW_OP_deref)
              DW_AT_byte_stride(expression=
              !   Looks up the byte stride of dimension i.
                  ...
              !   (analogous to DW_AT_upper_bound)
                  )
```

Figure D.13: Sample DWARF for the array descriptor in Figure D.12

```
typedef struct {
    int lower, upper, stride;
} dims_t;

typedef struct {
    int rank;
struct dims_t *dims;
} array_t;

array_t get_dynamic_array_dims(DW_TAG_array a) {
  array_t result;

  // Evaluate the DW_AT_rank expression to get the
  //    number of dimensions.
  dwarf_stack_t stack;
  dwarf_eval(stack, a.rank_expr);
  result.rank = dwarf_pop(stack);
  result.dims = new dims_t[rank];

  // Iterate over all dimensions and find their bounds.
  for (int i = 0; i < result.rank; i++) {
    // Evaluate the generic subrange's DW_AT_lower
    //    expression for dimension i.
    dwarf_push(stack, i);
    assert( stack.size == 1 );
    dwarf_eval(stack, a.generic_subrange.lower_expr);
    result.dims[i].lower = dwarf_pop(stack);
    assert( stack.size == 0 );

    dwarf_push(stack, i);
    dwarf_eval(stack, a.generic_subrange.upper_expr);
    result.dims[i].upper = dwarf_pop(stack);

    dwarf_push(stack, i);
    dwarf_eval(stack, a.generic_subrange.byte_stride_expr);
    result.dims[i].stride = dwarf_pop(stack);
  }
  return result;
}
```

Figure D.14: How to interpret the DWARF from Figure D.13

D.2.4 Fortran Dynamic Type Example

Consider the Fortran 90 example of dynamic properties in Figure D.15. This can be represented in DWARF as illustrated in Figure D.16 on the next page. Note that unnamed dynamic types are used to avoid replicating the full description of the underlying type `dt` that is shared by several variables.

```
PROGRAM Sample

      TYPE :: dt (l)
          INTEGER, LEN :: l
          INTEGER :: arr(l)
      END TYPE

      INTEGER :: n = 4
      CONTAINS

      SUBROUTINE S()
          TYPE (dt(n))                :: t1
          TYPE (dt(n)), pointer       :: t2
          TYPE (dt(n)), allocatable   :: t3, t4
      END SUBROUTINE

      END Sample
```

Figure D.15: Fortran dynamic type example: source

```
11$:    DW_TAG_structure_type
            DW_AT_name("dt")
            DW_TAG_member
                ...
...

13$:    DW_TAG_dynamic_type             ! plain version
            DW_AT_data_location (dwarf expression to locate raw data)
            DW_AT_type (11$)

14$:    DW_TAG_dynamic_type             ! 'pointer' version
            DW_AT_data_location (dwarf expression to locate raw data)
            DW_AT_associated (dwarf expression to test if associated)
            DW_AT_type (11$)

15$:    DW_TAG_dynamic_type             ! 'allocatable' version
            DW_AT_data_location (dwarf expression to locate raw data)
            DW_AT_allocated (dwarf expression to test is allocated)
            DW_AT_type (11$)

16$:    DW_TAG_variable
            DW_AT_name ("t1")
            DW_AT_type (13$)
            DW_AT_location (dwarf expression to locate descriptor)
17$:    DW_TAG_variable
            DW_AT_name ("t2")
            DW_AT_type (14$)
            DW_AT_location (dwarf expression to locate descriptor)
18$:    DW_TAG_variable
            DW_AT_name ("t3")
            DW_AT_type (15$)
            DW_AT_location (dwarf expression to locate descriptor)
19$:    DW_TAG_variable
            DW_AT_name ("t4")
            DW_AT_type (15$)
            DW_AT_location (dwarf expression to locate descriptor)
```

Figure D.16: Fortran dynamic type example: DWARF description

D.2.5 C/C++ Anonymous Structure Example

An example of a C/C++ structure is shown in Figure D.17. For this source, the DWARF description in Figure D.18 is appropriate. In this example, `b` is referenced as if it were defined in the enclosing structure `foo`.

```
struct foo {
    int a;
    struct {
        int b;
    };
} x;

void bar(void)
{
    struct foo t;
    t.a = 1;
    t.b = 2;
}
```

Figure D.17: Anonymous structure example: source fragment

```
1$:  DW_TAG_structure_type
         DW_AT_name("foo")
2$:      DW_TAG_member
             DW_AT_name("a")
3$:      DW_TAG_structure_type
             DW_AT_export_symbols
4$:          DW_TAG_member
                 DW_AT_name("b")
```

Figure D.18: Anonymous structure example: DWARF description

D.2.6 Ada Example

Figure D.19 on the following page illustrates two kinds of Ada parameterized array, one embedded in a record.

`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).

```
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 D.19: Ada example: source fragment

`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 D.20 on the next page.

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 described as a reference to a member, and the location of the upper bound is determined as for any member. That is, the location is determined using an address calculation relative to the base of the containing object.

   A consumer must notice that the referenced bound is a member of the same containing object and implicitly push the base address of the containing object just as for accessing a data member generally.

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

```
11$:  DW_TAG_variable
          DW_AT_name("M")
          DW_AT_type(reference to INTEGER)
12$:  DW_TAG_array_type
          ! No name, default (Ada) order, default stride
          DW_AT_type(reference to INTEGER)
13$:      DW_TAG_subrange_type
              DW_AT_type(reference to INTEGER)
              DW_AT_lower_bound(constant 1)
              DW_AT_upper_bound(reference to variable M at 11$)
14$:  DW_TAG_variable
          DW_AT_name("VEC1")
          DW_AT_type(reference to array type at 12$)
      . . .
21$:  DW_TAG_subrange_type
          DW_AT_name("TEENY")
          DW_AT_type(reference to INTEGER)
          DW_AT_lower_bound(constant 1)
          DW_AT_upper_bound(constant 100)
      . . .
26$:  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$:      DW_TAG_array_type
              ! No name, default (Ada) order, default stride
              ! Default data location
              DW_AT_type(reference to INTEGER)
29$:          DW_TAG_subrange_type
                  DW_AT_type(reference to subrange TEENY at 21$)
                  DW_AT_lower_bound(constant 1)
                  DW_AT_upper_bound(reference to member N at 27$)
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$:  DW_TAG_variable
          DW_AT_name("OBJ2B")
          DW_AT_type(reference to REC2 at 26$)
          DW_AT_location(...as appropriate...)
```

Figure D.20: Ada example: DWARF description

D.2.7 Pascal Example

The Pascal source in Figure D.21 following is used to illustrate the representation of packed unaligned bit fields.

```
TYPE T : PACKED RECORD                    { bit size is 2   }
         F5 : BOOLEAN;                    { bit offset is 0 }
         F6 : BOOLEAN;                    { bit offset is 1 }
         END;
VAR V :  PACKED RECORD
         F1 : BOOLEAN;                    { bit offset is 0 }
         F2 : PACKED RECORD               { bit offset is 1 }
              F3 : INTEGER;               { bit offset is 0 in F2,
                                            1 in V }
              END;
         F4 : PACKED ARRAY [0..1] OF T;   { bit offset is 33 }
         F7 : T;                          { bit offset is 37 }
         END;
```

Figure D.21: Packed record example: source fragment

The DWARF representation in Figure D.22 is appropriate. DW_TAG_packed_type entries could be added to better represent the source, but these do not otherwise affect the example and are omitted for clarity. Note that this same representation applies to both typical big- and little-endian architectures using the conventions described in Section 5.7.6 on page 118.

part 1 of 2

```
10$:  DW_TAG_base_type
          DW_AT_name("BOOLEAN")
              ...
11$:  DW_TAG_base_type
          DW_AT_name("INTEGER")
              ...
20$:  DW_TAG_structure_type
          DW_AT_name("T")
          DW_AT_bit_size(2)
          DW_TAG_member
              DW_AT_name("F5")
              DW_AT_type(reference to 10$)
              DW_AT_data_bit_offset(0)        ! may be omitted
              DW_AT_bit_size(1)
```

Figure D.22: Packed record example: DWARF description

part 2 of 2

```
          DW_TAG_member
              DW_AT_name("F6")
              DW_AT_type(reference to 10$)
              DW_AT_data_bit_offset(1)
              DW_AT_bit_size(1)
21$:  DW_TAG_structure_type                 ! anonymous type for F2
          DW_TAG_member
              DW_AT_name("F3")
              DW_AT_type(reference to 11$)
22$:  DW_TAG_array_type                     ! anonymous type for F4
          DW_AT_type(reference to 20$)
          DW_TAG_subrange_type
              DW_AT_type(reference to 11$)
              DW_AT_lower_bound(0)
              DW_AT_upper_bound(1)
          DW_AT_bit_stride(2)
          DW_AT_bit_size(4)
23$:  DW_TAG_structure_type                 ! anonymous type for V
          DW_AT_bit_size(39)
          DW_TAG_member
              DW_AT_name("F1")
              DW_AT_type(reference to 10$)
              DW_AT_data_bit_offset(0)        ! may be omitted
              DW_AT_bit_size(1) ! may be omitted
          DW_TAG_member
              DW_AT_name("F2")
              DW_AT_type(reference to 21$)
              DW_AT_data_bit_offset(1)
              DW_AT_bit_size(32) ! may be omitted
          DW_TAG_member
              DW_AT_name("F4")
              DW_AT_type(reference to 22$)
              DW_AT_data_bit_offset(33)
              DW_AT_bit_size(4) ! may be omitted
          DW_TAG_member
              DW_AT_name("F7")
              DW_AT_type(reference to 20$)    ! type T
              DW_AT_data_bit_offset(37)
              DW_AT_bit_size(2)               ! may be omitted
       DW_TAG_variable
          DW_AT_name("V")
          DW_AT_type(reference to 23$)
          DW_AT_location(...)
          ...
```

Figure D.22: Packed record example: DWARF description *(concluded)*

D.2.8 C/C++ Bit-Field Examples

Bit fields in C and C++ typically require the use of the DW_AT_data_bit_offset and DW_AT_bit_size attributes.

This Standard uses the following bit numbering and direction conventions in examples. These conventions are for illustrative purposes and other conventions may apply on particular architectures.

- *For big-endian architectures, bit offsets are counted from high-order to low-order bits within a byte (or larger storage unit); in this case, the bit offset identifies the high-order bit of the object.*
- *For little-endian architectures, bit offsets are counted from low-order to high-order bits within a byte (or larger storage unit); in this case, the bit offset identifies the low-order bit of the object.*

In either case, the bit so identified is defined as the beginning of the object.

This section illustrates one possible representation of the following C structure definition in both big- and little-endian byte orders:

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

Figures D.23 and D.24 on the following page show the structure layout and data bit offsets for example big- and little-endian architectures, respectively. Both diagrams show a structure that begins at address A and whose size is four bytes. Also, high order bits are to the left and low order bits are to the right.

Note that data member bit offsets in this example are the same for both big- and little-endian architectures even though the fields are allocated in different directions (high-order to low-order versus low-order to high-order); the bit naming conventions for memory and/or registers of the target architecture may or may not make this seem natural.

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

Addresses increase ->
|       A       |     A + 1     |    A + 2      |    A + 3      |

Data bit offsets increase ->
+---------------+---------------+---------------+---------------+
|0     4|5         10|11      15|16           23|24           31|
|   j   |     k      | m        |        n      |       <pad>   |
|       |            |          |               |               |
+---------------------------------------------------------------+
```

Figure D.23: Big-endian data bit offsets

```
j:0
k:5
m:11
n:16
                                           <- Addresses increase
|     A + 3     |     A + 2     |    A + 1      |       A       |

                                    <-  Data bit offsets increase
+---------------+---------------+---------------+---------------+
|31           24|23           16|15     11|10       5|4        0|
|     <pad>     |        n      |    m    |    k     |     j    |
|               |               |         |          |          |
+---------------------------------------------------------------+
```

Figure D.24: Little-endian data bit offsets

D.3 Namespace Examples

The C++ example in Figure D.25 is used to illustrate the representation of namespaces. The DWARF representation in Figure D.26 on the following page is appropriate.

```
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 D.25: Namespace example #1: source fragment

part 1 of 2

```
1$:   DW_TAG_base_type
          DW_AT_name("int")
          ...
2$:   DW_TAG_base_type
          DW_AT_name("float")
          ...
6$:   DW_TAG_namespace
          ! no DW_AT_name attribute
          DW_AT_export_symbols              ! Implied by C++, but can be explicit
          DW_TAG_variable
              DW_AT_name("i")
              DW_AT_type(reference to 1$)
              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$:          DW_TAG_subprogram
                  DW_AT_name("myfunc2")
                  DW_AT_low_pc ...
                  DW_AT_high_pc ...
                  DW_AT_type(reference to 1$)
                  ...
```

Figure D.26: Namespace example #1: DWARF description

part 2 of 2

```
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$:  DW_TAG_subprogram
          DW_AT_specification(reference to 34$)
          DW_AT_low_pc ...
          DW_AT_high_pc ...
          ...
```

Figure D.26: Namespace example #1: DWARF description *(concluded)*

As a further namespace example, consider the inlined namespace shown in Figure D.27. For this source, the DWARF description in Figure D.28 is appropriate. In this example, a may be referenced either as a member of the fully qualified namespace A::B, or as if it were defined in the enclosing namespace, A.

```
namespace A {
    inline namespace B {   // (1) inline namespace
        int a;
    }
}

void foo (void)
{
    using A::B::a;
    a = 1;
}

void bar (void)
{
    using A::a;
    a = 2;
}
```

Figure D.27: Namespace example #2: source fragment

```
1$:   DW_TAG_namespace
          DW_AT_name("A")
2$:       DW_TAG_namespace
              DW_AT_name("B")
              DW_AT_export_symbols
3$:           DW_TAG_variable
                  DW_AT_name("a")
```

Figure D.28: Namespace example #2: DWARF description

D.4 Member Function Examples

Consider the member function example fragment in Figure D.29. The DWARF representation in Figure D.30 is appropriate.

```
class A
{
    void func1(int x1);
    void func2() const;
    static void func3(int x3);
};
void A::func1(int x) {}
```

Figure D.29: Member function example: source fragment

part 1 of 2

```
2$: DW_TAG_base_type
        DW_AT_name("int")
        ...
3$: DW_TAG_class_type
        DW_AT_name("A")
        ...
4$:     DW_TAG_pointer_type
            DW_AT_type(reference to 3$)
            ...
5$:     DW_TAG_const_type
            DW_AT_type(reference to 3$)
            ...
6$:     DW_TAG_pointer_type
            DW_AT_type(reference to 5$)
            ...

7$:     DW_TAG_subprogram
            DW_AT_declaration
            DW_AT_name("func1")
            DW_AT_object_pointer(reference to 8$)
                ! References a formal parameter in this
                ! member function
            ...
```

Figure D.30: Member function example: DWARF description

part 2 of 2

```
8$:         DW_TAG_formal_parameter
                DW_AT_artificial(true)
                DW_AT_name("this")
                DW_AT_type(reference to 4$)
                    ! Makes type of 'this' as 'A*' =>
                    ! func1 has not been marked const
                    ! or volatile
                DW_AT_location ...
                ...
9$:         DW_TAG_formal_parameter
                DW_AT_name(x1)
                DW_AT_type(reference to 2$)
                ...
10$:    DW_TAG_subprogram
            DW_AT_declaration
            DW_AT_name("func2")
            DW_AT_object_pointer(reference to 11$)
            ! References a formal parameter in this
            ! member function
            ...
11$:        DW_TAG_formal_parameter
                DW_AT_artificial(true)
                DW_AT_name("this")
                DW_AT_type(reference to 6$)
                ! Makes type of 'this' as 'A const*' =>
                !     func2 marked as const
                DW_AT_location ...
                ...
12$:    DW_TAG_subprogram
            DW_AT_declaration
            DW_AT_name("func3")
            ...
                ! No object pointer reference formal parameter
                ! implies func3 is static
13$:        DW_TAG_formal_parameter
                DW_AT_name(x3)
                DW_AT_type(reference to 2$)
                ...
```

Figure D.30: Member function example: DWARF description *(concluded)*

As a further example illustrating &- and &&-qualification of member functions, consider the member function example fragment in Figure D.31. The DWARF representation in Figure D.32 on the next page is appropriate.

```
class A {
public:
    void f() const &&;
};

void g() {
    A a;
    // The type of pointer is "void (A::*)() const &&".
    auto pointer_to_member_function = &A::f;
}
```

Figure D.31: Reference- and rvalue-reference-qualification example: source fragment

```
100$:   DW_TAG_class_type
            DW_AT_name("A")
            DW_TAG_subprogram
                DW_AT_name("f")
                DW_AT_rvalue_reference(0x01)
                DW_TAG_formal_parameter
                    DW_AT_type(ref to 200$)     ! to const A*
                    DW_AT_artificial(0x01)

200$:   ! const A*
        DW_TAG_pointer_type
            DW_AT_type(ref to 300$)             ! to const A

300$:   ! const A
        DW_TAG_const_type
            DW_AT_type(ref to 100$)             ! to class A

400$:   ! mfptr
        DW_TAG_ptr_to_member_type
            DW_AT_type(ref to 500$)             ! to functype
            DW_AT_containing_type(ref to 100$)  ! to class A

500$:   ! functype
        DW_TAG_subroutine_type
            DW_AT_rvalue_reference(0x01)
            DW_TAG_formal_parameter
                DW_AT_type(ref to 200$)         ! to const A*
                DW_AT_artificial(0x01)

600$:   DW_TAG_subprogram
            DW_AT_name("g")
            DW_TAG_variable
                DW_AT_name("a")
                DW_AT_type(ref to 100$)         ! to class A
            DW_TAG_variable
                DW_AT_name("pointer_to_member_function")
                DW_AT_type(ref to 400$)
```

Figure D.32: Reference- and rvalue-reference-qualification example: DWARF description

D.5 Line Number Examples

D.5.1 Line Number Header Example

The information found in a DWARF Version 4 line number header can be encoded in a DWARF Version 5 header as shown in Figure D.33.

Field Number	Field Name	Value(s)
1	*Same as in Version 4*	...
2	version	5
3	*Not present in Version 4*	-
4	*Not present in Version 4*	-
5-12	*Same as in Version 4*	...
13	directory_entry_format_count	1
14	directory_entry_format	DW_LNCT_path, DW_FORM_string
15	directories_count	<n>
16	directories	<n>*<null terminated string>
17	file_name_entry_format_count	4
18	file_name_entry_format	DW_LNCT_path, DW_FORM_string, DW_LNCT_directory_index, DW_FORM_udata, DW_LNCT_timestamp, DW_FORM_udata, DW_LNCT_size, DW_FORM_udata
19	file_names_count	<m>
20	file_names	<m>*{<null terminated string>, <index>, <timestamp>, <size>}

Figure D.33: Pre-DWARF Version 5 line number program header information encoded using DWARF Version 5

D.5.2 Line Number Special Opcode Example

Suppose the line number header includes the following (header fields not needed are not shown):

opcode_base	13
line_base	-3
line_range	12
minimum_instruction_length	1
maximum_operations_per_instruction	1

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 [-3, 8] and (because of the limited number of opcodes available) when the difference between addresses is within the range [0, 20]. The resulting opcode mapping is shown in Figure D.34.

Note in the bottom row of the figure that not all line advances are available for the maximum operation advance.

Operation Advance	Line Advance											
	-3	-2	-1	0	1	2	3	4	5	6	7	8
0	13	14	15	16	17	18	19	20	21	22	23	24
1	25	26	27	28	29	30	31	32	33	34	35	36
2	37	38	39	40	41	42	43	44	45	46	47	48
3	49	50	51	52	53	54	55	56	57	58	59	60
4	61	62	63	64	65	66	67	68	69	70	71	72
5	73	74	75	76	77	78	79	80	81	82	83	84
6	85	86	87	88	89	90	91	92	93	94	95	96
7	97	98	99	100	101	102	103	104	105	106	107	108
8	109	110	111	112	113	114	115	116	117	118	119	120
9	121	122	123	124	125	126	127	128	129	130	131	132
10	133	134	135	136	137	138	139	140	141	142	143	144
11	145	146	147	148	149	150	151	152	153	154	155	156
12	157	158	159	160	161	162	163	164	165	166	167	168
13	169	170	171	172	173	174	175	176	177	178	179	180
14	181	182	183	184	185	186	187	188	189	190	191	192
15	193	194	195	196	197	198	199	200	201	202	203	204
16	205	206	207	208	209	210	211	212	213	214	215	216
17	217	218	219	220	221	222	223	224	225	226	227	228
18	229	230	231	232	233	234	235	236	237	238	239	240
19	241	242	243	244	245	246	247	248	249	250	251	252
20	253	254	255									

Figure D.34: Example line number special opcode mapping

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

D.5.3 Line Number Program Example

Consider the simple source file and the resulting machine code for the Intel 8086 processor in Figure D.35.

```
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 D.35: Line number program example: machine code

Suppose the line number program header includes the same values and resulting encoding illustrated in the previous Section D.5.2 on page 321.

Table D.2 on the following page shows one encoding of the line number program, which occupies 12 bytes.

Table D.2: Line number program example: one encoding

Opcode	Operand	Byte Stream
DW_LNS_advance_pc	LEB128(0x239)	0x2, 0xb9, 0x04
SPECIAL† (2, 0)		0x12 (18_{10})
SPECIAL† (2, 3)		0x36 (54_{10})
SPECIAL† (1, 8)		0x71 (113_{10})
SPECIAL† (1, 7)		0x65 (101_{10})
DW_LNS_advance_pc	LEB128(2)	0x2, 0x2
DW_LNE_end_sequence		0x0, 0x1, 0x1

† The opcode notation SPECIAL(*m*,*n*) indicates the special opcode generated for a line advance of *m* and an operation advance of *n*)

Table D.3 shows an alternate encoding of the same program using standard opcodes to advance the program counter; this encoding occupies 22 bytes.

Table D.3: Line number program example: alternate encoding

Opcode	Operand	Byte Stream
DW_LNS_fixed_advance_pc	0x239	0x9, 0x39, 0x2
SPECIAL‡ (2, 0)		0x12 (18_{10})
DW_LNS_fixed_advance_pc	0x3	0x9, 0x3, 0x0
SPECIAL‡ (2, 0)		0x12 (18_{10})
DW_LNS_fixed_advance_pc	0x8	0x9, 0x8, 0x0
SPECIAL‡ (1, 0)		0x11 (17_{10})
DW_LNS_fixed_advance_pc	0x7	0x9, 0x7, 0x0
SPECIAL‡ (1, 0)		0x11 (17_{10})
DW_LNS_fixed_advance_pc	0x2	0x9, 0x2, 0x0
DW_LNE_end_sequence		0x0, 0x1, 0x1

‡ SPECIAL is defined the same as in the preceding Table D.2.

D.6 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 eight 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 architectural ABI committee specifies that the stack pointer (R7) is the same as the CFA

Figure D.36 following shows 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.

An abstract table (see Section 6.4.1 on page 172) for the foo subroutine is shown in Table D.4 following. Corresponding fragments from the `.debug_frame` section are shown in Table D.5 on page 327.

The following notations apply in Table D.4 on the following page:

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
6. a = architectural rule

```
       ;; 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 D.36: Call frame information example: machine code fragments

Table D.4: Call frame information example: conceptual matrix

Location	**CFA**	**R0**	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**
foo	[R7]+0	s	u	u	u	s	s	s	a	r1
foo+4	[R7]+fs	s	u	u	u	s	s	s	a	r1
foo+8	[R7]+fs	s	u	u	u	s	s	s	a	c-4
foo+12	[R7]+fs	s	u	u	u	s	s	c-8	a	c-4
foo+16	[R6]+fs	s	u	u	u	s	s	c-8	a	c-4
foo+20	[R6]+fs	s	u	u	u	c-12	s	c-8	a	c-4
...										
foo+64	[R6]+fs	s	u	u	u	c-12	s	c-8	a	c-4
foo+68	[R6]+fs	s	u	u	u	s	s	c-8	a	c-4
foo+72	[R7]+fs	s	u	u	u	s	s	s	a	c-4
foo+76	[R7]+fs	s	u	u	u	s	s	s	a	r1
foo+80	[R7]+0	s	u	u	u	s	s	s	a	r1

Table D.5: Call frame information example: common information entry encoding

Address	Value	Comment
cie	36	length
cie+4	`0xffffffff`	CIE_id
cie+8	4	version
cie+9	0	augmentation
cie+10	4	address size
cie+11	0	segment size
cie+12	4	code_alignment_factor, <caf >
cie+13	-4	data_alignment_factor, <daf >
cie+14	8	R8 is the return addr.
cie+15	DW_CFA_def_cfa (7, 0)	CFA = [R7]+0
cie+18	DW_CFA_same_value (0)	R0 not modified (=0)
cie+20	DW_CFA_undefined (1)	R1 scratch
cie+22	DW_CFA_undefined (2)	R2 scratch
cie+24	DW_CFA_undefined (3)	R3 scratch
cie+26	DW_CFA_same_value (4)	R4 preserve
cie+28	DW_CFA_same_value (5)	R5 preserve
cie+30	DW_CFA_same_value (6)	R6 preserve
cie+32	DW_CFA_same_value (7)	R7 preserve
cie+34	DW_CFA_register (8, 1)	R8 is in R1
cie+37	DW_CFA_nop	padding
cie+38	DW_CFA_nop	padding
cie+39	DW_CFA_nop	padding
cie+40		

Table D.6: Call frame information example: frame description 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(12)	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		

†The following notations apply: `<fs>` = frame size, `<caf>` = code alignment factor, and `<daf>` = data alignment factor.

D.7 Inlining Examples

The pseudo-source in Figure D.37 following 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 D.37: Inlining examples: pseudo-source fragmment

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 `INNER`s
   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.7.1 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 D.38 on the next page.

Notice in Figure D.38 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 D.37 on the preceding page might be described as shown in Figure D.39 on page 332.

D.7.2 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 inlined instance of `OUTER` has its own normal instance of `INNER`. The abstract instance for `OUTER`, which includes `INNER`, is shown in Figure D.40 on page 334.

Note that the debugging information in Figure D.40 differs from that in Figure D.38 on the next page 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`.

```
    ! 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 D.38: 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 D.39: Inlining example #1: concrete instance

A resulting concrete inlined instance of OUTER is shown in Figure D.41 on page 336.

Notice in Figure D.41 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.

D.7.3 Alternative #3: inline OUTER, one normal INNER

In the third approach, one normal subprogram for INNER is compiled which is called from all concrete inlined instances of OUTER. The abstract instance for OUTER is shown in Figure D.42 on page 337.

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 inlined instance of OUTER is illustrated in Figure D.43 on page 338.

Notice in Figure D.43 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 inlined instances. A compiler might implement this in several ways, including the use of additional compiler-generated parameters that provide reference parameters for the

```
    ! Abstract instance for OUTER
    ! abstract instance
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 D.40: Inlining example #2: abstract instance

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.)

```
    ! 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 D.41: Inlining example #2: concrete instance

```
    ! 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 D.42: 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 D.43: Inlining example #3: concrete instance

D.8 Constant Expression Example

C++ generalizes the notion of constant expressions to include constant expression user-defined literals and functions. The constant declarations in Figure D.44 can be represented as illustrated in Figure D.45 on the following page.

```
constexpr double mass = 9.8;
constexpr int square (int x) { return x * x; }
float arr[square(9)]; // square() called and inlined
```

Figure D.44: Constant expressions: C++ source

```
        ! For variable mass
        !
1$:     DW_TAG_const_type
            DW_AT_type(reference to "double")
2$:     DW_TAG_variable
            DW_AT_name("mass")
            DW_AT_type(reference to 1$)
            DW_AT_const_expr(true)
            DW_AT_const_value(9.8)
        ! Abstract instance for square
        !
10$:    DW_TAG_subprogram
            DW_AT_name("square")
            DW_AT_type(reference to "int")
            DW_AT_inline(DW_INL_inlined)
11$:        DW_TAG_formal_parameter
                DW_AT_name("x")
                DW_AT_type(reference to "int")
        ! Concrete instance for square(9)
        !
20$:    DW_TAG_inlined_subroutine
            DW_AT_abstract_origin(reference to 10$)
            DW_AT_const_expr(present)
            DW_AT_const_value(81)
            DW_TAG_formal_parameter
                DW_AT_abstract_origin(reference to 11$)
                DW_AT_const_value(9)
        ! Anonymous array type for arr
        !
30$:    DW_TAG_array_type
            DW_AT_type(reference to "float")
            DW_AT_byte_size(324) ! 81*4
            DW_TAG_subrange_type
                DW_AT_type(reference to "int")
                DW_AT_upper_bound(reference to 20$)
        ! Variable arr
        !
40$:    DW_TAG_variable
            DW_AT_name("arr")
            DW_AT_type(reference to 30$)
```

Figure D.45: Constant expressions: DWARF description

D.9 Unicode Character Example

The Unicode character encodings in Figure D.46 can be described in DWARF as illustrated in Figure D.47.

```
// C++ source
//
char16_t chr_a = u'h';
char32_t chr_b = U'h';
```

Figure D.46: Unicode character example: source

```
! DWARF description
!
1$: DW_TAG_base_type
        DW_AT_name("char16_t")
        DW_AT_encoding(DW_ATE_UTF)
        DW_AT_byte_size(2)
2$: DW_TAG_base_type
        DW_AT_name("char32_t")
        DW_AT_encoding(DW_ATE_UTF)
        DW_AT_byte_size(4)
3$: DW_TAG_variable
        DW_AT_name("chr_a")
        DW_AT_type(reference to 1$)
4$: DW_TAG_variable
        DW_AT_name("chr_b")
        DW_AT_type(reference to 2$)
```

Figure D.47: Unicode character example: DWARF description

D.10 Type-Safe Enumeration Example

The C++ type-safe enumerations in Figure D.48 can be described in DWARF as illustrated in Figure D.49.

```
// C++ source
//
enum class E { E1, E2=100 };
E e1;
```

Figure D.48: Type-safe enumeration example: source

```
! DWARF description
!
11$:  DW_TAG_enumeration_type
          DW_AT_name("E")
          DW_AT_type(reference to "int")
          DW_AT_enum_class(present)
12$:      DW_TAG_enumerator
              DW_AT_name("E1")
              DW_AT_const_value(0)
13$:      DW_TAG_enumerator
              DW_AT_name("E2")
              DW_AT_const_value(100)
14$:  DW_TAG_variable
          DW_AT_name("e1")
          DW_AT_type(reference to 11$)
```

Figure D.49: Type-safe enumeration example: DWARF description

D.11 Template Examples

The C++ template example in Figure D.50 can be described in DWARF as illustrated in Figure D.51.

```
// C++ source
//
template<class T>
struct wrapper {
    T comp;
};
wrapper<int> obj;
```

Figure D.50: C++ template example #1: source

```
! DWARF description
!
11$:  DW_TAG_structure_type
          DW_AT_name("wrapper")
12$:      DW_TAG_template_type_parameter
              DW_AT_name("T")
              DW_AT_type(reference to "int")
13$:      DW_TAG_member
              DW_AT_name("comp")
              DW_AT_type(reference to 12$)
14$:  DW_TAG_variable
          DW_AT_name("obj")
          DW_AT_type(reference to 11$)
```

Figure D.51: C++ template example #1: DWARF description

The actual type of the component `comp` is `int`, but in the DWARF the type references the DW_TAG_template_type_parameter for `T`, which in turn references `int`. This implies that in the original template comp was of type `T` and that was replaced with `int` in the instance.

```
// C++ source
//
    template<class T>
    struct wrapper {
        T comp;
    };
    template<class U>
    void consume(wrapper<U> formal)
    {
        ...
    }
    wrapper<int> obj;
    consume(obj);
```

Figure D.52: C++ template example #2: source

```
! DWARF description
!
11$:  DW_TAG_structure_type
          DW_AT_name("wrapper")
12$:      DW_TAG_template_type_parameter
              DW_AT_name("T")
              DW_AT_type(reference to "int")
13$:      DW_TAG_member
              DW_AT_name("comp")
              DW_AT_type(reference to 12$)
14$:  DW_TAG_variable
          DW_AT_name("obj")
          DW_AT_type(reference to 11$)
21$:  DW_TAG_subprogram
          DW_AT_name("consume")
22$:      DW_TAG_template_type_parameter
              DW_AT_name("U")
              DW_AT_type(reference to "int")
23$:      DW_TAG_formal_parameter
              DW_AT_name("formal")
              DW_AT_type(reference to 11$)
```

Figure D.53: C++ template example #2: DWARF description

There exist situations where it is not possible for the DWARF to imply anything about the nature of the original template. Consider the C++ template source in Figure D.52 and the DWARF that can describe it in Figure D.53.

In the DW_TAG_subprogram entry for the instance of consume, `U` is described as `int`. The type of formal is `wrapper<U>` in the source. DWARF only represents instantiations of templates; there is no entry which represents `wrapper<U>` which is neither a template parameter nor a template instantiation. The type of formal is described as `wrapper<int>`, the instantiation of `wrapper<U>`, in the DW_AT_type attribute at 23$. There is no description of the relationship between template type parameter `T` at 12$ and `U` at 22$ which was used to instantiate `wrapper<U>`.

A consequence of this is that the DWARF information would not distinguish between the existing example and one where the formal parameter of `consume` were declared in the source to be `wrapper<int>`.

D.12 Template Alias Examples

The C++ template alias shown in Figure D.54 can be described in DWARF as illustrated in Figure D.55 on the following page.

```
// C++ source, template alias example 1
//
template<typename T, typename U>
struct Alpha {
    T tango;
    U uniform;
};
template<typename V> using Beta = Alpha<V,V>;
Beta<long> b;
```

Figure D.54: C++ template alias example #1: source

```
! DWARF representation for variable 'b'
!
20$:  DW_TAG_structure_type
          DW_AT_name("Alpha")
21$:      DW_TAG_template_type_parameter
              DW_AT_name("T")
              DW_AT_type(reference to "long")
22$:      DW_TAG_template_type_parameter
              DW_AT_name("U")
              DW_AT_type(reference to "long")
23$:      DW_TAG_member
              DW_AT_name("tango")
              DW_AT_type(reference to 21$)
24$:      DW_TAG_member
              DW_AT_name("uniform")
              DW_AT_type(reference to 22$)
25$:  DW_TAG_template_alias
          DW_AT_name("Beta")
          DW_AT_type(reference to 20$)
26$:      DW_TAG_template_type_parameter
              DW_AT_name("V")
              DW_AT_type(reference to "long")
27$:  DW_TAG_variable
          DW_AT_name("b")
          DW_AT_type(reference to 25$)
```

Figure D.55: C++ template alias example #1: DWARF description

Similarly, the C++ template alias shown in Figure D.56 can be described in DWARF as illustrated in Figure D.57 on the following page.

```
// C++ source, template alias example 2
//
template<class TX> struct X { };
template<class TY> struct Y { };
template<class T> using Z = Y<T>;
X<Y<int>> y;
X<Z<int>> z;
```

Figure D.56: C++ template alias example #2: source

```
! DWARF representation for X<Y<int>>
!
30$:  DW_TAG_structure_type
          DW_AT_name("Y")
31$:      DW_TAG_template_type_parameter
              DW_AT_name("TY")
              DW_AT_type(reference to "int")
32$:  DW_TAG_structure_type
          DW_AT_name("X")
33$:      DW_TAG_template_type_parameter
              DW_AT_name("TX")
              DW_AT_type(reference to 30$)
!
! DWARF representation for X<Z<int>>
!
40$:  DW_TAG_template_alias
          DW_AT_name("Z")
          DW_AT_type(reference to 30$)
41$:      DW_TAG_template_type_parameter
              DW_AT_name("T")
              DW_AT_type(reference to "int")
42$:  DW_TAG_structure_type
          DW_AT_name("X")
43$:      DW_TAG_template_type_parameter
              DW_AT_name("TX")
              DW_AT_type(reference to 40$)
!
! Note that 32$ and 42$ are actually the same type
!
50$:  DW_TAG_variable
          DW_AT_name("y")
          DW_AT_type(reference to $32)
51$:  DW_TAG_variable
          DW_AT_name("z")
          DW_AT_type(reference to $42)
```

Figure D.57: C++ template alias example #2: DWARF description

D.13 Implicit Pointer Examples

If the compiler determines that the value of an object is constant (either throughout the program, or within a specific range), it may choose to materialize that constant only when used, rather than store it in memory or in a register. The DW_OP_implicit_value operation can be used to describe such a value. Sometimes, the value may not be constant, but still can be easily rematerialized when needed. A DWARF expression terminating in DW_OP_stack_value can be used for this case. The compiler may also eliminate a pointer value where the target of the pointer resides in memory, and the DW_OP_stack_value operator may be used to rematerialize that pointer value. In other cases, the compiler will eliminate a pointer to an object that itself needs to be materialized. Since the location of such an object cannot be represented as a memory address, a DWARF expression cannot give either the location or the actual value or a pointer variable that would refer to that object. The DW_OP_implicit_pointer operation can be used to describe the pointer, and the debugging information entry to which its first operand refers describes the value of the dereferenced object. A DWARF consumer will not be able to show the location or the value of the pointer variable, but it will be able to show the value of the dereferenced pointer.

Consider the C source shown in Figure D.58. Assume that the function `foo` is not inlined, that the argument x is passed in register 5, and that the function `foo` is optimized by the compiler into just an increment of the volatile variable `v`. Given these assumptions a possible DWARF description is shown in Figure D.59 on the following page.

```
struct S { short a; char b, c; };
volatile int v;
void foo (int x)
{
    struct S s = { x, x + 2, x + 3 };
    char *p = &s.b;
    s.a++;
    v++;
}
int main ()
{
    foo (v+1);
    return 0;
}
```

Figure D.58: C implicit pointer example #1: source

```
1$:   DW_TAG_structure_type
          DW_AT_name("S")
          DW_AT_byte_size(4)
10$:      DW_TAG_member
              DW_AT_name("a")
              DW_AT_type(reference to "short int")
              DW_AT_data_member_location(constant 0)
11$:      DW_TAG_member
              DW_AT_name("b")
              DW_AT_type(reference to "char")
              DW_AT_data_member_location(constant 2)
12$:      DW_TAG_member
              DW_AT_name("c")
              DW_AT_type(reference to "char")
              DW_AT_data_member_location(constant 3)
2$:   DW_TAG_subprogram
          DW_AT_name("foo")
20$:      DW_TAG_formal_parameter
              DW_AT_name("x")
              DW_AT_type(reference to "int")
              DW_AT_location(DW_OP_reg5)
21$:      DW_TAG_variable
              DW_AT_name("s")
              DW_AT_type(reference to S at 1$)
              DW_AT_location(expression=
                  DW_OP_breg5(1) DW_OP_stack_value DW_OP_piece(2)
                  DW_OP_breg5(2) DW_OP_stack_value DW_OP_piece(1)
                  DW_OP_breg5(3) DW_OP_stack_value DW_OP_piece(1))
22$:      DW_TAG_variable
              DW_AT_name("p")
              DW_AT_type(reference to "char *")
              DW_AT_location(expression=
                  DW_OP_implicit_pointer(reference to 21$, 2))
```

Figure D.59: C implicit pointer example #1: DWARF description

In Figure D.59, even though variables s and p are both optimized away completely, this DWARF description still allows a debugger to print the value of the variable s, namely `(2, 3, 4)`. Similarly, because the variable s does not live in memory, there is nothing to print for the value of p, but the debugger should still be able to show that `p[0]` is 3, `p[1]` is 4, `p[-1]` is 0 and `p[-2]` is 2.

As a further example, consider the C source shown in Figure D.60. Make the following assumptions about how the code is compiled:

- The function `foo` is inlined into function `main`
- The body of the main function is optimized to just three blocks of instructions which each increment the volatile variable `v`, followed by a block of instructions to return 0 from the function
- Label `label0` is at the start of the main function, `label1` follows the first `v++` block, `label2` follows the second `v++` block and `label3` is at the end of the main function
- Variable `b` is optimized away completely, as it isn't used
- The string literal `"opq"` is optimized away as well

Given these assumptions a possible DWARF description is shown in Figure D.61 on the next page.

```
static const char *b = "opq";
volatile int v;
static inline void foo (int *p)
{
    (*p)++;
    v++;
    p++;
    (*p)++;
    v++;
}

int main ()
{
label0:
    int a[2] =  1, 2 ;
    v++;
label1:
    foo (a);
label2:
    return a[0] + a[1] - 5;
label3:
}
```

Figure D.60: C implicit pointer example #2: source

```
1$:   DW_TAG_variable
          DW_AT_name("b")
          DW_AT_type(reference to "const char *")
          DW_AT_location(expression=
              DW_OP_implicit_pointer(reference to 2$, 0))
2$:   DW_TAG_dwarf_procedure
          DW_AT_location(expression=
              DW_OP_implicit_value(4, {'o', 'p', 'q', '\0'}))
3$:   DW_TAG_subprogram
          DW_AT_name("foo")
          DW_AT_inline(DW_INL_declared_inlined)
30$:      DW_TAG_formal_parameter
              DW_AT_name("p")
              DW_AT_type(reference to "int *")
4$:   DW_TAG_subprogram
          DW_AT_name("main")
40$:      DW_TAG_variable
              DW_AT_name("a")
              DW_AT_type(reference to "int[2]")
              DW_AT_location(location list 98$)
41$:      DW_TAG_inlined_subroutine
              DW_AT_abstract_origin(reference to 3$)
42$:          DW_TAG_formal_parameter
                  DW_AT_abstract_origin(reference to 30$)
                  DW_AT_location(location list 99$)

! .debug_loclists section
98$:  DW_LLE_start_end[<label0 in main> .. <label1 in main>)
          DW_OP_lit1 DW_OP_stack_value DW_OP_piece(4)
          DW_OP_lit2 DW_OP_stack_value DW_OP_piece(4)
      DW_LLE_start_end[<label1 in main> .. <label2 in main>)
          DW_OP_lit2 DW_OP_stack_value DW_OP_piece(4)
          DW_OP_lit2 DW_OP_stack_value DW_OP_piece(4)
      DW_LLE_start_end[<label2 in main> .. <label3 in main>)
          DW_OP_lit2 DW_OP_stack_value DW_OP_piece(4)
          DW_OP_lit3 DW_OP_stack_value DW_OP_piece(4)
      DW_LLE_end_of_list
99$:  DW_LLE_start_end[<label1 in main> .. <label2 in main>)
          DW_OP_implicit_pointer(reference to 40$, 0)
      DW_LLE_start_end[<label2 in main> .. <label3 in main>)
          DW_OP_implicit_pointer(reference to 40$, 4)
      DW_LLE_end_of_list
```

Figure D.61: C implicit pointer example #2: DWARF description

D.14 String Type Examples

Consider the Fortran 2003 string type example source in Figure D.62 following.
The DWARF representation in Figure D.63 on the following page is appropriate.

```
program character_kind
    use iso_fortran_env
    implicit none
    integer, parameter :: ascii =
        selected_char_kind ("ascii")
    integer, parameter :: ucs4  =
        selected_char_kind ('ISO_10646')
    character(kind=ascii, len=26) :: alphabet
    character(kind=ucs4,  len=30) :: hello_world
    character (len=*), parameter :: all_digits="0123456789"

    alphabet = ascii_"abcdefghijklmnopqrstuvwxyz"
    hello_world = ucs4_'Hello World and Ni Hao -- ' &
                  // char (int (z'4F60'), ucs4)     &
                  // char (int (z'597D'), ucs4)

    write (*,*) alphabet
    write (*,*) all_digits

    open (output_unit, encoding='UTF-8')
    write (*,*) trim (hello_world)
end program character_kind
```

Figure D.62: String type example: source

```
1$: DW_TAG_base_type
        DW_AT_encoding (DW_ATE_ASCII)

2$: DW_TAG_base_type
        DW_AT_encoding (DW_ATE_UCS)
        DW_AT_byte_size (4)

3$: DW_TAG_string_type
        DW_AT_byte_size (10)

4$: DW_TAG_const_type
        DW_AT_type (reference to 3$)

5$: DW_TAG_string_type
        DW_AT_type (1$)
        DW_AT_string_length ( ... )
        DW_AT_string_length_byte_size ( ... )
        DW_AT_data_location ( ... )

6$: DW_TAG_string_type
        DW_AT_type (2$)
        DW_AT_string_length ( ... )
        DW_AT_string_length_byte_size ( ... )
        DW_AT_data_location ( ... )

7$: DW_TAG_variable
        DW_AT_name (alphabet)
        DW_AT_type (5$)
        DW_AT_location ( ... )

8$: DW_TAG_constant
        DW_AT_name (all_digits)
        DW_AT_type (4$)
        DW_AT_const_value ( ... )

9$: DW_TAG_variable
        DW_AT_name (hello_world)
        DW_AT_type (6$)
        DW_AT_location ( ... )
```

Figure D.63: String type example: DWARF representation

D.15 Call Site Examples

The following examples use a hypothetical machine which:

- Passes the first argument in register 0, the second in register 1, and the third in register 2.
- Keeps the stack pointer is register 3.
- Has one call preserved register 4.
- Returns a function value in register 0.

D.15.1 Call Site Example #1 (C)

Consider the C source in Figure D.64 following.

```
extern void fn1 (long int, long int, long int);

long int
fn2 (long int a, long int b, long int c)
{
    long int q = 2 * a;
    fn1 (5, 6, 7);
    return 0;
}

long int
fn3 (long int x, long int (*fn4) (long int *))
{
    long int v, w, w2, z;
    w = (*fn4) (&w2);
    v = (*fn4) (&w2);
    z = fn2 (1, v + 1, w);
    {
        int v1 = v + 4;
        z += fn2 (w, v * 2, x);
    }
    return z;
}
```

Figure D.64: Call Site Example #1: Source

Possible generated code for this source is shown using a suggestive pseudo-assembly notation in Figure D.65 on the next page.

```
fn2:
L1:
    %reg2 = 7   ! Load the 3rd argument to fn1
    %reg1 = 6   ! Load the 2nd argument to fn1
    %reg0 = 5   ! Load the 1st argument to fn1
L2:
    call fn1
    %reg0 = 0   ! Load the return value from the function
    return
L3:
fn3:
    ! Decrease stack pointer to reserve local stack frame
    %reg3 = %reg3 - 32
    [%reg3] = %reg4        ! Save the call preserved register to
                           !   stack
    [%reg3 + 8] = %reg0    ! Preserve the x argument value
    [%reg3 + 16] = %reg1   ! Preserve the fn4 argument value
    %reg0 = %reg3 + 24     ! Load address of w2 as argument
    call %reg1             ! Call fn4 (indirect call)
L6:
    %reg2 = [%reg3 + 16]   ! Load the fn4 argument value
    [%reg3 + 16] = %reg0   ! Save the result of the first call (w)
    %reg0 = %reg3 + 24     ! Load address of w2 as argument
    call %reg2             ! Call fn4 (indirect call)
L7:
    %reg4 = %reg0          ! Save the result of the second call (v)
                           !   into register.
    %reg2 = [%reg3 + 16]   ! Load 3rd argument to fn2 (w)
    %reg1 = %reg4 + 1      ! Compute 2nd argument to fn2 (v + 1)
    %reg0 = 1              ! Load 1st argument to fn2
    call fn2
L4:
    %reg2 = [%reg3 + 8]    ! Load the 3rd argument to fn2 (x)
    [%reg3 + 8] = %reg0    ! Save the result of the 3rd call (z)
    %reg0 = [%reg3 + 16]   ! Load the 1st argument to fn2 (w)
    %reg1 = %reg4 + %reg4  ! Compute the 2nd argument to fn2 (v * 2)
    call fn2
L5:
    %reg2 = [%reg3 + 8]    ! Load the value of z from the stack
    %reg0 = %reg0 + %reg2  ! Add result from the 4th call to it
L8:
    %reg4 = [%reg3]        ! Restore original value of call preserved
                           !   register
    %reg3 = %reg3 + 32     ! Leave stack frame
    return
```

Figure D.65: Call Site Example #1: Code

The location list for variable `a` in function `fn2` might look like the following (where the notation "*Range* `[m .. n)`" specifies the range of addresses from `m` through but not including `n` over which the following location description applies):

```
! Before the assignment to register 0, the argument a is live in register 0
!
Range [L1 .. L2)
    DW_OP_reg0

! Afterwards, it is not. The value can perhaps be looked up in the caller
!
Range [L2 .. L3)
    DW_OP_entry_value 1 DW_OP_reg0 DW_OP_stack_value
End-of-list
```

Similarly, the variable `q` in `fn2` then might have this location list:

```
! Before the assignment to register 0, the value of q can be computed as
! two times the contents of register 0
!
Range [L1 .. L2)
    DW_OP_lit2 DW_OP_breg0 0 DW_OP_mul DW_OP_stack_value

! Afterwards. it is not. It can be computed from the original value of
! the first parameter, multiplied by two
!
Range [L2 .. L3)
    DW_OP_lit2 DW_OP_entry_value 1 DW_OP_reg0 DW_OP_mul DW_OP_stack_value
End-of-list
```

Variables `b` and `c` each have a location list similar to that for variable `a`, except for a different label between the two ranges and they use DW_OP_reg1 and DW_OP_reg2, respectively, instead of DW_OP_reg0.

The call sites for all the calls in function `fn3` are children of the DW_TAG_subprogram entry for `fn3` (or of its DW_TAG_lexical_block entry if there is any for the whole function). This is shown in Figure D.66 on the following page.

part 1 of 2

```
    DW_TAG_call_site
        DW_AT_call_return_pc(L6) ! First indirect call to (*fn4) in fn3.
        ! The address of the call is preserved across the call in memory at
        ! stack pointer + 16 bytes.
        DW_AT_call_target(DW_OP_breg3 16 DW_OP_deref)
        DW_TAG_call_site_parameter
            DW_AT_location(DW_OP_reg0)
            ! Value of the first parameter is equal to stack pointer + 24 bytes.
            DW_AT_call_value(DW_OP_breg3 24)
    DW_TAG_call_site
        DW_AT_call_return_pc(L7) ! Second indirect call to (*fn4) in fn3.
        ! The address of the call is not preserved across the call anywhere, but
        ! could be perhaps looked up in fn3's caller.
        DW_AT_call_target(DW_OP_entry_value 1 DW_OP_reg1)
        DW_TAG_call_site_parameter
            DW_AT_location(DW_OP_reg0)
            DW_AT_call_value(DW_OP_breg3 24)
    DW_TAG_call_site
        DW_AT_call_return_pc(L4) ! 3rd call in fn3, direct call to fn2
        DW_AT_call_origin(reference to fn2 DW_TAG_subprogram)
        DW_TAG_call_site_parameter
            DW_AT_call_parameter(reference to formal parameter a in subprogram fn2)
            DW_AT_location(DW_OP_reg0)
            ! First parameter to fn2 is constant 1
            DW_AT_call_value(DW_OP_lit1)
        DW_TAG_call_site_parameter
            DW_AT_call_parameter(reference to formal parameter b in subprogram fn2)
            DW_AT_location(DW_OP_reg1)
            ! Second parameter to fn2 can be computed as the value of the call
            !   preserved register 4 in the fn3 function plus one
            DW_AT_call_value(DW_OP_breg4 1)
        DW_TAG_call_site_parameter
            DW_AT_call_parameter(reference to formal parameter c in subprogram fn2)
            DW_AT_location(DW_OP_reg2)
            ! Third parameter's value is preserved in memory at fn3's stack pointer
            !   plus 16 bytes
            DW_AT_call_value(DW_OP_breg3 16 DW_OP_deref)
```

Figure D.66: Call site example #1: DWARF encoding

part 2 of 2

```
DW_TAG_lexical_block
    DW_AT_low_pc(L4)
    DW_AT_high_pc(L8)
    DW_TAG_variable
        DW_AT_name("v1")
        DW_AT_type(reference to int)
        ! Value of the v1 variable can be computed as value of register 4 plus 4
        DW_AT_location(DW_OP_breg4 4 DW_OP_stack_value)
    DW_TAG_call_site
        DW_AT_call_return_pc(L5) ! 4th call in fn3, direct call to fn2
        DW_AT_call_target(reference to subprogram fn2)
        DW_TAG_call_site_parameter
            DW_AT_call_parameter(reference to formal parameter a in subprogram fn2)
            DW_AT_location(DW_OP_reg0)
            ! Value of the 1st argument is preserved in memory at fn3's stack
            !   pointer + 16 bytes.
            DW_AT_call_value(DW_OP_breg3 16 DW_OP_deref)
        DW_TAG_call_site_parameter
            DW_AT_call_parameter(reference to formal parameter b in subprogram fn2)
            DW_AT_location(DW_OP_reg1)
            ! Value of the 2nd argument can be computed using the preserved
            !   register 4 multiplied by 2
            DW_AT_call_value(DW_OP_lit2 DW_OP_reg4 0 DW_OP_mul)
        DW_TAG_call_site_parameter
            DW_AT_call_parameter(reference to formal parameter c in subprogram fn2)
            DW_AT_location(DW_OP_reg2)
            ! Value of the 3rd argument is not preserved, but could be perhaps
            ! computed from the value passed fn3's caller.
            DW_AT_call_value(DW_OP_entry_value 1 DW_OP_reg0)
```

Figure D.66 Call site example #1: DWARF encoding *(concluded)*

D.15.2 Call Site Example #2 (Fortran)

Consider the Fortran source in Figure D.67 which is used to illustrate how Fortran's "pass by reference" parameters can be handled.

```
subroutine fn4 (n)
    integer :: n, x
    x = n
    n = n / 2
    call fn6
end subroutine
subroutine fn5 (n)
    interface fn4
        subroutine fn4 (n)
            integer :: n
        end subroutine
    end interface fn4
    integer :: n, x
    call fn4 (n)
    x = 5
    call fn4 (x)
end subroutine fn5
```

Figure D.67: Call site example #2: source

Possible generated code for this source is shown using a suggestive pseudo-assembly notation in Figure D.68.

```
fn4:
    %reg2 = [%reg0]    ! Load value of n (passed by reference)
    %reg2 = %reg2 / 2  ! Divide by 2
    [%reg0] = %reg2    ! Update value of n
    call fn6           ! Call some other function
    return

fn5:
    %reg3 = %reg3 - 8  ! Decrease stack pointer to create stack frame
    call fn4           ! Call fn4 with the same argument by reference
                       !   as fn5 has been called with
L9:
    [%reg3] = 5        ! Pass value of 5 by reference to fn4
    %reg0 = %reg3      ! Put address of the value 5 on the stack
                       !   into 1st argument register
    call fn4
L10:
    %reg3 = %reg3 + 8  ! Leave stack frame
    return
```

Figure D.68: Call site example #2: code

The location description for variable x in function fn4 might be:

```
    DW_OP_entry_value 4 DW_OP_breg0 0 DW_OP_deref_size 4
       DW_OP_stack_value
```

The call sites in (just) function fn5 might be as shown in Figure D.69 on the next page.

```
DW_TAG_call_site
    DW_AT_call_return_pc(L9)                       ! First call to fn4
    DW_AT_call_origin(reference to subprogram fn4)
    DW_TAG_call_site_parameter
        DW_AT_call_parameter(reference to formal parameter n in subprogram fn4)
        DW_AT_location(DW_OP_reg0)
        ! The value of register 0 at the time of the call can be perhaps
        !   looked up in fn5's caller
        DW_AT_call_value(DW_OP_entry_value 1 DW_OP_reg0)
        ! DW_AT_call_data_location(DW_OP_push_object_address) ! left out, implicit
        ! And the actual value of the parameter can be also perhaps looked up in
        ! fn5's caller
        DW_AT_call_data_value(DW_OP_entry_value 4 DW_OP_breg0 0 DW_OP_deref_size 4)

DW_TAG_call_site
    DW_AT_call_return_pc(L10)                      ! Second call to fn4
    DW_AT_call_origin(reference to subprogram fn4)
    DW_TAG_call_site_parameter
        DW_AT_call_parameter(reference to formal parameter n in subprogram fn4)
        DW_AT_location(DW_OP_reg0)
        ! The value of register 0 at the time of the call is equal to the stack
        ! pointer value in fn5
        DW_AT_call_value(DW_OP_breg3 0)
        ! DW_AT_call_data_location(DW_OP_push_object_address) ! left out, implicit
        ! And the value passed by reference is constant 5
        DW_AT_call_data_value(DW_OP_lit5)
```

Figure D.69: Call site example #2: DWARF encoding

D.16 Macro Example

Consider the C source in Figure D.70 following which is used to illustrate the DWARF encoding of macro information (see Section 6.3 on page 165).

File a.c

```
#include "a.h"
#define FUNCTION_LIKE_MACRO(x) 4+x
#include "b.h"
```

File a.h

```
#define LONGER_MACRO 1
#define B 2
#include "b.h"
#define B 3
```

File b.h

```
#undef B
#define D 3
#define FUNCTION_LIKE_MACRO(x) 4+x
```

Figure D.70: Macro example: source

Two possible encodings are shown. The first, in Figure D.71 on the next page, is perhaps the simplest possible encoding. It includes all macro information from the main source file (`a.c`) as well as its two included files (`a.h` and `b.h`) in a single macro unit. Further, all strings are included as immediate operands of the macro operators (that is, there is no string pooling). The size of the macro unit is 160 bytes.

The second encoding, in Figure D.72 on page 363, saves space in two ways:

1. Longer strings are pooled by storing them in the `.debug_str` section where they can be referenced more than once.
2. Macro information entries contained in included files are represented as separate macro units which are then imported for each `#include` directive.

The combined size of the three macro units and their referenced strings is 129 bytes.

```
! *** Section .debug_macro contents
! Macro unit for "a.c"
0$h:    Version:        5
        Flags:          2
            offset_size_flag: 0           ! 4-byte offsets
            debug_line_offset_flag: 1     ! Line number offset present
            opcode_operands_table_flag: 0 ! No extensions
        Offset in .debug_line section: 0  ! Line number offset
0$m:    DW_MACRO_start_file, 0, 0     ! Implicit Line: 0, File: 0 "a.c"
        DW_MACRO_start_file, 1, 1     ! #include Line: 1, File: 1 "a.h"
        DW_MACRO_define, 1, "LONGER_MACRO 1"
                                      ! #define Line: 1, String: "LONGER_MACRO 1"
        DW_MACRO_define, 2, "B 2"     ! #define Line: 2, String: "B 2"
        DW_MACRO_start_file, 3, 2     ! #include Line: 3, File: 2 "b.h"
        DW_MACRO_undef, 1, "B"        ! #undef Line: 1, String: "b"
        DW_MACRO_define 2, "D 3"      ! #define Line: 2, String: "D 3"
        DW_MACRO_define, 3, "FUNCTION_LIKE_MACRO(x) 4+x"
                                      ! #define Line: 3,
                                      !   String: "FUNCTION_LIKE_MACRO(x) 4+x"
        DW_MACRO_end_file             ! End "b.h" -> back to "a.h"
        DW_MACRO_define, 4, "B 3"     ! #define Line: 4, String: "B 3"
        DW_MACRO_end_file             ! End "a.h" -> back to "a.c"
        DW_MACRO_define, 2, "FUNCTION_LIKE_MACRO(x) 4+x"
                                      ! #define Line: 2,
                                      !   String: "FUNCTION_LIKE_MACRO(x) 4+x"
        DW_MACRO_start_file, 3, 2     ! #include Line: 3, File: 2 "b.h"
        DW_MACRO_undef, 1, "B"        ! #undef Line: 1, String: "b"
        DW_MACRO_define, 2, "D 3"     ! #define Line: 2, String: "D 3"
        DW_MACRO_define, 3, "FUNCTION_LIKE_MACRO(x) 4+x"
                                      ! #define Line: 3,
                                      !   String: "FUNCTION_LIKE_MACRO(x) 4+x"
        DW_MACRO_end_file             ! End "b.h" -> back to "a.c"
        DW_MACRO_end_file             ! End "a.c" -> back to ""
        0                             ! End macro unit
```

Figure D.71: Macro example: simple DWARF encoding

```
! *** Section .debug_macro contents
! Macro unit for "a.c"
0$h:    Version:        5
        Flags:          2
            offset_size_flag: 0           ! 4-byte offsets
            debug_line_offset_flag: 1     ! Line number offset present
            opcode_operands_table_flag: 0 ! No extensions
        Offset in .debug_line section: 0  ! Line number offset
0$m:    DW_MACRO_start_file, 0, 0     ! Implicit Line: 0, File: 0 "a.c"
        DW_MACRO_start_file, 1, 1     ! #include Line: 1, File: 1 "a.h"
        DW_MACRO_import, i$1h         ! Import unit at i$1h (lines 1-2)
        DW_MACRO_start_file, 3, 2     ! #include Line: 3, File: 2 "b.h"
        DW_MACRO_import, i$2h         ! Import unit i$2h (lines all)
        DW_MACRO_end_file             ! End "b.h" -> back to "a.h"
        DW_MACRO_define, 4, "B 3"     ! #define Line: 4, String: "B 3"
        DW_MACRO_end_file             ! End "a.h" -> back to "a.c"
        DW_MACRO_define, 2, s$1       ! #define Line: 3,
                                      !   String: "FUNCTION_LIKE_MACRO(x) 4+x"
        DW_MACRO_start_file, 3, 2     ! #include Line: 3, File: 2 "b.h"
        DW_MACRO_import, i$2h         ! Import unit i$2h (lines all)
        DW_MACRO_end_file             ! End "b.h" -> back to "a.c"
        DW_MACRO_end_file             ! End "a.c" -> back to ""
        0                             ! End macro unit
! Macro unit for "a.h" lines 1-2
i$1h:   Version:        5
        Flags:          0
            offset_size_flag: 0           ! 4-byte offsets
            debug_line_offset_flag: 0     ! No line number offset
            opcode_operands_table_flag: 0 ! No extensions
i$1m:   DW_MACRO_define_strp, 1, s$2  ! #define Line: 1, String: "LONGER_MACRO 1"
        DW_MACRO_define, 2, "B 2"     ! #define Line: 2, String: "B 2"
        0                             ! End macro unit
! Macro unit for "b.h"
i$2h:   Version:        5
        Flags:          0
            offset_size_flag: 0           ! 4-byte offsets
            debug_line_offset_flag: 0     ! No line number offset
            opcode_operands_table_flag: 0 ! No extensions
i$2m:   DW_MACRO_undef, 1, "B"        ! #undef Line: 1, String: "B"
        DW_MACRO_define, 2, "D 3"     ! #define Line: 2, String: "D 3"
        DW_MACRO_define_strp, 3, s$1  ! #define Line: 3,
                                      !   String: "FUNCTION_LIKE_MACRO(x) 4+x"
        0                             ! End macro unit
! *** Section .debug_str contents
s$1:    String: "FUNCTION_LIKE_MACRO(x) 4+x"
s$2:    String: "LONGER_MACRO 1"
```

Figure D.72: Macro example: sharable DWARF encoding

A number of observations are worth mentioning:

- Strings that are the same size as a reference or less are better represented as immediate operands. Strings longer than twice the size of a reference are better stored in the string table if there are at least two references.
- There is a trade-off between the size of the macro information of a file and the number of times it is included when evaluating whether to create a separate macro unit. However, the amount of overhead (the size of a macro header) needed to represent a unit as well as the size of the operation to import a macro unit are both small.
- A macro unit need not describe all of the macro information in a file. For example, in Figure D.72 the second macro unit (beginning at `i$1h`) includes macros from just the first two lines of file `a.h`.
- An implementation may be able to share macro units across object files (not shown in this example). To support this, it may be advantageous to create macro units in cases where they do not offer an advantage in a single compilation of itself.
- The header of a macro unit that contains a DW_MACRO_start_file operation must include a reference to the compilation line number header to allow interpretation of the file number operands in those commands. However, the presence of those offsets complicates or may preclude sharing across compilations.

Appendix E

DWARF Compression 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 Using Compilation Units

E.1.1 Overview

The general approach is to break up the debug information of a compilation into separate normal and partial 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 meant to suggest traditional contents as a way of explaining the approach, not to be limiting.

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

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

A relocatable object file 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:

1. Given multiple identical (duplicate) section groups, one of them is chosen to be kept and used, while the rest are discarded.
2. Given a section group that is not referenced from any section outside of the section group, the section group is discarded.

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

For example, if a linker determines 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 to reference the `.debug_info` information of one compilation unit from the `.debug_info` section of another compilation unit (DW_FORM_ref_addr provides this).
2. A means to combine multiple contributions to specific sections (for example, `.debug_info`) into a single object file.
3. A means to identify a section group (giving it a name).
4. A means to indicate 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 to indicate 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.1.2 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 E.1 through Figure E.8 of Section E.1.3 on page 371.

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`.

Debugging Information Entry Names

Global labels for debugging information entries (the need for which is 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.1.2.1 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.

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

This also applies to Fortran INCLUDE lines when declarations are included into a subprogram 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 entry may have, as appropriate, any of the attributes assigned to a DW_TAG_compile_unit.

E.1.2.2 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.1.2.3 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.1.3 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 examples 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.1.3.1 C++ Example

The C++ source in Figure E.1 on the following page is used to illustrate the DWARF representation intended to allow duplicate elimination.

Figure E.2 on the next page shows the section group corresponding to the included file wa.h.

Figure E.3 on page 373 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.

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.

File wa.h

```
struct A {
   int i;
};
```

File wa.c

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

Figure E.1: Duplicate elimination example #1: C++ Source

```
==== 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_AT_name("A")
DW.cpp.wa.h.123456.4:     ! linker global symbol
        DW_TAG_member
        DW_AT_name("i")
        DW_AT_type(DW_FORM_ref<n> to DW.cpp.wa.h.123456.2)
            ! (This is a local reference, so the more
            ! compact form DW_FORM_ref<n>
            ! for n = 1,2,4, or 8 can be used)
```

Figure E.2: Duplicate elimination example #1: DWARF section group

E.1.3.2 C Example

The C++ example in this Section might appear to be equally valid as a C example. However, for C it is prudent to include a DW_TAG_imported_unit in the primary unit (see Figure E.3 on the next page) as well as an DW_AT_import attribute that refers to the proper unit in the section group.

```
== 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 E.3: Duplicate elimination example #1: primary compilation unit

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.1.3.3 Fortran Example

For a Fortran example, consider Figure E.4.

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 E.4: Duplicate elimination example #2: Fortran source

Figure E.5 on the next page shows the section group corresponding to the included file CommonStuff.fh.

Figure E.6 on page 375 shows the sections for the primary compilation unit.

A companion main program is shown in Figure E.7 on page 375

```
==== 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 E.5: Duplicate elimination example #2: DWARF section group

```
== 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 E.6: Duplicate elimination example #2: primary unit

File Main.f

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

Figure E.7: 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 E.8 on the following page.

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

E.2 Using Type Units

A large portion of debug information is type information, and in a typical compilation environment, many types are duplicated many times. One method of controlling the amount of duplication is separating each type into a separate COMDAT `.debug_info` section and arranging for the linker to recognize and

```
== 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 E.8: Duplicate elimination example #2: companion DWARF

eliminate duplicates at the individual type level.

Using this technique, each substantial type definition is placed in its own individual section, while the remainder of the DWARF information (non-type information, incomplete type declarations, and definitions of trivial types) is placed in the usual debug information section. In a typical implementation, the relocatable object file may contain one of each of these debug sections:

```
.debug_abbrev
.debug_info
.debug_line
```

and any number of additional COMDAT `.debug_info` sections containing type units.

As discussed in the previous section (Section E.1 on page 365), many linkers today support the concept of a COMDAT group or linkonce section. The general idea is that a "key" can be attached to a section or a group of sections, and the linker will include only one copy of a section group (or individual section) for any given key. For COMDAT `.debug_info` sections, the key is the type signature formed from the algorithm given in Section 7.32 on page 245.

E.2.1 Signature Computation Example

As an example, consider a C++ header file containing the type definitions shown in Figure E.9.

```
namespace N {

    struct B;

    struct C {
        int x;
        int y;
    };

    class A {
    public:
        A(int v);
        int v();
    private:
        int v_;
        struct A *next;
        struct B *bp;
        struct C c;
    };
}
```

Figure E.9: Type signature examples: C++ source

Next, consider one possible representation of the DWARF information that describes the type "struct C" as shown in E.10 on the following page.

In computing a signature for the type `N::C`, flatten the type description into a byte stream according to the procedure outlined in Section 7.32 on page 245. The result is shown in Figure E.11 on page 379.

```
  DW_TAG_type_unit
      DW_AT_language : DW_LANG_C_plus_plus (4)
    DW_TAG_namespace
        DW_AT_name : "N"
L1:
      DW_TAG_structure_type
          DW_AT_name : "C"
          DW_AT_byte_size : 8
          DW_AT_decl_file : 1
          DW_AT_decl_line : 5
        DW_TAG_member
            DW_AT_name : "x"
            DW_AT_decl_file : 1
            DW_AT_decl_line : 6
            DW_AT_type : reference to L2
            DW_AT_data_member_location : 0
        DW_TAG_member
            DW_AT_name : "y"
            DW_AT_decl_file : 1
            DW_AT_decl_line : 7
            DW_AT_type : reference to L2
            DW_AT_data_member_location : 4
L2:
     DW_TAG_base_type
         DW_AT_byte_size : 4
         DW_AT_encoding : DW_ATE_signed
         DW_AT_name : "int"
```

Figure E.10: Type signature computation #1: DWARF representation

Running an MD5 hash over this byte stream, and taking the low-order 64 bits, yields the final signature: 0xd28081e8 dcf5070a.

Next, consider a representation of the DWARF information that describes the type "class A" as shown in Figure E.12 on page 380.

In this example, the structure types `N::A` and `N::C` have each been placed in separate type units. For `N::A`, the actual definition of the type begins at label L1. The definition involves references to the `int` base type and to two pointer types. The information for each of these referenced types is also included in this type unit, since base types and pointer types are trivial types that are not worth the overhead of a separate type unit. The last pointer type contains a reference to an incomplete type `N::B`, which is also included here as a declaration, since the complete type is unknown and its signature is therefore unavailable. There is also a reference to `N::C`, using DW_FORM_ref_sig8 to refer to the type signature for that type.

```
// Step 2: 'C' DW_TAG_namespace "N"
0x43 0x39 0x4e 0x00
// Step 3: 'D' DW_TAG_structure_type
0x44 0x13
// Step 4: 'A' DW_AT_name DW_FORM_string "C"
0x41 0x03 0x08 0x43 0x00
// Step 4: 'A' DW_AT_byte_size DW_FORM_sdata 8
0x41 0x0b 0x0d 0x08
// Step 7: First child ("x")
    // Step 3: 'D' DW_TAG_member
    0x44 0x0d
    // Step 4: 'A' DW_AT_name DW_FORM_string "x"
    0x41 0x03 0x08 0x78 0x00
    // Step 4: 'A' DW_AT_data_member_location DW_FORM_sdata 0
    0x41 0x38 0x0d 0x00
    // Step 6: 'T' DW_AT_type (type #2)
    0x54 0x49
        // Step 3: 'D' DW_TAG_base_type
        0x44 0x24
        // Step 4: 'A' DW_AT_name DW_FORM_string "int"
        0x41 0x03 0x08 0x69 0x6e 0x74 0x00
        // Step 4: 'A' DW_AT_byte_size DW_FORM_sdata 4
        0x41 0x0b 0x0d 0x04
        // Step 4: 'A' DW_AT_encoding DW_FORM_sdata DW_ATE_signed
        0x41 0x3e 0x0d 0x05
        // Step 7: End of DW_TAG_base_type "int"
        0x00
    // Step 7: End of DW_TAG_member "x"
    0x00
// Step 7: Second child ("y")
    // Step 3: 'D' DW_TAG_member
    0x44 0x0d
    // Step 4: 'A' DW_AT_name DW_FORM_string "y"
    0x41 0x03 0x08 0x79 0x00
    // Step 4: 'A' DW_AT_data_member_location DW_FORM_sdata 4
    0x41 0x38 0x0d 0x04
    // Step 6: 'R' DW_AT_type (type #2)
    0x52 0x49 0x02
    // Step 7: End of DW_TAG_member "y"
    0x00
// Step 7: End of DW_TAG_structure_type "C"
0x00
```

Figure E.11: Type signature computation #1: flattened byte stream

part 1 of 2

```
  DW_TAG_type_unit
      DW_AT_language : DW_LANG_C_plus_plus (4)
    DW_TAG_namespace
        DW_AT_name : "N"
L1:
        DW_TAG_class_type
             DW_AT_name : "A"
             DW_AT_byte_size : 20
             DW_AT_decl_file : 1
             DW_AT_decl_line : 10
           DW_TAG_member
                DW_AT_name : "v_"
                DW_AT_decl_file : 1
                DW_AT_decl_line : 15
                DW_AT_type : reference to L2
                DW_AT_data_member_location : 0
                DW_AT_accessibility : DW_ACCESS_private
          DW_TAG_member
               DW_AT_name : "next"
               DW_AT_decl_file : 1
               DW_AT_decl_line : 16
               DW_AT_type : reference to L3
               DW_AT_data_member_location : 4
               DW_AT_accessibility : DW_ACCESS_private
          DW_TAG_member
               DW_AT_name : "bp"
               DW_AT_decl_file : 1
               DW_AT_decl_line : 17
               DW_AT_type : reference to L4
               DW_AT_data_member_location : 8
               DW_AT_accessibility : DW_ACCESS_private
          DW_TAG_member
               DW_AT_name : "c"
               DW_AT_decl_file : 1
               DW_AT_decl_line : 18
               DW_AT_type : 0xd28081e8 dcf5070a (signature for struct C)
               DW_AT_data_member_location : 12
               DW_AT_accessibility : DW_ACCESS_private
```

Figure E.12: Type signature computation #2: DWARF representation

part 2 of 2

```
      DW_TAG_subprogram
           DW_AT_external : 1
           DW_AT_name : "A"
           DW_AT_decl_file : 1
           DW_AT_decl_line : 12
           DW_AT_declaration : 1
        DW_TAG_formal_parameter
           DW_AT_type : reference to L3
           DW_AT_artificial : 1
        DW_TAG_formal_parameter
           DW_AT_type : reference to L2
       DW_TAG_subprogram
           DW_AT_external : 1
           DW_AT_name : "v"
           DW_AT_decl_file : 1
           DW_AT_decl_line : 13
           DW_AT_type : reference to L2
           DW_AT_declaration : 1
         DW_TAG_formal_parameter
           DW_AT_type : reference to L3
           DW_AT_artificial : 1
L2:
    DW_TAG_base_type
         DW_AT_byte_size : 4
         DW_AT_encoding : DW_ATE_signed
         DW_AT_name : "int"
L3:
    DW_TAG_pointer_type
         DW_AT_type : reference to L1
L4:
    DW_TAG_pointer_type
         DW_AT_type : reference to L5
    DW_TAG_namespace
         DW_AT_name : "N"
L5:
       DW_TAG_structure_type
           DW_AT_name : "B"
           DW_AT_declaration : 1
```

Figure E.12: Type signature computation #2: DWARF representation *(concluded)*

part 1 of 3

```
// Step 2: 'C' DW_TAG_namespace "N"
0x43 0x39 0x4e 0x00
// Step 3: 'D' DW_TAG_class_type
0x44 0x02
// Step 4: 'A' DW_AT_name DW_FORM_string "A"
0x41 0x03 0x08 0x41 0x00
// Step 4: 'A' DW_AT_byte_size DW_FORM_sdata 20
0x41 0x0b 0x0d 0x14
// Step 7: First child ("v_")
    // Step 3: 'D' DW_TAG_member
    0x44 0x0d
    // Step 4: 'A' DW_AT_name DW_FORM_string "v_"
    0x41 0x03 0x08 0x76 0x5f 0x00
    // Step 4: 'A' DW_AT_accessibility DW_FORM_sdata DW_ACCESS_private
    0x41 0x32 0x0d 0x03
    // Step 4: 'A' DW_AT_data_member_location DW_FORM_sdata 0
    0x41 0x38 0x0d 0x00
    // Step 6: 'T' DW_AT_type (type #2)
    0x54 0x49
        // Step 3: 'D' DW_TAG_base_type
        0x44 0x24
        // Step 4: 'A' DW_AT_name DW_FORM_string "int"
        0x41 0x03 0x08 0x69 0x6e 0x74 0x00
        // Step 4: 'A' DW_AT_byte_size DW_FORM_sdata 4
        0x41 0x0b 0x0d 0x04
        // Step 4: 'A' DW_AT_encoding DW_FORM_sdata DW_ATE_signed
        0x41 0x3e 0x0d 0x05
        // Step 7: End of DW_TAG_base_type "int"
        0x00
    // Step 7: End of DW_TAG_member "v_"
    0x00
// Step 7: Second child ("next")
    // Step 3: 'D' DW_TAG_member
    0x44 0x0d
    // Step 4: 'A' DW_AT_name DW_FORM_string "next"
    0x41 0x03 0x08 0x6e 0x65 0x78 0x74 0x00
    // Step 4: 'A' DW_AT_accessibility DW_FORM_sdata DW_ACCESS_private
    0x41 0x32 0x0d 0x03
    // Step 4: 'A' DW_AT_data_member_location DW_FORM_sdata 4
    0x41 0x38 0x0d 0x04
```

Figure E.13: Type signature example #2: flattened byte stream

part 2 of 3

```
    // Step 6: 'T' DW_AT_type (type #3)
    0x54 0x49
        // Step 3: 'D' DW_TAG_pointer_type
        0x44 0x0f
        // Step 5: 'N' DW_AT_type
        0x4e 0x49
        // Step 5: 'C' DW_TAG_namespace "N" 'E'
        0x43 0x39 0x4e 0x00 0x45
        // Step 5: "A"
        0x41 0x00
        // Step 7: End of DW_TAG_pointer_type
        0x00
    // Step 7: End of DW_TAG_member "next"
    0x00
// Step 7: Third child ("bp")
    // Step 3: 'D' DW_TAG_member
    0x44 0x0d
    // Step 4: 'A' DW_AT_name DW_FORM_string "bp"
    0x41 0x03 0x08 0x62 0x70 0x00
    // Step 4: 'A' DW_AT_accessibility DW_FORM_sdata DW_ACCESS_private
    0x41 0x32 0x0d 0x03
    // Step 4: 'A' DW_AT_data_member_location DW_FORM_sdata 8
    0x41 0x38 0x0d 0x08
    // Step 6: 'T' DW_AT_type (type #4)
    0x54 0x49
        // Step 3: 'D' DW_TAG_pointer_type
        0x44 0x0f
        // Step 5: 'N' DW_AT_type
        0x4e 0x49
        // Step 5: 'C' DW_TAG_namespace "N" 'E'
        0x43 0x39 0x4e 0x00 0x45
        // Step 5: "B"
        0x42 0x00
        // Step 7: End of DW_TAG_pointer_type
        0x00
    // Step 7: End of DW_TAG_member "next"
    0x00
// Step 7: Fourth child ("c")
    // Step 3: 'D' DW_TAG_member
    0x44 0x0d
    // Step 4: 'A' DW_AT_name DW_FORM_string "c"
    0x41 0x03 0x08 0x63 0x00
    // Step 4: 'A' DW_AT_accessibility DW_FORM_sdata DW_ACCESS_private
    0x41 0x32 0x0d 0x03
```

Figure E.13: Type signature example #2: flattened byte stream *(continued)*

part 3 of 3

```
    // Step 4: 'A' DW_AT_data_member_location DW_FORM_sdata 12
    0x41 0x38 0x0d 0x0c
    // Step 6: 'T' DW_AT_type (type #5)
    0x54 0x49
        // Step 2: 'C' DW_TAG_namespace "N"
        0x43 0x39 0x4e 0x00
        // Step 3: 'D' DW_TAG_structure_type
        0x44 0x13
        // Step 4: 'A' DW_AT_name DW_FORM_string "C"
        0x41 0x03 0x08 0x43 0x00
        // Step 4: 'A' DW_AT_byte_size DW_FORM_sdata 8
        0x41 0x0b 0x0d 0x08
        // Step 7: First child ("x")
            // Step 3: 'D' DW_TAG_member
            0x44 0x0d
            // Step 4: 'A' DW_AT_name DW_FORM_string "x"
            0x41 0x03 0x08 0x78 0x00
            // Step 4: 'A' DW_AT_data_member_location DW_FORM_sdata 0
            0x41 0x38 0x0d 0x00
            // Step 6: 'R' DW_AT_type (type #2)
            0x52 0x49 0x02
            // Step 7: End of DW_TAG_member "x"
            0x00
        // Step 7: Second child ("y")
            // Step 3: 'D' DW_TAG_member
            0x44 0x0d
            // Step 4: 'A' DW_AT_name DW_FORM_string "y"
            0x41 0x03 0x08 0x79 0x00
            // Step 4: 'A' DW_AT_data_member_location DW_FORM_sdata 4
            0x41 0x38 0x0d 0x04
            // Step 6: 'R' DW_AT_type (type #2)
            0x52 0x49 0x02
            // Step 7: End of DW_TAG_member "y"
            0x00
        // Step 7: End of DW_TAG_structure_type "C"
        0x00
    // Step 7: End of DW_TAG_member "c"
    0x00
// Step 7: Fifth child ("A")
    // Step 3: 'S' DW_TAG_subprogram "A"
    0x53 0x2e 0x41 0x00
// Step 7: Sixth child ("v")
    // Step 3: 'S' DW_TAG_subprogram "v"
    0x53 0x2e 0x76 0x00
// Step 7: End of DW_TAG_structure_type "A"
0x00
```

Figure E.13: Type signature example #2: flattened byte stream *(concluded)*

In computing a signature for the type `N::A`, flatten the type description into a byte stream according to the procedure outlined in Section 7.32 on page 245. The result is shown in Figure E.13 on page 382.

Running an MD5 hash over this byte stream, and taking the low-order 64 bits, yields the final signature: 0xd6d160f5 5589f6e9.

A source file that includes this header file may declare a variable of type `N::A`, and its DWARF information may look like that shown in Figure E.14.

```
DW_TAG_compile_unit
...
DW_TAG_subprogram
  ...
  DW_TAG_variable
    DW_AT_name : "a"
    DW_AT_type : (signature) 0xd6d160f5 5589f6e9
    DW_AT_location : ...
  ...
```

Figure E.14: Type signature example usage

E.2.2 Type Signature Computation Grammar

Figure E.15 on the next page presents a semi-formal grammar that may aid in understanding how the bytes of the flattened type description are formed during the type signature computation algorithm of Section 7.32 on page 245.

```
signature
    : opt-context debug-entry attributes children
opt-context               // Step 2
    : 'C' tag-code string opt-context
    : empty
debug-entry               // Step 3
    : 'D' tag-code
attributes                // Steps 4, 5, 6
    : attribute attributes
    : empty
attribute
    : 'A' at-code form-encoded-value     // Normal attributes
    : 'N' at-code opt-context 'E' string // Reference to type by name
    : 'R' at-code back-ref               // Back-reference to visited type
    : 'T' at-code signature              // Recursive type
children                 //  Step 7
    : child children
    : '\0'
child
    : 'S' tag-code string
    : signature
tag-code
    : <ULEB128>
at-code
    : <ULEB128>
form-encoded-value
    : DW_FORM_sdata value
    : DW_FORM_flag value
    : DW_FORM_string string
    : DW_FORM_block block
DW_FORM_string
    : '\x08'
DW_FORM_block
    : '\x09'
DW_FORM_flag
    : '\x0c'
DW_FORM_sdata
    : '\x0d'
value
    : <SLEB128>
block
    : <ULEB128> <fixed-length-block> // The ULEB128 gives the length of the block
back-ref
    : <ULEB128>
string
    : <null-terminated-string>
empty
    :
```

Figure E.15: Type signature computation grammar

E.2.3 Declarations Completing Non-Defining Declarations

Consider a compilation unit that contains a definition of the member function `N::A::v()` from Figure E.9 on page 377. A possible representation of the debug information for this function in the compilation unit is shown in Figure E.16.

```
  DW_TAG_namespace
      DW_AT_name : "N"
L1:
    DW_TAG_class_type
        DW_AT_name : "A"
        DW_AT_declaration : true
        DW_AT_signature : 0xd6d160f5 5589f6e9
L2:
      DW_TAG_subprogram
          DW_AT_external : 1
          DW_AT_name : "v"
          DW_AT_decl_file : 1
          DW_AT_decl_line : 13
          DW_AT_type : reference to L3
          DW_AT_declaration : 1
        DW_TAG_formal_parameter
            DW_AT_type : reference to L4
            DW_AT_artificial : 1
...
L3:
  DW_TAG_base_type
      DW_AT_byte_size : 4
      DW_AT_encoding : DW_ATE_signed
      DW_AT_name : "int"
...
L4:
  DW_TAG_pointer_type
      DW_AT_type : reference to L1
...
  DW_TAG_subprogram
      DW_AT_specification : reference to L2
      DW_AT_decl_file : 2
      DW_AT_decl_line : 25
      DW_AT_low_pc : ...
      DW_AT_high_pc : ...
    DW_TAG_lexical_block
    ...
...
```

Figure E.16: Completing declaration of a member function: DWARF encoding

E.3 Summary of Compression Techniques

E.3.1 #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.3.2 Eliminating function duplication

Function templates (C++) result in code for the same template instantiation being compiled into multiple archives or relocatable object files. 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 section groups are marked to ensure duplicate removal, so that 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 is slightly different in that the `<file-designator>` should be interpreted as a `<file-designator>`.

E.3.3 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.3.4 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.

E.3.5 Separate Type Units

Each complete declaration of a globally-visible type can be placed in its own separate type section, with a group key derived from the type signature. The linker can then remove all duplicate type declarations based on the key.

(empty page)

Appendix F

Split DWARF Object Files (Informative)

With the traditional DWARF format, debug information is designed with the expectation that it will be processed by the linker to produce an output binary with complete debug information, and with fully-resolved references to locations within the application. For very large applications, however, this approach can result in excessively large link times and excessively large output files.

Several vendors have independently developed proprietary approaches that allow the debug information to remain in the relocatable object files, so that the linker does not have to process the debug information or copy it to the output file. These approaches have all required that additional information be made available to the debug information consumer, and that the consumer perform some minimal amount of relocation in order to interpret the debug info correctly. The additional information required, in the form of load maps or symbol tables, and the details of the relocation are not covered by the DWARF specification, and vary with each vendor's implementation.

Section 7.3.2 on page 187 describes a platform-independent mechanism that allows a producer to split the debugging information into relocatable and non-relocatable partitions. This Appendix describes the use of split DWARF object files and provides some illustrative examples.

F.1 Overview

DWARF Version 5 introduces an optional set of debugging sections that allow the compiler to partition the debugging information into a set of (small) sections that require link-time relocation and a set of (large) sections that do not. The sections that require relocation are written to the relocatable object file as usual, and are linked into the final executable. The sections that do not require relocation, however, can be written to the relocatable object (.o) file but ignored by the linker, or they can be written to a separate DWARF object (.dwo) file that need not be accessed by the linker.

The optional set of debugging sections includes the following:

- `.debug_abbrev.dwo` - Contains the abbreviations table(s) used by the `.debug_info.dwo` section.
- `.debug_info.dwo` - Contains the DW_TAG_compile_unit and DW_TAG_type_unit DIEs and their descendants. This is the bulk of the debugging information for the compilation unit that is normally found in the `.debug_info` section.
- `.debug_loclists.dwo` - Contains the location lists referenced by the debugging information entries in the `.debug_info.dwo` section. This contains the location lists normally found in the `.debug_loclists` section.
- `.debug_str.dwo` - Contains the string table for all indirect strings referenced by the debugging information in the `.debug_info.dwo` sections.
- `.debug_str_offsets.dwo` - Contains the string offsets table for the strings in the `.debug_str.dwo` section.
- `.debug_macro.dwo` - Contains macro definition information, normally found in the `.debug_macro` section.
- `.debug_line.dwo` - Contains specialized line number tables for the type units in the `.debug_info.dwo` section. These tables contain only the directory and filename lists needed to interpret DW_AT_decl_file attributes in the debugging information entries. Actual line number tables remain in the `.debug_line` section, and remain in the relocatable object (.o) files.

In a `.dwo` file, there is no benefit to having a separate string section for directories and file names because the primary string table will never be stripped. Accordingly, no `.debug_line_str.dwo` section is defined. Content descriptions corresponding to DW_FORM_line_strp in an executable file (for example, in the skeleton compilation unit) instead use one of the forms DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 or DW_FORM_strx4. This allows directory and file name strings to be merged with general strings and across compilations in package files (where they are not subject to potential stripping).

In a `.dwo` file, referring to a string using DW_FORM_strp is valid, but such use results in a file that cannot be incorporated into a package file (which involves string merging).

In order for the consumer to locate and process the debug information, the compiler must produce a small amount of debug information that passes through the linker into the output binary. A skeleton `.debug_info` section for each compilation unit contains a reference to the corresponding `.o` or `.dwo` file, and the `.debug_line` section (which is typically small compared to the `.debug_info` sections) is linked into the output binary, as is the `.debug_addr` section.

The debug sections that continue to be linked into the output binary include the following:

- `.debug_abbrev` - Contains the abbreviation codes used by the skeleton `.debug_info` section.
- `.debug_addr` - Contains references to loadable sections, indexed by attributes of one of the forms DW_FORM_addrx, DW_FORM_addrx1, DW_FORM_addrx2, DW_FORM_addrx3, DW_FORM_addrx4, or location expression DW_OP_addrx opcodes.
- `.debug_aranges` - Contains the accelerated range lookup table for the compilation unit.
- `.debug_frame` - Contains the frame tables.
- `.debug_info` - Contains a skeleton skeleton compilation unit DIE, which has no children.
- `.debug_line` - Contains the line number tables. (These could be moved to the .dwo file, but in order to do so, each DW_LNE_set_address opcode would need to be replaced by a new opcode that referenced an entry in the `.debug_addr` section. Furthermore, leaving this section in the .o file allows many debug info consumers to remain unaware of .dwo files.)
- `.debug_line_str` - Contains strings for file names used in combination with the `.debug_line` section.
- `.debug_names` - Contains the names for use in building an index section. The section header refers to a compilation unit offset, which is the offset of the skeleton compilation unit in the `.debug_info` section.
- `.debug_str` - Contains any strings referenced by the skeleton `.debug_info` sections (via DW_FORM_strp, DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 or DW_FORM_strx4).

- `.debug_str_offsets` - Contains the string offsets table for the strings in the `.debug_str` section (if one of the forms DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 or DW_FORM_strx4 is used).

The skeleton compilation unit DIE may have the following attributes:

DW_AT_addr_base	DW_AT_high_pc	DW_AT_stmt_list
DW_AT_comp_dir	DW_AT_low_pc	DW_AT_str_offsets_base
DW_AT_dwo_name	DW_AT_ranges	

All other attributes of the compilation unit DIE are moved to the full DIE in the `.debug_info.dwo` section.

The `dwo_id` field is present in headers of the skeleton DIE and the header of the full DIE, so that a consumer can verify a match.

Relocations are neither necessary nor useful in `.dwo` files, because the `.dwo` files contain only debugging information that does not need to be processed by a linker. Relocations are rendered unnecessary by these strategies:

1. Some values needing relocation are kept in the `.o` file (for example, references to the line number program from the skeleton compilation unit).

2. Some values do not need a relocation because they refer from one `.dwo` section to another `.dwo` section in the same compilation unit.

3. Some values that need a relocation to refer to a relocatable program address use one of the DW_FORM_addrx, DW_FORM_addrx1, DW_FORM_addrx2, DW_FORM_addrx3 or DW_FORM_addrx4 forms, referencing a relocatable value in the `.debug_addr` section (which remains in the .o file).

Table F.1 on the following page summarizes which attributes are defined for use in the various kinds of compilation units (see Section 3.1 on page 59). It compares and contrasts both conventional and split object-related kinds.

The split dwarf object file design depends on having an index of debugging information available to the consumer. For name lookups, the consumer can use the `.debug_names` index section (see Section 6.1 on page 135) to locate a skeleton compilation unit. The DW_AT_comp_dir and DW_AT_dwo_name attributes in the skeleton compilation unit can then be used to locate the corresponding DWARF object file for the compilation unit. Similarly, for an address lookup, the consumer can use the `.debug_aranges` table, which will also lead to a skeleton compilation unit. For a file and line number lookup, the skeleton compilation units can be used to locate the line number tables.

Table F.1: Unit attributes by unit kind

Attribute	Unit Kind				
	Conventional		Skeleton and Split		
	Full & Partial	Type	Skeleton	Split Full	Split Type
DW_AT_addr_base	√		√		
DW_AT_base_types	√				
DW_AT_comp_dir	√		√		
DW_AT_dwo_name			√		
DW_AT_entry_pc	√			√	
DW_AT_high_pc	√		√		
DW_AT_identifier_case	√			√	
DW_AT_language	√	√		√	√
DW_AT_loclists_base	√				
DW_AT_low_pc	√		√		
DW_AT_macros	√			√	
DW_AT_main_subprogram	√			√	
DW_AT_name	√			√	
DW_AT_producer	√			√	
DW_AT_ranges	√			√	
DW_AT_rnglists_base	√				
DW_AT_stmt_list	√	√	√		√
DW_AT_str_offsets_base	√	√	√		
DW_AT_use_UTF8	√	√	√	√	√

F.2 Split DWARF Object File Example

Consider the example source code in Figure F.1, Figure F.2 on the following page and Figure F.3 on page 398. When compiled with split DWARF, we will have two DWARF object files, `demo1.o` and `demo2.o`, and two split DWARF object files, `demo1.dwo` and `demo2.dwo`.

In this section, we will use this example to show how the connections between the relocatable object file and the split DWARF object file are maintained through the linking process. In the next section, we will use this same example to show how two or more split DWARF object files are combined into a DWARF package file.

File demo1.cc

```
#include "demo.h"

bool Box::contains(const Point& p) const
{
    return (p.x() >= ll_.x() && p.x() <= ur_.x() &&
            p.y() >= ll_.y() && p.y() <= ur_.y());
}
```

Figure F.1: Split object example: source fragment #1

File demo2.cc

```
#include "demo.h"

bool Line::clip(const Box& b)
{
  float slope = (end_.y() - start_.y()) / (end_.x() - start_.x());
  while (1) {
    // Trivial acceptance.
    if (b.contains(start_) && b.contains(end_)) return true;

    // Trivial rejection.
    if (start_.x() < b.l() && end_.x() < b.l()) return false;
    if (start_.x() > b.r() && end_.x() > b.r()) return false;
    if (start_.y() < b.b() && end_.y() < b.b()) return false;
    if (start_.y() > b.t() && end_.y() > b.t()) return false;

    if (b.contains(start_)) {
      // Swap points so that start_ is outside the clipping
      // rectangle.
      Point temp = start_;
      start_ = end_;
      end_ = temp;
    }

    if (start_.x() < b.l())
      start_ = Point(b.l(),
                     start_.y() + (b.l() - start_.x()) * slope);
    else if (start_.x() > b.r())
      start_ = Point(b.r(),
                     start_.y() + (b.r() - start_.x()) * slope);
    else if (start_.y() < b.b())
      start_ = Point(start_.x() + (b.b() - start_.y()) / slope,
                     b.b());
    else if (start_.y() > b.t())
      start_ = Point(start_.x() + (b.t() - start_.y()) / slope,
                     b.t());
  }
}
```

Figure F.2: Split object example: source fragment #2

File demo.h

```
class A {
  public:
    Point(float x, float y) : x_(x), y_(y){}
    float x() const { return x_; }
    float y() const { return y_; }
  private:
    float x_;
    float y_;
};

class Line {
  public:
    Line(Point start, Point end) : start_(start), end_(end){}
    bool clip(const Box& b);
    Point start() const { return start_; }
    Point end() const { return end_; }
  private:
    Point start_;
    Point end_;
};

class Box {
  public:
    Box(float l, float r, float b, float t) : ll_(l, b), ur_(r, t){}
    Box(Point ll, Point ur) : ll_(ll), ur_(ur){}
    bool contains(const Point& p) const;
    float l() const { return ll_.x(); }
    float r() const { return ur_.x(); }
    float b() const { return ll_.y(); }
    float t() const { return ur_.y(); }
  private:
    Point ll_;
    Point ur_;
};
```

Figure F.3: Split object example: source fragment #3

F.2.1 Contents of the Object Files

The object files each contain the following sections of debug information:

```
.debug_abbrev
.debug_info
.debug_line
.debug_str
.debug_addr
.debug_names
.debug_aranges
```

The `.debug_abbrev` section contains just a single entry describing the skeleton compilation unit DIE.

The DWARF description in the `.debug_info` section contains just a single DIE, the skeleton compilation unit, which may look like Figure F.4 following.

```
DW_TAG_skeleton_unit
    DW_AT_comp_dir: (reference to directory name in .debug_str)
    DW_AT_dwo_name: (reference to "demo1.dwo" in .debug_str)
    DW_AT_addr_base: (reference to .debug_addr section)
    DW_AT_stmt_list: (reference to .debug_line section)
```

Figure F.4: Split object example: skeleton DWARF description

The DW_AT_comp_dir and DW_AT_dwo_name attributes provide the location of the corresponding split DWARF object file that contains the full debug information; that file is normally expected to be in the same directory as the object file itself.

The `dwo_id` field in the header of the skeleton unit provides an ID or key for the debug information contained in the DWARF object file. This ID serves two purposes: it can be used to verify that the debug information in the split DWARF object file matches the information in the object file, and it can be used to find the debug information in a DWARF package file.

The DW_AT_addr_base attribute contains the relocatable offset of this object file's contribution to the `.debug_addr` section.

The DW_AT_stmt_list attribute contains the relocatable offset of this file's contribution to the `.debug_line` table.

The .debug_line section contains the full line number table for the compiled code in the object file. As shown in Figure F.1 on page 396, the line number program header lists the two file names, demo.h and demo1.cc, and contains line number programs for Box::contains, Point::x, and Point::y.

The .debug_str section contains the strings referenced indirectly by the compilation unit DIE and by the line number program.

The .debug_addr section contains relocatable addresses of locations in the loadable text and data that are referenced by debugging information entries in the split DWARF object. In the example in F.3 on page 398, demo1.o may have three entries:

Slot	Location referenced
0	low PC value for Box::contains
1	low PC value for Point::x
2	low PC value for Point::y

The .debug_names section contains the names defined by the debugging information in the split DWARF object file (see Section 6.1.1.1 on page 137), and references the skeleton compilation unit. When linked together into a final executable, they can be used by a DWARF consumer to lookup a name to find one or more skeleton compilation units that provide information about that name. From the skeleton compilation unit, the consumer can find the split DWARF object file that it can then read to get the full DWARF information.

The .debug_aranges section contains the PC ranges defined in this compilation unit, and allow a DWARF consumer to map a PC value to a skeleton compilation unit, and then to a split DWARF object file.

F.2.2 Contents of the Linked Executable File

When demo1.o and demo2.o are linked together (along with a main program and other necessary library routines that we will ignore here for simplicity), the resulting executable file will contain at least the two skeleton compilation units in the .debug_info section, as shown in Figure F.5 following.

Each skeleton compilation unit has a DW_AT_stmt_list attribute, which provides the relocated offset to that compilation unit's contribution in the executable's .debug_line section. In this example, the line number information for demo1.dwo begins at offset 120, and for demo2.dwo, it begins at offset 200.

```
    DW_TAG_skeleton_unit
        DW_AT_comp_dir: (reference to directory name in .debug_str)
        DW_AT_dwo_name: (reference to "demo1.dwo" in .debug_str)
        DW_AT_addr_base: 48 (offset in .debug_addr)
        DW_AT_stmt_list: 120 (offset in .debug_line)
    DW_TAG_skeleton_unit
        DW_AT_comp_dir: (reference to directory name in .debug_str)
        DW_AT_dwo_name: (reference to "demo2.dwo" in .debug_str)
        DW_AT_addr_base: 80 (offset in .debug_addr)
        DW_AT_stmt_list: 200 (offset in .debug_line)
```

Figure F.5: Split object example: executable file DWARF excerpts

Each skeleton compilation unit also has a DW_AT_addr_base attribute, which provides the relocated offset to that compilation unit's contribution in the executable's `.debug_addr` section. Unlike the DW_AT_stmt_list attribute, the offset refers to the first address table slot, not to the section header. In this example, we see that the first address (slot 0) from `demo1.o` begins at offset 48. Because the `.debug_addr` section contains an 8-byte header, the object file's contribution to the section actually begins at offset 40 (for a 64-bit DWARF object, the header would be 16 bytes long, and the value for the DW_AT_addr_base attribute would then be 56). All attributes in `demo1.dwo` that use DW_FORM_addrx, DW_FORM_addrx1, DW_FORM_addrx2, DW_FORM_addrx3 or DW_FORM_addrx4 would then refer to address table slots relative to that offset. Likewise, the `.debug_addr` contribution from `demo2.dwo` begins at offset 72, and its first address slot is at offset 80. Because these contributions have been processed by the linker, they contain relocated values for the addresses in the program that are referred to by the debug information.

The linked executable will also contain `.debug_abbrev`, `.debug_str`, `.debug_names` and `.debug_aranges` sections, each the result of combining and relocating the contributions from the relocatable object files.

F.2.3 Contents of the Split DWARF Object Files

The split DWARF object files each contain the following sections:

```
.debug_abbrev.dwo
.debug_info.dwo (for the compilation unit)
.debug_info.dwo (one COMDAT section for each type unit)
.debug_loclists.dwo
.debug_line.dwo
.debug_macro.dwo
.debug_rnglists.dwo
.debug_str_offsets.dwo
.debug_str.dwo
```

The `.debug_abbrev.dwo` section contains the abbreviation declarations for the debugging information entries in the `.debug_info.dwo` section.

The `.debug_info.dwo` section containing the compilation unit contains the full debugging information for the compile unit, and looks much like a normal `.debug_info` section in a non-split object file, with the following exceptions:

- The DW_TAG_compile_unit DIE does not need to repeat the DW_AT_ranges, DW_AT_low_pc, DW_AT_high_pc, and DW_AT_stmt_list attributes that are provided in the skeleton compilation unit.
- References to strings in the string table use the form code DW_FORM_strx, DW_FORM_strx1, DW_FORM_strx2, DW_FORM_strx3 or DW_FORM_strx4, referring to slots in the `.debug_str_offsets.dwo` section.
- References to relocatable addresses in the object file use one of the form codes DW_FORM_addrx, DW_FORM_addrx1, DW_FORM_addrx2, DW_FORM_addrx3 or DW_FORM_addrx4, referring to slots in the `.debug_addr` table, relative to the base offset given by DW_AT_addr_base in the skeleton compilation unit.

Figure F.6 following presents excerpts from the `.debug_info.dwo` section for `demo1.dwo`.

In the defining declaration for `Box::contains` at 5$, the DW_AT_low_pc attribute is represented using DW_FORM_addrx, which refers to slot 0 in the `.debug_addr` table from `demo1.o`. That slot contains the relocated address of the beginning of the function.

part 1 of 2

```
    DW_TAG_compile_unit
        DW_AT_producer [DW_FORM_strx]: (slot 15) (producer string)
        DW_AT_language: DW_LANG_C_plus_plus
        DW_AT_name [DW_FORM_strx]: (slot 7) "demo1.cc"
        DW_AT_comp_dir [DW_FORM_strx]: (slot 4) (directory name)
1$:     DW_TAG_class_type
            DW_AT_name [DW_FORM_strx]: (slot 12) "Point"
            DW_AT_signature [DW_FORM_ref_sig8]: 0x2f33248f03ff18ab
            DW_AT_declaration: true
2$:         DW_TAG_subprogram
                DW_AT_external: true
                DW_AT_name [DW_FORM_strx]: (slot 12) "Point"
                DW_AT_decl_file: 1
                DW_AT_decl_line: 5
                DW_AT_linkage_name [DW_FORM_strx]: (slot 16) "_ZN5PointC4Eff"
                DW_AT_accessibility: DW_ACCESS_public
                DW_AT_declaration: true
            ...
3$:     DW_TAG_class_type
            DW_AT_name [DW_FORM_string]: "Box"
            DW_AT_signature [DW_FORM_ref_sig8]: 0xe97a3917c5a6529b
            DW_AT_declaration: true
          ...
4$:         DW_TAG_subprogram
                DW_AT_external: true
                DW_AT_name [DW_FORM_strx]: (slot 0) "contains"
                DW_AT_decl_file: 1
                DW_AT_decl_line: 28
                DW_AT_linkage_name [DW_FORM_strx: (slot 8)
                                                   "_ZNK3Box8containsERK5Point"
                DW_AT_type: (reference to 7$)
                DW_AT_accessibility: DW_ACCESS_public
                DW_AT_declaration: true
          ...
```

Figure F.6: Split object example: `demo1.dwo` excerpts

Each type unit is contained in its own COMDAT `.debug_info.dwo` section, and looks like a normal type unit in a non-split object, except that the DW_TAG_type_unit DIE contains a DW_AT_stmt_list attribute that refers to a specialized `.debug_line.dwo` section. This section contains a normal line number program header with a list of include directories and filenames, but no line number program. This section is used only as a reference for filenames needed for DW_AT_decl_file attributes within the type unit.

part 2 of 2

```
5$:   DW_TAG_subprogram
          DW_AT_specification: (reference to 4$)
          DW_AT_decl_file: 2
          DW_AT_decl_line: 3
          DW_AT_low_pc [DW_FORM_addrx]: (slot 0)
          DW_AT_high_pc [DW_FORM_data8]: 0xbb
          DW_AT_frame_base: DW_OP_call_frame_cfa
          DW_AT_object_pointer: (reference to 6$)
6$:       DW_TAG_formal_parameter
              DW_AT_name [DW_FORM_strx]: (slot 13): "this"
              DW_AT_type: (reference to 8$)
              DW_AT_artificial: true
              DW_AT_location: DW_OP_fbreg(-24)
          DW_TAG_formal_parameter
              DW_AT_name [DW_FORM_string]: "p"
              DW_AT_decl_file: 2
              DW_AT_decl_line: 3
              DW_AT_type: (reference to 11$)
              DW_AT_location: DW_OP_fbreg(-32)
      ...
7$:   DW_TAG_base_type
          DW_AT_byte_size: 1
          DW_AT_encoding: DW_ATE_boolean
          DW_AT_name [DW_FORM_strx]: (slot 5) "bool"
      ...
8$:   DW_TAG_const_type
          DW_AT_type: (reference to 9$)
9$:   DW_TAG_pointer_type
          DW_AT_byte_size: 8
          DW_AT_type: (reference to 10$)
10$:  DW_TAG_const_type
          DW_AT_type: (reference to 3$)
      ...
11$:  DW_TAG_const_type
          DW_AT_type: (reference to 12$)
12$:  DW_TAG_reference_type
          DW_AT_byte_size: 8
          DW_AT_type: (reference to 13$)
13$:  DW_TAG_const_type
          DW_AT_type: (reference to 1$)
      ...
```

Figure F.6: Split object example: `demo1.dwo` DWARF excerpts *(concluded)*

The .debug_str_offsets.dwo section contains an entry for each unique string in the string table. Each entry in the table is the offset of the string, which is contained in the .debug_str.dwo section.

In a split DWARF object file, all references to strings go through this table (there are no other offsets to .debug_str.dwo in a split DWARF object file). That is, there is no use of DW_FORM_strp in a split DWARF object file.

The offsets in these slots have no associated relocations, because they are not part of a relocatable object file. When combined into a DWARF package file, however, each slot must be adjusted to refer to the appropriate offset within the merged string table (.debug_str.dwo). The tool that builds the DWARF package file must understand the structure of the .debug_str_offsets.dwo section in order to apply the necessary adjustments. Section F.3 on page 409 presents an example of a DWARF package file.

The .debug_rnglists.dwo section contains range lists referenced by any DW_AT_ranges attributes in the split DWARF object. In our example, demo1.o would have just a single range list for the compilation unit, with range list entries for the function Box::contains and for out-of-line copies of the inline functions Point::x and Point::y.

The .debug_loclists.dwo section contains the location lists referenced by DW_AT_location attributes in the .debug_info.dwo section. This section has a similar format to the .debug_loclists section in a non-split object, but it has some small differences as explained in Section 7.7.3 on page 226.

In demo2.dwo as shown in Figure F.7 on the next page, the debugging information for Line::clip starting at 2$ describes a local variable slope at 7$ whose location varies based on the PC. Figure F.8 on page 408 presents some excerpts from the .debug_info.dwo section for demo2.dwo.

part 1 of 2

```
1$: DW_TAG_class_type
        DW_AT_name [DW_FORM_strx]: (slot 20) "Line"
        DW_AT_signature [DW_FORM_ref_sig8]: 0x79c7ef0eae7375d1
        DW_AT_declaration: true
        ...
2$:     DW_TAG_subprogram
            DW_AT_external: true
            DW_AT_name [DW_FORM_strx]: (slot 19) "clip"
            DW_AT_decl_file: 2
            DW_AT_decl_line: 16
            DW_AT_linkage_name [DW_FORM_strx]: (slot 2) "_ZN4Line4clipERK3Box"
            DW_AT_type: (reference to DIE for bool)
            DW_AT_accessibility: DW_ACCESS_public
            DW_AT_declaration: true
        ...
```

Figure F.7: Split object example: `demo2.dwo` DWARF `.debug_info.dwo` excerpts

part 2 of 2

```
3$:   DW_TAG_subprogram
          DW_AT_specification: (reference to 2$)
          DW_AT_decl_file: 1
          DW_AT_decl_line: 3
          DW_AT_low_pc [DW_FORM_addrx]: (slot 32)
          DW_AT_high_pc [DW_FORM_data8]: 0x1ec
          DW_AT_frame_base: DW_OP_call_frame_cfa
          DW_AT_object_pointer: (reference to 4$)
4$:       DW_TAG_formal_parameter
              DW_AT_name: (indexed string: 0x11): this
              DW_AT_type: (reference to DIE for type const Point* const)
              DW_AT_artificial: 1
              DW_AT_location: 0x0 (location list)
5$:       DW_TAG_formal_parameter
              DW_AT_name: b
              DW_AT_decl_file: 1
              DW_AT_decl_line: 3
              DW_AT_type: (reference to DIE for type const Box& const)
              DW_AT_location [DW_FORM_sec_offset]: 0x2a
6$:       DW_TAG_lexical_block
              DW_AT_low_pc [DW_FORM_addrx]: (slot 17)
              DW_AT_high_pc: 0x1d5
7$:           DW_TAG_variable
                  DW_AT_name [DW_FORM_strx]: (slot 28): "slope"
                  DW_AT_decl_file: 1
                  DW_AT_decl_line: 5
                  DW_AT_type: (reference to DIE for type float)
                  DW_AT_location [DW_FORM_sec_offset]: 0x49
```

Figure F.7: Split object example: `demo2.dwo` DWARF `.debug_info.dwo` excerpts *(concluded)*

In Figure F.7 on page 406, the DW_TAG_formal_parameter entries at `4$` and `5$` refer to the location lists at offset `0x0` and `0x2a`, respectively, and the DW_TAG_variable entry for `slope` refers to the location list at offset `0x49`. Figure F.8 shows a representation of the location lists at those offsets in the `.debug_loclists.dwo` section.

Entry type		Range		Counted Location Description	
offset	(DW_LLE_*)	start	length	length	expression
0x00	start_length	[9]	0x002f	0x01	DW_OP_reg5 (rdi)
0x09	start_length	[11]	0x01b9	0x01	DW_OP_reg3 (rbx)
0x12	start_length	[29]	0x0003	0x03	DW_OP_breg12 (r12): -8; DW_OP_stack_value
0x1d	start_length	[31]	0x0001	0x03	DW_OP_entry_value: (DW_OP_reg5 (rdi)); DW_OP_stack_value
0x29	end_of_list				
——					
0x2a	start_length	[9]	0x002f	0x01	DW_OP_reg4 (rsi))
0x33	start_length	[11]	0x01ba	0x03	DW_OP_reg6 (rbp))
0x3c	start_length	[30]	0x0003	0x03	DW_OP_entry_value: (DW_OP_reg4 (rsi)); DW_OP_stack_value
0x48	end_of_list				
——					
0x49	start_length	[10]	0x0004	0x01	DW_OP_reg18 (xmm1)
0x52	start_length	[11]	0x01bd	0x02	DW_OP_fbreg: -36
0x5c	end_of_list				

Figure F.8: Split object example: `demo2.dwo` DWARF `.debug_loclists.dwo` excerpts

In each DW_LLE_start_length entry, the start field is the index of a slot in the `.debug_addr` section, relative to the base offset defined by the compilations unit's DW_AT_addr_base attribute. The `.debug_addr` slots referenced by these entries give the relocated address of a label within the function where the address range begins. The following length field gives the length of the address range. The location, consisting of its own length and a DWARF expression, is last.

F.3 DWARF Package File Example

A DWARF package file (see Section 7.3.5 on page 190) is a collection of split DWARF object files. In general, it will be much smaller than the sum of the split DWARF object files, because the packaging process removes duplicate type units and merges the string tables. Aside from those two optimizations, however, each compilation unit and each type unit from a split DWARF object file is copied verbatim into the package file.

The package file contains the same set of sections as a split DWARF object file, plus two additional sections described below.

The packaging utility, like a linker, combines sections of the same name by concatenation. While a split DWARF object may contain multiple `.debug_info.dwo` sections, one for the compilation unit, and one for each type unit, a package file contains a single `.debug_info.dwo` section. The combined `.debug_info.dwo` section contains each compilation unit and one copy of each type unit (discarding any duplicate type signatures).

As part of merging the string tables, the packaging utility treats the `.debug_str.dwo` and `.debug_str_offsets.dwo` sections specially. Rather than combining them by simple concatenation, it instead builds a new string table consisting of the unique strings from each input string table. Because all references to these strings use form DW_FORM_strx, the packaging utility only needs to adjust the string offsets in each `.debug_str_offsets.dwo` contribution after building the new `.debug_str.dwo` section.

Each compilation unit or type unit consists of a set of inter-related contributions to each section in the package file. For example, a compilation unit may have contributions in `.debug_info.dwo`, `.debug_abbrev.dwo`, `.debug_line.dwo`, `.debug_str_offsets.dwo`, and so on. In order to maintain the ability for a consumer to follow references between these sections, the package file contains two additional sections: a compilation unit (CU) index, and a type unit (TU) index. These indexes allow a consumer to look up a compilation unit (by its compilation unit ID) or a type unit (by its type unit signature), and locate each contribution that belongs to that unit.

For example, consider a package file, `demo.dwp`, formed by combining `demo1.dwo` and `demo2.dwo` from the previous example (see Appendix F.2 on page 396). For an executable file named "`demo`" (or "`demo.exe`"), a debugger would typically expect to find `demo.dwp` in the same directory as the executable file. The resulting package file would contain the sections shown in Figure F.9 on the next page, with contributions from each input file as shown.

Section	**Source of section contributions**

`.debug_abbrev.dwo`	`.debug_abbrev.dwo` from `demo1.dwo`
	`.debug_abbrev.dwo` from `demo2.dwo`

`.debug_info.dwo` (for the compilation units and type units)	compilation unit from `demo1.dwo` compilation unit from `demo2.dwo` type unit for class `Box` from `demo1.dwo` type unit for class `Point` from `demo1.dwo` type unit for class `Line` from `demo2.dwo`

`.debug_rnglists.dwo`	`.debug_rnglists.dwo` from `demo1.dwo`
	`.debug_rnglists.dwo` from `demo2.dwo`

`.debug_loclists.dwo`	`.debug_loclists.dwo` from `demo1.dwo`
	`.debug_loclists.dwo` from `demo2.dwo`

`.debug_line.dwo`	`.debug_line.dwo` from `demo1.dwo`
	`.debug_line.dwo` from `demo2.dwo`

`.debug_str_offsets.dwo`	`.debug_str_offsets.dwo` from `demo1.dwo`, adjusted `.debug_str_offsets.dwo` from `demo2.dwo`, adjusted
`.debug_str.dwo`	merged string table generated by package utility
`.debug_cu_index`	CU index generated by package utility
`.debug_tu_index`	TU index generated by package utility

Figure F.9: Sections and contributions in example package file `demo.dwp`

The `.debug_abbrev.dwo`, `.debug_rnglists.dwo`, `.debug_loclists.dwo` and `.debug_line.dwo` sections are copied over from the two `.dwo` files as individual contributions to the corresponding sections in the `.dwp` file. The offset of each contribution within the combined section and the size of each contribution is recorded as part of the CU and TU index sections.

The `.debug_info.dwo` sections corresponding to each compilation unit are copied as individual contributions to the combined `.debug_info.dwo` section, and one copy of each type unit is also copied. The type units for class `Box` and class `Point`, for example, are contained in both `demo1.dwo` and `demo2.dwo`, but only one instance of each is copied into the package file.

The `.debug_str.dwo` sections from each file are merged to form a new string table with no duplicates, requiring the adjustment of all references to those strings. The `.debug_str_offsets.dwo` sections from the `.dwo` files are copied as individual contributions, but the string table offset in each slot of those contributions is adjusted to point to the correct offset in the merged string table.

The `.debug_cu_index` and `.debug_tu_index` sections provide a directory to these contributions. Figure F.10 following shows an example CU index section containing the two compilation units from `demo1.dwo` and `demo2.dwo`. The CU index shows that for the compilation unit from `demo1.dwo`, with compilation unit ID `0x044e413b8a2d1b8f`, its contribution to the `.debug_info.dwo` section begins at offset 0, and is 325 bytes long. For the compilation unit from `demo2.dwo`, with compilation unit ID `0xb5f0ecf455e7e97e`, its contribution to the `.debug_info.dwo` section begins at offset 325, and is 673 bytes long.

Likewise, we can find the contributions to the related sections. In Figure F.8 on page 408, we see that the DW_TAG_variable DIE at `7$` has a reference to a location list at offset 0x49 (decimal 73). Because this is part of the compilation unit for `demo2.dwo`, with unit signature `0xb5f0ecf455e7e97e`, we see that its contribution to `.debug_loclists.dwo` begins at offset 84, so the location list from Figure F.8 on page 408 can be found in `demo.dwp` at offset 157 (84 + 73) in the combined `.debug_loclists.dwo` section.

Figure F.11 following shows an example TU index section containing the three type units for classes `Box`, `Point`, and `Line`. Each type unit contains contributions from `.debug_info.dwo`, `.debug_abbrev.dwo`, `.debug_line.dwo` and `.debug_str_offsets.dwo`. In this example, the type units for classes `Box` and `Point` come from `demo1.dwo`, and share the abbreviations table, line number table, and string offsets table with the compilation unit from `demo1.dwo`. Likewise, the type unit for class `Line` shares tables from `demo2.dwo`.

The sharing of these tables between compilation units and type units is typical for some implementations, but is not required by the DWARF standard.

Section header

Version:	5
Number of columns:	6
Number of used entries:	2
Number of slots:	16

Offset table

slot	signature	info	abbrev	loc	line	str_off	rng
14	`0xb5f0ecf455e7e97e`	325	452	84	52	72	350
15	`0x044e413b8a2d1b8f`	0	0	0	0	0	0

Size table

slot	info	abbrev	loc	line	str_off	rng
14	673	593	93	52	120	34
15	325	452	84	52	72	15

Figure F.10: Example CU index section

Section header

Version:	5
Number of columns:	4
Number of used entries:	3
Number of slots:	32

Offset table

slot	signature	info	abbrev	line	str_off
11	`0x2f33248f03ff18ab`	1321	0	0	0
17	`0x79c7ef0eae7375d1`	1488	452	52	72
27	`0xe97a3917c5a6529b`	998	0	0	0

Size table

slot	info	abbrev	line	str_off
11	167	452	52	72
17	217	593	52	120
27	323	452	52	72

Figure F.11: Example TU index section

(empty page)

Appendix G

DWARF Section Version Numbers (Informative)

Most DWARF sections have a version number in the section header. This version number is not tied to the DWARF standard revision numbers, but instead is incremented when incompatible changes to that section are made. The DWARF standard that a producer is following is not explicitly encoded in the file. Version numbers in the section headers are represented as two byte unsigned integers.

Table G.1 on the following page shows what version numbers are in use for each section. In that table:

- "V2" means DWARF Version 2, published July 27, 1993.
- "V3" means DWARF Version 3, published December 20, 2005.
- "V4" means DWARF Version 4, published June 10, 2010.
- "V5" means DWARF Version 5[1], published February 13, 2017.

There are sections with no version number encoded in them; they are only accessed via the `.debug_info` sections and so an incompatible change in those sections' format would be represented by a change in the `.debug_info` section version number.

[1]Higher numbers are reserved for future use.

Table G.1: Section version numbers

Section Name	V2	V3	V4	V5
`.debug_abbrev`	*	*	*	*
`.debug_addr`	-	-	-	5
`.debug_aranges`	2	2	2	2
`.debug_frame`[2]	1	3	4	4
`.debug_info`	2	3	4	5
`.debug_line`	2	3	4	5
`.debug_line_str`	-	-	-	*
`.debug_loc`	*	*	*	-
`.debug_loclists`	-	-	-	5
`.debug_macinfo`	*	*	*	-
`.debug_macro`	-	-	-	5
`.debug_names`	-	-	-	5
`.debug_pubnames`	2	2	2	-
`.debug_pubtypes`	-	2	2	-
`.debug_ranges`	-	*	*	-
`.debug_rnglists`	-	-	-	5
`.debug_str`	*	*	*	*
`.debug_str_offsets`	-	-	-	5
`.debug_sup`	-	-	-	5
`.debug_types`	-	-	4	-
(split object sections)				
`.debug_abbrev.dwo`	-	-	-	*
`.debug_info.dwo`	-	-	-	5
`.debug_line.dwo`	-	-	-	5
`.debug_loclists.dwo`	-	-	-	5
`.debug_macro.dwo`	-	-	-	5
`.debug_rnglists.dwo`	-	-	-	5
`.debug_str.dwo`	-	-	-	*
`.debug_str_offsets.dwo`	-	-	-	5

Continued on next page

[2]*For the* `.debug_frame` *section, version 2 is unused.*

Section Name	**V2**	**V3**	**V4**	**V5**
(package file sections)				
`.debug_cu_index`	-	-	-	5
`.debug_tu_index`	-	-	-	5

Notes:

- "*" means that a version number is not applicable (the section does not include a header or the section's header does not include a version).
- "-" means that the section was not defined in that version of the DWARF standard.
- The version numbers for corresponding .debug_<kind> and .debug_<kind>.dwo sections are the same.

(empty page)

Appendix H

GNU Free Documentation License

PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft," which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

H.1 APPLICABILITY AND DEFINITIONS

H.2 VERBATIM COPYING

H.5 COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section H.5 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History;” likewise combine any sections Entitled “Acknowledgements,” and any sections Entitled “Dedications.” You must delete all sections Entitled “Endorsements.”

H.6 COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

H.7 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation

H.8 TRANSLATION

H.9 TERMINATION

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

H.10 FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

H.11 RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a

ADDENDUM: How to use this License for your documents

(empty page)

Index